Optimization techniques for Craig Interpolant compaction in Unbounded Model Checking

G. Cabodi and C. Loiacono and D. Vendraminetto
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
Email: {gianpiero.cabodi, carmelo.loiacono, danilo.vendraminetto}@polito.it

***Abstract*—This paper addresses the problem of reducing the size of Craig interpolants generated within inner steps of SAT-based Unbounded Model Checking. Craig interpolants are obtained from refutation proofs of unsatisfiable SAT runs, in terms of and/or circuits of linear size, w.r.t. the proof. Existing techniques address proof reduction, whereas interpolant compaction is typically considered as an implementation problem, tackled using standard logic synthesis techniques. We propose an integrated three step process, in which we: (1) exploit an existing technique to detect and remove redundancies in refutation proofs, (2) apply combinational logic reductions (constant propagation, ODC-based simplifications, and BDD-based sweeping) directly on the proof graph data structure, (3) eventually apply ad hoc combinational logic synthesis steps on interpolant circuits. The overall procedure is novel (as well as parts of the above listed steps), and represents an advance w.r.t. the state-of-the art. The paper includes an experimental evaluation, showing the benefits of the proposed technique, on a set of benchmarks from the Hardware Model Checking Competition 2011.**

I. Introduction

Craig interpolants (ITPs for short) [1], [2], introduced by McMillan [3] in the Unbounded Model Checking (UMC) field, have shown to be effective on difficult verification instances. Though recently challenged by new techniques (IC3, Incremental Construction of Inductive Clauses for Indubitable Correctness [4]), our experience within the field of HWMCC competitions [5] and industrial cooperations shows that interpolation-based approaches still play an important role within a portfolio-based tool.

From a (high-level) Model-Checking perspective, Craig interpolation is an operator able to compute over-approximated images. The approach can be viewed as an iterative refinement of proof-based abstractions, to narrow down a proof to relevant facts. Over-approximations of the reachable states are computed from refutation proofs of (unsatisfied) BMC-like runs, in terms of AND/OR circuits, generated in linear time and space, w.r.t. the proof.

Craig interpolants' most interesting features are their completeness and the automated abstraction mechanism. Whereas one of their major challenges is the inherent redundancy of interpolant circuits, as well as the need for fast and scalable techniques to compact them. Improvements over the base method [3] were proposed in [6], [7], [8], [9] and [10], in order to push forward applicability and scalability of the technique.

Going to the details of interpolant computation, we need to consider SAT solvers and inner aspects of their deduction mechanism. An unsatisfiability proof is a series of applications of the resolution rule, i.e., if $(a \vee b) \wedge (\neg a \vee c)$ holds then we can deduce $b \vee c$. In the application of the rule, a is known as pivot. A resolution proof is generated by repeatedly applying the resolution rule on the clauses of an unsatisfiable Boolean formula, to deduce false. Modern SAT solvers (Boolean satisfiability checkers) implement some variation of DPLL [11], [12] that is enhanced with conflict driven clause learning [13].

Such solvers can produce resolution proofs as byproducts of unsatisfiable runs, with an almost negligible overhead [14]. Due to the nature of the algorithms employed by SAT solvers, a resolution proof may contain redundant parts. An extensive discussion on resolution proofs and their applications is beyond the scope of this paper. More details on proof transformations/reductions can be found in [15], [16], [17].

As our main purpose is to optimize interpolant generation, in this paper we address a more focused and specialized issue, than mere proof reduction.

- We start from low complexity algorithms for reduction of resolution proofs, as proposed in [16], but mere proof reduction is just a preliminary step,
- We further minimize a proof, taking into account interpolant generation. The resulting data structure is not an equivalent proof any more. we can rather consider it as an intermediate step between proof and interpolant.
- We finally apply combinational logic simplifications, targeted to highly redundant interpolant circuits.

Though mostly relying on known techniques, The overall approach is novel. This work also has a practical value, in its seek for a scalable, well balanced, and experimentally tuned approach to interpolant compaction.

Section II introduces background notions on notation, BMC and UMC, SAT-based Craig interpolant Model Checking, refutation proofs. Sections III and IV present our main contributions. Section V discusses the experiments we performed. Section VI concludes with some summarizing remarks.

