Reducing Interpolant Circuit Size by Ad Hoc Logic Synthesis and SAT-Based Weakening

Reducing Interpolant Circuit Size by Ad-Hoc Logic Synthesis and SAT-Based Weakening

G. Cabodi, P. E. Camurati, M. Palena, P. Pasini, D. Vendraminetto
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Turin, Italy
Email: {gianpiero.cabodi, paolo.camurati, marco.palena, paolo.pasini, danilo.vendraminetto}@polito.it

***Abstract*—We address the problem of reducing the size of Craig interpolants used in SAT-based Model Checking. Craig interpolants are AND-OR circuits, generated by post-processing refutation proofs of SAT solvers. Whereas it is well known that interpolants are highly redundant, their compaction is typically tackled by reducing the proof graph and/or by exploiting standard logic synthesis techniques. Furthermore, strengthening and weakening have been studied as an option to control interpolant quality.**

In this paper we propose two interpolant compaction techniques: (1) A set of ad-hoc logic synthesis functions that, revisiting known logic synthesis approaches, specifically address speed and scalability. Though general and not restricted to interpolants, these techniques target the main sources of redundancy in interpolant circuits. (2) An interpolant weakening technique, where the UNSAT core extracted from an additional SAT query is used to obtain a gate-level abstraction of the interpolant. The abstraction introduces fresh new variables at gate cuts that must be quantified out in order to obtain a valid interpolant. We show how to efficiently quantify them out, by working on an NNF representation of the circuit.

The paper includes an experimental evaluation, showing the benefits of the proposed techniques, on a set of benchmark interpolants arising from both hardware and software model checking problems.

I. Introduction

Craig interpolants (ITPs) [1], introduced by McMillan [2] in the Unbounded Model Checking (UMC) field, have shown to be effective on difficult verification instances.

From a Hardware 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 Bounded Model Checking–like runs, in terms of AND-OR circuits, generated in linear time and space, w.r.t. the proof.

From the perspective of Software Model Checking, instead, interpolants are used to strengthen the results of predicate abstraction [3]. In case the inductive invariant representing a program is insufficient to prove a given property, interpolants can be used as predicates to refine such an abstraction [4].

The most interesting features of Craig interpolants are their completeness and the fact can be used as an automated abstraction mechanism, whereas one of their major drawbacks 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 [2] were proposed in [5], [6], [7], [8] and [9], in order to push forward applicability and scalability of the technique.

Craig interpolants can be computed as AND-OR circuits, generated by post-processing refutation proofs of SAT solvers. Modern SAT solvers are capable, without incurring into large additional cost, to generate a resolution proof from unsatisfiable runs [10]. Due to the nature of the algorithms employed by SAT solvers, a resolution proof may contain redundant parts and a strictly smaller resolution proof can be obtained.

Although a Craig interpolant is linear in the proof size, the proof itself may be large and highly redundant. SAT solvers are not usually targeted to produce proofs of minimal size, therefore they may be deemed ultimately responsible for Craig interpolant size and redundancy. This is the main reason why most efforts on interpolant size reduction have been addressed as SAT solver improvement and/or proof reduction.

A. Contributions

In this paper we propose a fast and scalable logic synthesis approach, as well as a novel interpolant weakening (and strengthening) technique that also addresses circuit compaction. The main contributions are thus two interpolant compaction techniques:

- A set of ad-hoc logic synthesis functions specifically addressing speed and scalability. Though general and not limited to interpolants, they target the main sources of redundancy int interpolant circuits;
- An interpolant weakening technique, where an additional SAT query is performed in order to obtain a gate-level abstraction of the interpolant. Although fresh new variables are introduced at gate cuts, clearly outside the set of shared symbols, we show how to quantify them out by working on an NNF encoding of the circuit.

B. Related works

Interpolant compaction has been addressed in [11] and [12]. With respect to [11], we present additional techniques addressing scalability and interpolant compaction by weakening/strengthening. Interpolant weakening/strengthening is the subject of many papers, with little relation with our work. Among them, we consider [13] for an interesting discussion on the relationship between interpolant strength and quality.

The notion of dominance between nodes of a directed graph is central in this work. Dominators have been used in the context of logic synthesis before, such as [14], [15].

C. Outline

Section II introduces background notions and notation about Boolean circuits, Craig interpolants, gate-level abstraction and circuit compaction techniques. Section III describes the proposed ad-hoc logic synthesis functions, whereas our interpolant weakening technique is illustrated in Section IV. Section V presents and discusses the experiments we performed. Finally, Section VI concludes with some summarizing remarks.

II. Background

A. Combinational Boolean Circuits

Definition 1. *A* Boolean circuit *(or* network*) is a directed acyclic graph $\mathcal{G} = (V, E)$, where a node $v \in V$ represents either a* logic gate*, a* primary input *(PI) or a* primary output *(PO) of the circuit and each directed edge $(u, v) \in E$ represents a signal in the circuit connecting the output of node u to an input of node v. The fanin (fanout) of a node is the set of incoming (outgoing) edges of that node. Primary inputs are nodes with no fanin, whereas primary outputs are nodes with no fanout. Every logic gate $v \in V$ is associated with a Boolean function $f_v : \mathbb{B}^n \rightarrow \mathbb{B}$, where n is its number of inputs.*

The *fanin* (*fanout*) sets are typically represented by *lists*. With abuse of notation we use the terms fanin and fanout to identify both edges and the related sets of adjacent nodes. Given a gate node v, $type(v)$ is used to indicate the type of logic function associated with v (AND, OR, NOT, etc.).

Definition 2. *Given a circuit $\mathcal{G} = (V, E)$, a node u* dominates[1] *a node v iff every path from v to any of the primary outputs of $\mathcal{G}$ contains u. A node u that dominates a node v is called a* dominator *of v.*

Definition 3. *Given a circuit $\mathcal{G} = (V, E)$ and a node r, a* cone *$\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ rooted in r is a sub-graph of $\mathcal{G}$ consisting of r and some of its non–primary input predecessors such that any node in $\mathcal{C}$ has a path to r that lies entirely in $\mathcal{C}$. The fanin (fanout) of a cone is the number of nodes u not in $\mathcal{C}$ that are inputs (outputs) of a node t in $\mathcal{C}$.*

Node r is called *root* of the cone $\mathcal{C}$, and denoted by $root(\mathcal{C})$, non-root nodes of the cone are called *internal nodes*, whereas nodes in the fanin of the cone are called *cut nodes* of $\mathcal{C}$ and denoted by $cut(\mathcal{C})$. Nodes of $\mathcal{C}$ that have at least one cut node v in their fanin are called *entry points* in $\mathcal{C}$ for v. The Boolean function f_v associated with the cone root is called *cone function*. With abuse of notation we sometimes use $v \in \mathcal{C}$ to mean that $v \in V_{\mathcal{C}}$.

[1]Note that the notion of dominance as defined here corresponds to the dual notion of post-dominance from graph theory. For the sake of conciseness, we herein use the term dominance, with the definition provided above, to refer to the actual notion of post-dominance.

Definition 4. *A* cluster *is a cone $\mathcal{C}$ rooted in r such that, for each node v in $\mathcal{C}$, v has unit fanout and is dominated by r in $\mathcal{G}$.*

Note that cut nodes of a cluster $\mathcal{C}$ are either a PI or fanout branches, and the root r of $\mathcal{C}$ is either a PO or a fanout stem. Note also that the sub-graph of the circuit that defines a cluster $\mathcal{C}$ is a tree. Given a node $v \in \mathcal{C}$, every successor u of v in $\mathcal{C}$ is a dominator of v in $\mathcal{G}$.