II. Background

A. Model and Notation

We address systems modeled by labeled state transition structures and represented implicitly by Boolean formulas. From our standpoint, a system M is a triplet $M = (S, S_0, T)$, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is a total transition relation. The system

[1] This work was supported in part by SRC contract 2012-TJ-2328.
978-3-9815370-0-0/DATE13/©2013 EDAA

state space is encoded with an indexed set of Boolean variables $X = \{x_1, \ldots, x_n\}$, so that a state $s \in S$ corresponds to a valuation of the variables in X, and a set of states can be represented with a Boolean formula over X. A literal is a Boolean variable or its negation. A clause is a disjunction of literals. A CNF formula is a conjunction of clauses. Most modern SAT solvers [18], [19] adopt clauses as their main representation and manipulation formalism for Boolean functions. Whenever necessary, given a Boolean function F, we will use notation F_{CNF} for its CNF representation.

B. Bounded and Unbounded Model Checking

Given a sequential system M and an invariant property p, SAT-based BMC [20] is an iterative process to check the validity of p up to a given bound. To perform this task, the system's transition relation T is unrolled k times

$$T^k(X^{0..k}) \quad = \quad \bigwedge_{i=0}^{k-1} T(X^i, X^{i+1})$$

in order to implicitly represent all state paths of length k. BMC tools use SAT checks such as:

$$bmc^k(X^{0..k}) \quad = \quad S_0(X^0) \wedge T^k(X^{0..k}) \wedge \bigvee_{i=0}^{k} \neg p(X^i)$$

to look for counterexamples (of length $\leq k$), that start from the initial states S_0 and falsify p.

Though BMC tools are effective at finding bugs, their verification method is not complete. Therefore, specific techniques are required in order to support Unbounded Model Checking (UMC). The ability to check reachability fix-points and/or to find inductive invariants, is thus the main difference (and additional complication) between BMC and UMC.

C. Craig Interpolants

Let A and B be two inconsistent Boolean formulas, i.e., such that $A \wedge B \equiv \bot$. An ITP I for (A, B) is a formula such that: 1) $A \Rightarrow I$, 2) $I \wedge B \equiv \bot$, and 3) $supp(I) \subseteq supp(A) \cap supp(B)$.

```
INTERPOLANTMC (S_0, T, ¬p)
  k = 0
  do
    Cone_{0..k} = CIRCUITUNROLL(¬p, δ, k)
    res = FINITERUN (S_0, T, Cone_{0..k})
    k = k + 1
  while (res = undecided)

FINITERUN (S_0, T, Cone)
  if SAT(S_0 ∧ T ∧ Cone) return (reachable)
  R = S_0
  while (true)
    Image = ITP(R ∧ T, Cone)
    if (Image = undefined)
      return (undecided)
    if (Image ⇒ R) return (unreachable)
    R = R ∨ Image
```

Fig. 1. Interpolant-based Verification.

An interpolant $I = \text{ITP}(A, B)$ can be derived, as an AND/OR circuit, from the refutation proof of $A \wedge B$. McMillan [3] proposed an effective fully SAT-based Unbounded Model Checking algorithm, exploiting interpolants, as sketched in Figure 1.

While INTERPOLANTMC is the entry point of the algorithm, routine FINITERUN operates a forward traversal, where interpolation is used as an over-approximate image operator. The latter function may end up with three possible results:

- *reachable*, if it proves $\neg p$ reachable in k steps, hence the property has been disproved
- *unreachable*, if the approximate traversal using the IMG^{+}_{Adq} image computation reaches a fix-point. In this case the property is proved
- *undecided*, if $\neg p$ intersects the over-approximate state sets. Then, k in increased for a new FINITERUN call.

The previous algorithm is sound and complete [3].

D. From Resolution Proofs to Interpolants

Most modern SAT solvers rely on variants of the DPLL algorithm [12], where resolution is at the base of conflict driven learning. Given two clauses $C_1 = (l \vee l_1 \vee ... \vee l_n)$ and $C_2 = (\neg l \vee l'_1 \vee ... \vee l'_m)$, a resolution step is an application of the *binary resolution* inference rule

$$\frac{(l \vee l_1 \vee ... \vee l_n) \quad (\neg l \vee l'_1 \vee ... \vee l'_m)}{(l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_m)}$$

where variable l is called the *pivot* (or resolution) variable. The resolvent C is thus defined as: $C = Res(C_1, C_2) = (l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_m)$.

Craig interpolants are generated from resolution proofs using the following rules [3]. Let (A, B) be a pair of Boolean functions, and (A_{CNF}, B_{CNF}) the corresponding sets of CNF clauses. Let $\phi = A \wedge B = \bot$ (a SAT run on $\phi_{CNF} = A_{CNF} \cup B_{CNF}$ returns UNSAT). Let Π be a proof of unsatisfiability (a resolution proof) of ϕ_{CNF}. The proof Π can be represented by a directed acyclic graph (V_Π, E_Π), where V_Π is a set of vertices (nodes) and E_Π is a set of directed edges, such that:

- a unique leaf $v_\bot$ is associated to the empty clause $\bot$.
- root nodes are associated to original clauses in ϕ_{CNF}.
- each other vertex $v_i \in V_\Pi$ is associated to a resolution step. Let $Cl(v_i)$ denote the resolvent clause. The v_i vertex has exactly two predecessors $v_{i,0}$ and $v_{i,1}$ (connected by edges $(v_{i,0}, v_i)$ and $(v_{i,1}, v_i)$), such that $Cl(v_i)$ is the resolvent of $Cl(v_{i,0})$ and $Cl(v_{i,1})$.

Let us call *global variable*, a variable appearing both in A and B, and *global literal* a literal of a global variable. Non-global variables are called *local*. Let $c \in A_{CNF}$ be a clause, $g(c)$ is a clause generated by c by removing all non-global literals. Then the Π-interpolant of (A, B), or $\text{ITP}_\Pi(A, B)$, is defined as $p(v_\bot)$, with $p(v_i)$ a Boolean formula defined as

- if v_i is a root node, then $p(v_i) = g(v_i)$ (if $Cl(v_i) \in A_{CNF}$), otherwise $p(v_i) = \top$,
- else, let $v_{i,0}$, and $v_{i,1}$ be the predecessors of v_i in V_Π and let $pivot(v_i)$ be the pivot variable, then $p(v_i) = p(v_{i,0})\ OP\ p(v_{i,1})$, with OP being $\vee$ ($\wedge$), depending on $pivot(v_i)$ being local (non-local) to A.

We denote as **proof node chain** a linear tuple of V_Π nodes $Ch_k = (v_{k_1}, v_{k_2}, ..., v_{k_n})$, where all nodes (except the first one) have exactly one successor in V_Π. More formally, for each couple of adjacent nodes $v_{k_i}, v_{k_{i+1}}$ in Ch_k, they are connected by an edge $(v_{k_{i+1}}, v_{k_i}) \in E_\Pi$, and $v_{k_{i+1}}$ has no other successor in V_Π (*out-degree*$(v_{k_{i+1}}) = 1$). We call node v_{k_i} header of the chain ($Header(Ch_k)$). Node chains are explicitly available in SAT solvers such as Minisat [19], given the way refutation proofs are generated from conflict clauses. Though not strictly necessary for interpolant generation, we found them very useful for low cost identification/management of tree sub-graphs of proofs.

III. PROOF REDUCTION

The first part of our interpolant compaction approach relies on proof compaction techniques, that we split in two contributions. In Section (III-A) we discuss how to transform a given proof into a simpler one, that can still be considered as a proof of the original problem. More aggressive techniques, potentially losing the proof equivalence property (but guaranteeing equivalent interpolants) are described in Section III-B. As reduction procedures are visiting a proof graph from leaf ($v_\perp$) to roots, in the following we're using the transposed graph V_Π^T, instead of V_Π. We're also using node chains, as intermediate groups of nodes that help visiting the graph topology.