Definition 5. *A* macrogate *is a cluster $\mathcal{M}$ such that every node v in $\mathcal{M}$ represents the same associative Boolean function. An* OR-macrogate *(*AND-macrogate*) is a macrogate composed of logical disjunction (conjunction) nodes.*

The definitions provided for cones are naturally extended to clusters and macrogates. An example of clusters and macrogates appears in Figure 1, where one cluster includes one OR- and two AND-macrogates.

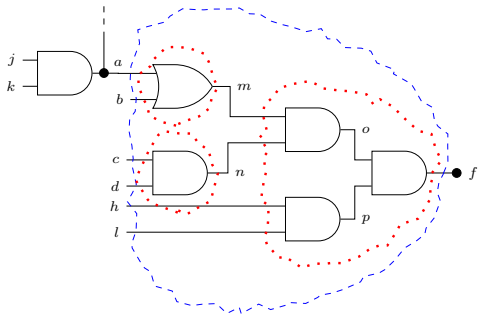


Fig. 1: A subcircuit partitioned in clusters (enclosed by a blue dashed line) and macrogates (enclosed by a dotted red line).

Definition 6. *Given a cone $\mathcal{C}$ rooted in r and a variable $a \in cut(\mathcal{C})$, variable a is not observable on f_r iff $f_r(X, \bot) \equiv f_r(X, \top)$, with $X = cut(\mathcal{C}) \setminus a$.*

A *literal* is either a Boolean variable or its negation. A *clause* is a disjunction of literals. A Boolean formula F is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses. Given a Boolean formula F, we denote with $supp(F)$ the set of Boolean variables over which F is defined.

A Boolean formula F is in *Negation Normal Form* (*NNF*) if the negation operator ($\neg$) is only applied to its variables, and the only other operators allowed are conjunction ($\wedge$) and disjunction ($\vee$). Any formula can be transformed to NNF in linear time through direct application of De Morgan's laws and the elimination of double negations. In the worst case, the size of the circuit implementing a formula F might double when F is transformed into NNF.

B. Craig Interpolants

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

We use ITP to denote the interpolation operation. An interpolant $I = \text{ITP}(A, B)$ can be derived, as an AND-OR circuit, from the refutation proof of $A \wedge B$. Most modern SAT solvers

are capable of producing resolution proofs. A resolution proof provides evidence of unsatisfiability for a CNF formula F as a series of applications of the *binary resolution* inference rule. 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 resolvent C is computed using a resolution operator, defined as: $C = Res(C_1, C_2) = (l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_m)$. Starting from the clauses of F, such a rule is applied until the empty clause is derived.

Craig interpolants are generated from resolution proofs as described in [2]. The resulting ITP circuit is isomorphic to the proof: where original clauses are translated as either OR gates or constants and resolutions steps are translated as either AND or OR gates. Interpolants in the range between A and $\neg B$ depend on SAT solver decisions, thus their resulting strength/weakness is not under user control. This motivated research on ex-post interpolant strengthening/weakening.

C. Combinational Circuit Compaction

This subsection briefly overviews, without any claim of completeness and generality, some combinational synthesis techniques our circuit compaction approach is based upon.

Redundancies affecting non canonical combinational circuits are removed by structural hashing, cut-based [16], BDD-based [17] and SAT-based [18] sweeping. The above methods basically rely on finding and merging classes of functionally equivalent circuit nodes. Other reduction efforts exploit various decomposition, rewriting and balancing strategies. In [19] a mix of locally canonical transformations and DAG-aware rewritings on technologically independent circuits have been first proposed. [14] introduces a technique for preprocessing combinational logic before technology mapping. We follow [14] in its use of And-Inverter Graphs (AIGs), composed of two-input ANDs and inverters[2]. Scalability is achieved by making all operations local, and moving to a global scope by iterated application of local reductions. The result is that the cumulative effect of several rewriting steps is often superior to traditional synthesis in terms of quality.