A. Equivalent Proof

We follow the RECYCLE-PIVOTS linear-time reduction of [16], whose main ideas can be summarized as follows:

- simplifications are done on tree sub-graphs, in order to keep linear complexity. let us denote it *proof sub-tree* (notice that trees are rooted towards leafs of the proof).
- given a proof sub-tree, a node is redundant if its pivot is found again on the (unique) path to the sub-tree root.

```
REDUCEPROOFREDUNDANTPIVOTS(V_Π^T)
  foreach node v_i ∈ V_Π^T following pre-order
    ReduceNode (v_i, ⊥)
  RestoreResGraph(V_Π^T)

REDUCENODE(v_i, recurred)
  if v_i is a chain header
    if recurred and in − degree(v_i) > 1 then return
    Mark v_i as visited
  if root then return
  piv = pivot(v_i)
  if implied piv then
    v_{i,0} = nil; ReduceNode (v_{i,1}, ⊤)
  else if implied ¬piv then
    v_{i,1} = nil; ReduceNode (v_{i,0}, ⊤)
  else
    imply piv; ReduceNode (v_{i,0}, ⊤); unimply piv
    imply ¬piv; ReduceNode (v_{i,1}, ⊤); imply ¬piv
```

Fig. 2. Proof reduction.

More in detail, given a resolution step that produces a clause using some pivot piv, the resolution step is called redundant if each deduction sequence from the clause to $v_\perp$ contains another resolution step with piv as pivot. A redundant resolution can easily be removed by local modifications in the proof structure. After removing the redundant resolution step, a strictly smaller proof is obtained. Detecting and removing all such redundancies in the most general case (a DAG-structured proof) is hard. RECYCLE-PIVOTS is a single pass algorithm that partially removes redundant resolutions. From each clause, the algorithm follows the deduction sequences to find equal pivots. The algorithm stops looking for equal pivots if it reaches a clause used in other deductions, i.e. it has more than one successor in V_Π.

Proof sub-trees are thus identified by considering *in-degree* of nodes in V_Π^T. Sub-trees of V_Π^T can be identified by DFS visits stopping at nodes with *in-degree* greater than 1.

Our procedure, though similar to RECYCLE-PIVOTS of [16] (and of same linear complexity), follows a two-level scheme, that allows us to use a simple data structure for sets of implied pivots. Whereas in [16] a single depth-first visit of V_Π^T is done, and several sets of implied pivots (one for each sub-tree) are dynamically created and updated.

Our strategy for sub-tree visit (function REDUCE-PROOFREDUNDANTPIVOTS) is based on first sorting nodes in pre-order, to be iteratively selected as roots of sub-trees, visited by function REDUCENODE. A single set of implied pivots is kept, encoded by an array of Booleans (one for each literal). Parameter $recurred$ is exploited to block tree visit at nodes with *in-degree* > 1, in all cases but the first (non recursive) call from REDUCEPROOFREDUNDANTPIVOTS. Function RestoreResGraph(V_Π^T) is finally called to clean-up the data structure after removals of redundant edges (and nodes).

B. Proof Reduction for Interpolants

In this second phase, we operate another set of simplifications that work on the proof graph, and reduce it, taking into account the corresponding interpolant circuit. Due to variable quantification on root nodes (clauses) of V_Π, and to transformation of internal nodes into AND/OR gates, we simplify the proof graph by applying combinational reductions: (forward) propagation of constants/equivalences, and (backward) propagation of Observability Don't Cares.

Forward propagation of constants and node equivalences is done by exploiting both (cheaper) structural equivalences and (more expensive) BDD-based sweeping.

The potentially high impact of constant propagation heavily relies on literal and/or full clause substitutions at root clauses (see Section II-D). Starting from root nodes, we propagate constants and equivalences to intermediate proof nodes. Due to non canonical representations, equivalences are enforced by (canonical) BDD-based sweeping [21]. As a cost-efficiency trade-off, we just store BDDs for chain header nodes, we disable dynamic reordering, and we adopt a threshold on BDD sizes. Whenever the BDD size of a node is greater than the threshold, BDD computation is disabled on node successors.