Redundancy removal under Observability Don't Cares (ODCs) is a powerful variant of redundancy removal, where node equivalences are established taking into account their observability at circuit outputs. All ODC-based approaches rely on a computation of don't care conditions for nodes involved in redundancy checks. As exact computation is prohibitively expensive, approximate techniques have been proposed. BDD-based Compatible Observability Don't Care (CODC) sets were computed in SIS [21]. Approximated ODCs (by "windowing") were introduced in [22], where scalability was achieved by restricting the sub-circuit environment to a locality. SAT-based quantifier elimination [23], augmented with random sampling, is a further attempt to exploit the power of SAT solvers.

D. Gate-Level Abstraction

Abstraction techniques are a well known area of research in Model Checking. Our paper is related to a form of localization abstraction [24] called Gate-Level Abstraction [25]. Abstraction by localization is based on removing circuit components (i.e. cutting wires) not necessary for a proof. Detection of unnecessary parts has been proposed following two main schemes:

- Counterexample-Based Abstraction-refinement (CBA) [26], where an initially weak abstraction is iteratively refined (strengthened) based on spurious counterexample analysis;
- Proof Based Abstraction (PBA), exploiting the ability of modern SAT solvers to generate proofs of unsatisfiability, is a more recently followed variant, investigated in stand-alone mode or combined with CBA, as in [27].

In most model checkers, localization is done at register boundaries. Gate-Level Abstraction [25] is a particular abstraction scheme (compatible in principle with both CBA and PBA strategies), where localization is done at gate nodes.

III. Interpolants Compaction by Ad-Hoc Logic Synthesis

In this section we present a set of procedures to reduce the size of Boolean circuits, based on local simplification techniques arising from logic synthesis. Although applicable to any Boolean circuit, our approach specifically targets the main sources of redundancy of interpolant circuits: gates that can be replaced by a constant value, or sub-circuits that can be merged being functionally equivalent (though topologically distinct). We consider an interpolant as a single-output circuit $\mathcal{G}$. Starting, from an AIG representation of the circuit, we:

- Identify AND and OR gates;
- Partition $\mathcal{G}$ into a set of maximal clusters;
- Group trees of AND (resp. OR) gates in macrogates.

Our target is to address gate redundancies by fast operations, where circuit transformations are performed within clusters. The reason for limiting our scope to clusters is related to the fact that fanout stems propagate *shared* subformulas through different paths within the circuit graph. Simplifications affecting multiple fanout paths are both complex and of limited impact.

The circuit $\mathcal{G}$ is partitioned into a maximal set of clusters, each of which is in turn partitioned into a set of macrogates. This is done by means of a depth-first visit of $\mathcal{G}$ starting from its root node r. Each node v is associated with two pieces of information: its cluster dominator, $domC(v)$, and its macrogate dominator, $domG(v)$. As long as the visited nodes have unit fanout, cluster dominator information in propagated. As long as the visited nodes have unit fanout and are of the same type, macrogate dominator information in propagated. Performing such an operation requires $O(|E|)$ time.

We thus propose a procedure based on two kinds of local simplifications:

- Redundancy removal (gates equivalent to a constant) based on ODC-like implications within clusters.
- Enforcement of sub-formula sharing (equivalent gates merging) through macrogate refactoring.

[2]Another motivation for our choice is the fact that AIGER is the netlist interchange format chosen for Hardware Model Checking Competitions [20].

A. ODC Implications Removal

The first simplification technique we propose aims at finding local ODC implications that can be exploited to replace a gate with a constant. Such a technique relies on the following two identities:

$$f(X,a) = a \wedge g(X,a) \equiv a \wedge g(X,\top)$$
$$f(X,a) = a \vee g(X,a) \equiv a \vee g(X,\bot)$$

Let us consider a Boolean function $f(X,a)$ expressed as the conjunction (resp. disjunction) of a variable a and a function g of a. Then a can be replaced by the $\top$ (resp. $\bot$) constant in g. Note that the instance of variable a in the support of g is not observable on f. From a circuit graph perspective, given $\mathcal{G}$ implementing f, a is an input variable and g is a subcircuit of $\mathcal{G}$ with a in its fanin. There are at least two re-convergent paths from node a to the output node of f.