The procedure is represented in figure 3, where function PROPAGATEEQUIV is called for each node (in post-order), in order to forward propagate equivalences and implications.

```
REDUCEITPPROOF (V_Π^T)
  // init sweepHash and equiv classes
  foreach node v_i ∈ V_Π^T
    ldr[v_i] = v_i
  foreach node v_i ∈ V_Π^T following post-order
    PROPAGETEEQUIV (v_i)
  MERGEEQUIV(V_Π^T)
  foreach node v_i ∈ V_Π^T following pre-order
    ODCREDUCE (v_i, 0)
  RESTORERESITPGRAPH(V_Π^T)

PROPAGATEEQUIV(v_i)
  if v_i is root then compute p(v_i) with ITP rule
  else
    OP = choose between ∨ and ∧ with ITP rule
    const_i = propagateConstants (v_{i,0}, v_{i,1}, OP)
  if is constant then ldr[v_i] = const_i
  else
    // try computing BDD
    BDD_i = computeBddWithSizeThresh(v_{i,0}, v_{i,1}, OP)
    // Hash if chain header
    if not aborted and v_i is chain header then
      v_j = lookup(sweepHash,BDD_i)
    if not nil v_j then
      ldr[v_i] = ldr[v_j] // merge v_i to v_j

ODCREDUCE(v_i, recurred)
  if v_i is a chain header
    if recurred and in − degree(v_i) > 1 then return
    if visited then return
    Mark v_i as visited
  if root then return
  OP = choose between ∨ and ∧ with ITP rule
  v_0 = choose out of chain between v_{i,0} and v_{i,1}
  v_1 = choose other node between v_{i,0} and v_{i,1}
  if impliedNode v_0 then v_0 = nil // remove edge
  else
    ODCREDUCE (v_0, ⊤)
    if reduced to constant v_0 then handle constant
    IMPLYNODE (v_0)
    if IMPLIEDNODE (v_1) then v_1 = nil // remove edge
    else
      ODCREDUCE (v_1, ⊤)
      if reduced to constant v_1 then handle constant
    UNIMPLYNODE (v_0)
```

Fig. 3. Proof reduction taking into account interpolation.

After all propagations have been operated, function MERGEEQUIV (details not shown) is called, in order to re-build V_Π^T, taking into account previously found constants and equivalences. Equivalence classes are handled by the classical linear scheme, in which an element is considered as class leader, and all class elements point to it in the Ldr array.

On the other direction, we backward propagate Observability Don't Cares, using linear-time constant propagations, at each intermediate AND/OR node. Function ODCREDUCE is thus called in pre-order (from $v_\perp$ towards root clauses). A single array table is maintained, where each v_i node can be directly implied (implyNode), or unimplied (unimplyNode).

Whenever processing the generic v_i node, we consider successor nodes with *in-degree* > 1 as potential ODCs. Let's for instance suppose that v_i has two successors, named v_0 and v_1, and that the node is translated into an AND gate. Let us also suppose that *in-degree*$(v_0) > 1$. So we imply v_0 to constant value 1, throughout all recursive processing of node v_i. Implication is undone after recurring on v_1. The effect is to remove (by redirecting to constant) all other edges to v_0 that are potentially found in a proof sub-tree rooted at v_1.

IV. INTERPOLANT COMPACTION BY COMBINATIONAL SYNTHESIS

The last step of our reduction process is done on an AIG (And-Inverted Graph [22]) representation, i.e. a combinational circuit generated as described in II-D. We have now fully lost the (reduced) proof graph, so our purpose is to further simplify a Boolean function/circuit, taking into account that it can still be highly redundant, despite the simplifications done at previous steps.

```
COMBITPCOMPACT(ITP)
  for (i = 0; i < maxIter and enough progress; i++)
    do
      AIGBALANCERIGHT(ITP)
      AIGBALANCELEFT(ITP)
      AIGREWRITE(ITP)
      AIGBDDSWEEP(ITP)
    while enough progress
    if i < 2 then
      AIGITEDECOMPREC(ITE, maxRec, Unbalanced)
    else
      AIGITEDECOMPREC(ITE, maxRec, Balanced)

AIGITEDECOMPREC(F, maxRec, strategy)
  if –maxRec ≤ 1 or |ITP| < th then
    localOptimize(F)
    return
  do
    search node N_i with highest fan-out
    compute F_{N_i} and F_{¬N_i} cofactors
    if ACCEPTDECOMP(N_i, F_{N_i}, F_{¬N_i}, strategy) then
      AIGITEDECOMPREC(N_i, maxRec-1, strategy)
      AIGITEDECOMPREC(F_{N_i}, maxRec-1, strategy)
      AIGITEDECOMPREC(F_{¬N_i}, maxRec-1, strategy)
      F = ITE(N_i, F_{N_i}, F_{¬N_i})
      FASTOPTIMIZE(F)
    else
      exclude N_i from decomp candidates
  while not max try done
```