We call such cases *ODC implications* for f, as the implications $f \rightarrow a$ and $\neg a \rightarrow \neg f$ (resp. $\neg f \rightarrow \neg a$ and $a \rightarrow f$) dually hold in each of the two respective cases.

We exploit the notion of ODC implications to perform local simplification of functions in the Boolean circuit. This is done by detecting cones $\mathcal{C}$ in the circuit whose function can be expressed as either $a \wedge g(X,a)$ or $a \vee g(X,a)$. In these cases, $\mathcal{C}$ can be simplified by disconnecting the redundant edge from a to its entry point in $\mathcal{C}$ and injecting a constant. Detection of ODC implications is restricted at macrogate and/or cluster boundaries in order to avoid problems arising from shared elements.

We consider both *direct ODC implications* and *transitive ODC implications*. Direct ODC implications arise when the input of a function f is directly implied by f. Figure 2 exemplifies a direct ODC implication. Input b is a direct ODC implication for f_t since $f_t(a,b,c) = b \wedge g(a,b,c)$ with $g(a,b,c) = c \wedge (a \vee b)$, and therefore $f_t \rightarrow b$. Transitive ODC implications occur when the input of a function f is transitively implied by f through another of its inputs. Figure 3 provides an example of transitive ODC implication. Input b is a transitive ODC implication for f_t, in fact, d is a direct ODC implication for f_t and b is a direct ODC implication of f_d, therefore, $f_t \rightarrow d \rightarrow b$.

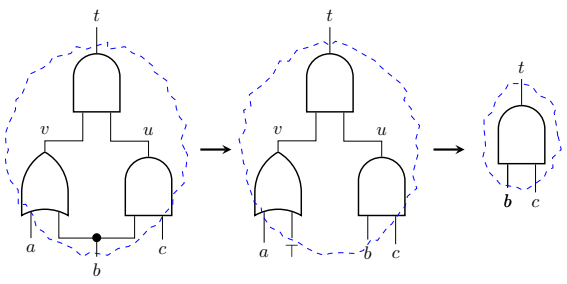


Fig. 2: Example of direct ODC implication.

The DIRECTODCSIMPLIFY procedure (Algorithm 1) tries to identify cluster inputs that are made redundant by direct ODC implications. Given a cluster $\mathcal{C}$ rooted in r and one of its inputs v, the algorithm tries to find a node d in $\mathcal{C}$ such that

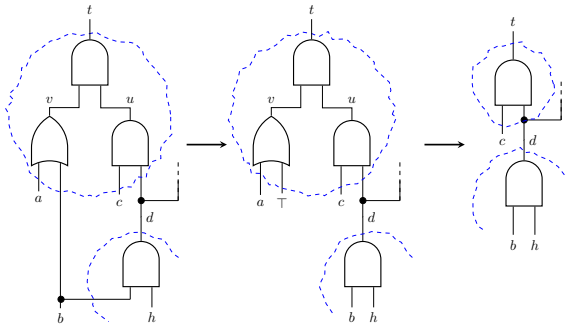


Fig. 3: Example of transitive ODC implication.