Fig. 4. Itp combinational compaction.

Our simplifications (see figure 4) repeatedly apply preliminary cheap reductions, based on AIG balancing (in two preferred directions) and light weight rewriting, followed by a BDD-based sweeping, more aggressive than the one described in section III-A, as (though still disabling dynamic reordering and using a size threshold) we compute BDDs for all internal AIG nodes and allow new variables at internal cut points. The above described simplifications are repeated as far as they are obtaining enough reduction. Then we operate an ITE-based decomposition (function AIGITEDECOMPREC) specifically thought for cases of large circuits (e.g. AIGs of size greater than 50000 gates). Our final simplification step is based on a more expensive procedure (LOCALOPTIMIZE) where we apply rewriting and refactoring from [23], [24], and, in case of limited set of support variables, BDD-based resynthesis. As we empirically observed that LOCALOPTIMIZE

is too expensive on large circuits, we recursively split it in sub-circuits, based on the ITE (If-Then-Else) decomposition. Given a function (circuit) F, we select an internal node with high fan-out, N_i, and express F in terms of N_i and of the two cofactors of F w.r.t. N_i:

$$F_{ITE} = ITE(N_i, F_{N_i}, F_{\neg N_i})$$

Function ACCEPTDECOMP filters out bad decompositions: a decomposition is accepted if the shared circuit size of F_{ITE} has an acceptable overhead w.r.t. the size of F, i.e. $|F_{ITE}| < \alpha|F|$ (typical values for α are in the range 1.1–1.2).

V. EXPERIMENTAL RESULTS

We implemented a prototype version of our methodology on top of the PdTRAV tool [25], a state-of-the-art verification framework which ranked within the top three tools (for single property, UNSAT category) in all past Model Checking Competitions [5]. The experimental data in this section are oriented to provide an evaluation of the techniques proposed in this paper, as well as a comparison with standard interpolation. Our experiments ran on a Quad-core workstation, with 2.5 GHz CPU frequency and 16 GB of main memory. We set time and memory limits to 1200 seconds and 2 GB, respectively.

We performed an extensive experimentation on a selected sub-set of 25 publicly available benchmarks from the HWMCC2011 [26] suite. We selected them by excluding problems that PdTRAV could solve in less than 1 minute (at the HWMCC2011), and those that we couldn't solve with any technique (including the one presented here). It is worth noticing that most of the selected benchmarks are from industrial origin (IBM, Intel).

Table I provides detailed data, showing (column Std Itp) the best results we could obtain, without the simplifications described in this paper. Columns Opt 1 through Opt 5, show an incremental application of our optimizations, starting from (Opt 1) pure combinational logic synthesis (see section IV), to various flavours of techniques in section III: pure proof reduction (Opt 2), incrementally complemented by constant propagation and structural equivalences (Opt 3), ODC-based simplifications, BDD-based sweeping (Opt 4 and Opt 5). Though we generally observed a lower memory usage in the optimized cases, we don't explicitly report memory data, as we consider them a secondary contribution, compared to overall run-time and experiment completion.

Albeit table I doesn't highlight a clear winner for all benchmarks, it clearly shows that Opt 4 and Opt 5 (the most complete techniques, differing only by a more aggressive search for equivalences) are more effective/robust overall.

The plots of figures 5 and 6 give a different view of the same results. Figure 5 plots run-times for individual optimization strategies, with different orders for each approach, in order to provide monotonic curves. It basically confirms that we extended the overall capability of our portfolio, with techniques Opt 4 and Opt 5 showing best performance.

Figure 6 provides a better comparison between best results we could obtain without and with any of the optimization proposed. Again, it's easy to observe that the improvement is not negligible.

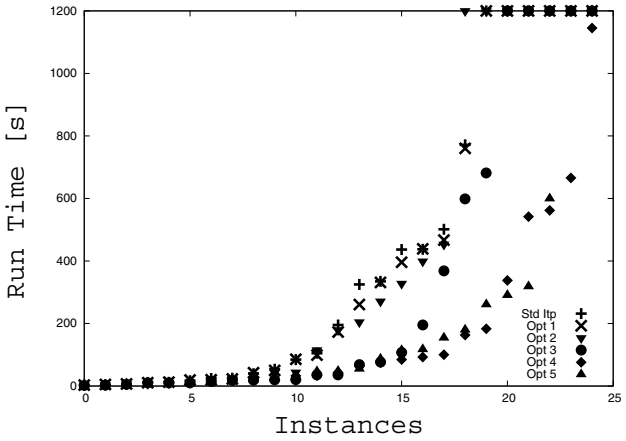

Fig. 5. Performance comparison between interpolation without and with various levels of the presented optimizations.

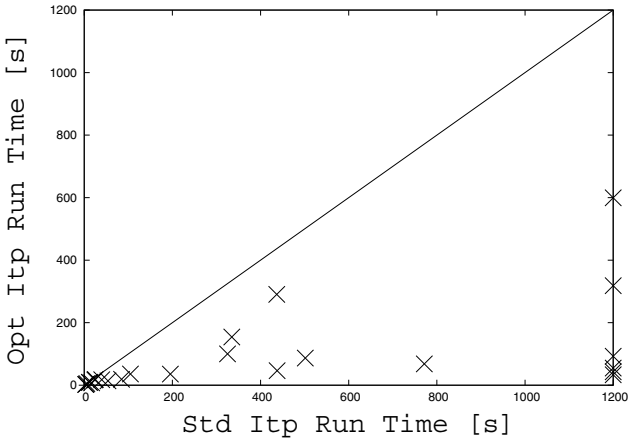

Fig. 6. Performance comparison between standard interpolation and optimized (best results for both approaches compared).

VI. CONCLUSIONS

We addressed the problem of optimizing interpolants for SAT-based Unbounded Model Checking. Our main contribution is to provide an integrated approach, that improves over standard interpolation, by reducing memory and time performance. Though we didn't target a new Model Checking algorithm, we experimentally observed that the proposed optimizations have improved both performance and scalability of our existing UMC approaches. Albeit we need to put some extra effort in a better engineering and overall integration of the proposed techniques, as well as more experimental work, we deem that present experimental data clearly witness the improvements attained.

REFERENCES

[1] W. Craig, "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory," *The Journal of Symbolic Logic*, vol. 22, no. 3, pp. 269–285, 1957.

[2] R. C. Lyndon, "An Interpolation Theorem in the Predicate Calculus," *Pacific Journal of Mathematics*, pp. 155–164, 1959.

[3] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[4] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, Austin, Texas, Jan. 2011, pp. 70–87.