v is a direct ODC implication for f_d. Considering the cluster as a tree of macrogates, this corresponds to finding a common successor d for two of the entry points of v in $\mathcal{C}$, called u and t, so that d is a direct successor of either u or t. Since we are considering a tree of macrogates, d being a direct successor of t means that t is connected to d through either a chain of only AND or OR gates. For each cluster $\mathcal{C}_i$, the algorithm scans each of its cut nodes. For each $v \in cut(\mathcal{C}_i)$, every pair u,t of distinct entry points of v in $\mathcal{C}_i$ is considered. In order to find a common successor for u and t, first each macrogate dominator of u is marked by the procedure MARKDOMINATORS. Then, the algorithm checks if the macrogate dominator of t is marked. If that is the case, being $d = domG(t)$, we have either $f_d(X,v) = v \wedge g(X,v)$ or $f_d(X,v) = v \vee g(X,v)$ for some g. Therefore, v in g is not observable on f_d and the circuit can be simplified by calling function SIMPLIFY. Such a function takes a couple of nodes and a gate type as arguments, removes the edge (v,u) from the circuit and injects an appropriate constant value in the newly created free input. The injected constant is $\top$ if the gate type passed as argument is AND, $\bot$ if is OR. After injecting the constant, the circuit is simplified accordingly. Otherwise, if $domG(t)$ is not marked, the algorithm proceeds with the next pair of entry points. Time complexity of DIRECTODCSIMPLIFY is $O(|V| \max_{\mathcal{C}_i \in \mathcal{G}}\{|cut(\mathcal{C}_i)|\})$.

```
DIRECTODCSIMPLIFY(G)
 1: for all clusters C_i ∈ G do
 2:   for all nodes v in cut(C_i) do
 3:     for all pair (u,t) in fanout(v) ∩ C_i with u ≠ t do
 4:       MARKDOMINATORS(u)
 5:       if domG(t) is marked then
 6:         SIMPLIFY(v,u,type(t))
 7:       UNMARKDOMINATORS(u)
```

Algorithm 1. DIRECTODCSIMPLIFY*(G)*

The TRANSITIVEODCSIMPLIFY procedure (Algorithm 2) tries to identify cluster inputs that are made redundant by transitive ODC implications. Two lists are maintained for each cluster: a direct implication list and a transitive implication list. Given a cluster $\mathcal{C}$ rooted in r, its direct implication list, denoted as $Impl(\mathcal{C})$, contains all cluster inputs v for which at least one of the entry points of v in $\mathcal{C}$ has r as macrogate dominator. Therefore, for each $v \in Impl(\mathcal{C})$ either $f_r \rightarrow v$, if $type(r)$ is

AND, or $\neg f_r \rightarrow \neg v$, if $type(r)$ is OR. Direct implication lists are provided as an argument to TRANSITIVEODCSIMPLIFY. Transitive implication lists, denoted as $Trans(\mathcal{C})$, are used to collect those nodes v for which there exists a sequence of clusters $\mathcal{C}_0, \ldots, \mathcal{C}_n$ such that the following conditions hold:

- $\mathcal{C}_0 = \mathcal{C}$;
- $\mathcal{C}_{i+1} \in Impl(\mathcal{C}_i)$ for each $0 \leq i < n$;
- $type(\mathcal{C}_{i+1}) = type(\mathcal{C}_i)$ for each $0 \leq i < n$;
- $v \notin Impl(\mathcal{C}_i)$ for $0 \leq i < n$;
- $v \in Impl(\mathcal{C}_n)$.

Transitive implication lists are computed while TRANSITIVEODCSIMPLIFY runs and used to detect transitive ODC implications w.r.t. the root of each cluster.

In TRANSITIVEODCSIMPLIFY clusters are scanned in topological order. For each cluster $\mathcal{C}_i$, its transitive implication list is first computed. This is done by conjoining the current $Trans(\mathcal{C}_i)$ with every node that is either in the transitive or direct implication list of the clusters that are in $Impl(\mathcal{C}_i)$ and are of the same type of $\mathcal{C}_i$. Once the transitive implication list for $\mathcal{C}_i$ has been computed, the procedure scans each node $v \in cut(\mathcal{C}_i)$ that is in $Trans(\mathcal{C}_i)$. These nodes are inputs of $\mathcal{C}_i$ for which a transitive ODC implication exists (through some of the other inputs of $\mathcal{C}_i$). Therefore, each entry point u of these nodes can be simplified by calling SIMPLIFY. Time complexity of Algorithm 2 depends on the size of the transitive lists: $O(|V| \max_{\mathcal{C}_i \in \mathcal{G}}\{|Trans(\mathcal{C}_i)|\})$. Although the sizes of such lists, in the worst case, could be quadratic in the number of nodes, experimentally it is possible to notice that in our context of application the size of these lists stays within $O(|V|)$.

```
TRANSITIVEODCSIMPLIFY(G, Impl)
 1: for all clusters C_i ∈ G in topological order do
 2:   Trans(C_i) ← ∅
 3:   for all clusters C_k in Impl(C_i) do
 4:     for all v in Trans(C_k) ∪ Impl(C_k) do
 5:       if type(C_k) = type(C_i) then
 6:         Trans(C_i) ← Trans(C_i) ∪ {v}
 7:   for all nodes v in cut(C_i) do
 8:     if v in Trans(C_i) then
 9:       for all node u in fanout(v) ∩ C_i do
10:         SIMPLIFY(v, u, type(C_i))
```