Model				Std Itp	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5
Name	#PI	#FF	#AIG Node	Time	Time	Time	Time	Time	Time
6s2	856	781	13517	-	-	-	-	**92,74**	260,66
6s43	30	965	8440	-	-	-	-	541,79	**54,75**
6s49	17	180	1215	-	-	-	-	**33,4**	-
6s6	168	429	5132	501,43	438,76	398,98	368,24	337,72	**86,7**
6s9	252	607	16392	-	-	-	681,69	**42,85**	-
intel006	345	350	3259	3,96	5,13	3,86	**2,87**	3,08	3,56
intel007	1302	1307	13263	53,79	49,84	23,89	14,73	**13,93**	19,12
intel010	534	539	8985	324,97	260,73	271,17	107,69	**100,2**	114,08
intel011	528	533	8942	771,61	760,24	453,16	**68,05**	163,09	179,71
intel015	548	553	8488	436,76	467,08	-	598,36	561,93	**290,48**
intel018	486	491	6642	39,63	42,52	42,18	19,82	**17,54**	20,43
intel019	505	510	6907	84,63	85,8	35,91	18,86	**19,06**	26,42
intel020	349	354	5730	11,96	10,68	9,91	9,97	**8,83**	8,97
intel021	360	365	5877	16,72	19,15	**7,64**	10,11	9,84	15,8
intel022	525	530	8743	437,6	396,09	204,48	76,46	74,92	**46,35**
intel023	353	358	5721	195,31	172,88	115,16	**35,67**	63,9	47,54
intel024	352	357	5705	24,12	22,55	20,75	20,3	18,35	**16,82**
intel026	486	492	6258	11,65	12,56	11,91	10,41	9,51	**9,49**
intel029	559	564	8811	25,02	20,05	43,73	**18,26**	32,61	27,92
intel031	523	531	8795	104,79	99,18	179,12	**35,78**	84,53	117,24
intel054	567	572	5317	5,96	6,62	7,11	5,54	**5,34**	6,82
intel062	976	981	10286	334,69	331,47	327,94	195,77	182,98	**154,1**
intel063	288	293	2967	3,21	2,93	2,65	2,36	2,35	**2,24**
bobaesdinvdmit	516	1335	14697	-	-	-	-	1145,12	**599,44**
pdtswvsam6x8p4	9	128	8269	-	-	-	-	665,56	**318,23**

TABLE I. RESULTS ON SELECTED HWMCC2011 BENCHMARKS, COMPARING OUR BASIC VS. OPTIMIZED INTERPOLATION VERSIONS.

[5] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc."

[6] K. L. McMillan and R. Jhala, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 39–51.

[7] J. Marques-Silva, "Improvements to the implementation of Interpolant–Based Model Checking," in *Proc. Correct Hardware Design and Verification Methods*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 367–370.

[8] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Boosting Interpolation with Dynamic Localized Abstraction and Redundancy Removal," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 309–340, Jan. 2008.

[9] G. Cabodi, P. Camurati, and M. Murciano, "Automated Abstraction by Incremental Refinement in Interpolant-based Model Checking," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: ACM Press, Nov. 2008, pp. 129–136.

[10] B. Li and F. Somenzi, "Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3920, 2006, pp. 227–241.

[11] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[12] M. Davis, G. Logemann, and D. Loveland, "A Machine Procedure for Theorem-Proving," *Journal of the ACM*, vol. 5, pp. 394–397, 1962.

[13] J. P. Marques-Silva and K. A. Sakalla, "GRASP – A New Search Algorithm for Satisfiability," in *Int'l Conference on Tool with Artificial Intelligence*, 1996.

[14] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 10 880–.

[15] V. D'Silva, M. Purandare, and D. Kroening, "Approximation Refinement for Interpolation-Based Model Checking," in *Verification, Model Checking and Abstract Interpretation*, ser. Lecture Notes in Computer Science, vol. 4905. Springer, 2008, pp. 68–82.

[16] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman, "Linear-time reductions of resolution proofs," vol. 5394, pp. 114–128, 2009.

[17] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. Lecture Notes in Computer Science, vol. 5944. Springer, January 2010, pp. 129–145.

[18] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. 38th Design Automation Conf.* Las Vegas, Nevada: IEEE Computer Society, Jun. 2001.

[19] N. Eén and N. Sörensson, "The Minisat SAT Solver, http://minisat.se," Apr. 2009.

[20] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 317–320.

[21] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. 34th Design Automation Conf.* Anaheim, California: IEEE Computer Society, Jun. 1997, pp. 263–268.

[22] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proc. Design Automation Conf.* Las Vegas, Nevada: IEEE Computer Society, Jun. 2001.

[23] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression For Formal Verification," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California: IEEE Computer Society, Nov. 2004.

[24] R. K. Brayton and S. Chatterjee and A. Mishchenko, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. Design Automation Conf.*, 2006, pp. 532–536.

[25] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[26] A. Biere and K. Heljanko, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc11," 2011.