Algorithm 2. TRANSITIVEODCSIMPLIFY*(*$\mathcal{G}$*, Impl)*

B. Macrogate Refactoring

The second simplification approach we propose tries to refactor portions of the circuit implementing the same type of Boolean function in order to explicit sub-functions implemented by nodes already present in the circuit. If successful, sharing can be enforced to reduce the overall size of the circuit. This technique is applied to macrogates in order to guarantee that each node removed by means of refactorization has unit fanout and thus the size of the circuit actually decreases. As an example, consider an AND-macrogate in Figure 4, implementing the function $f_t(a, b, c, d) = (a \wedge b) \wedge (c \wedge d)$. The idea is to identify a couple of inputs (i, j), such that the node realizing $i \wedge j$ does not appear in the macrogate but it exists in a different point of the circuit. Suppose a node m implementing $f_m = c \wedge b$ exists, the macrogate function f_t can be refactored as $f_t(a, b, c, d) = m \wedge (a \wedge d)$ so that the gate m can be shared. The final result of such a step of refactoring is a reparenthesization of the original macrogate function, for which the number of nodes decreases by one, one being now shared. A similar reasoning applies to OR-macrogates as well.

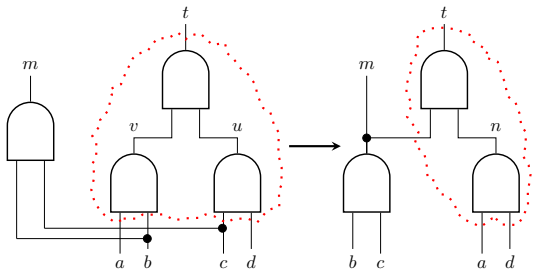


Fig. 4: Example of macrogate refactoring.

Note that refactoring a macrogate may change the current circuit partitioning as a previously non-shared node becomes shared.

The MACROGATEREFACTOR procedure (Algorithm 3) tries to refactor macrogates of the circuit in order to enforce better sharing. For each macrogate $\mathcal{M}_i$, first its cut nodes are marked. Then, for each input node of $\mathcal{M}_i$, the procedure scans all the nodes in its fanout list that do not appear in $\mathcal{M}_i$ but are of the same type. Those nodes u are gates of the same type of $\mathcal{M}_i$ that share an input with $\mathcal{M}_i$. For each of those nodes, the algorithm checks whether its other input node is shared with $\mathcal{M}_i$, by testing if such a node is marked. In such a case, $\mathcal{M}_i$ can be refactored to enforce sharing with u. Function REFACTOR handles macrogate refactoring. It also updates any other macrogate that could have been affected by the refactoring. Time complexity of MACROGATEREFACTOR is $O(|V| \max_{v \in V}\{|fanout(v)|\})$.

```
MACROGATEREFACTOR(G)
 1: for all macrogate M_i ∈ G do
 2:   Mark nodes in cut(M_i)
 3:   for all v in cut(M_i) do
 4:     for all u in fanout(v) do
 5:       if domG(v) ≠ domG(u) and type(v) = type(u)
          then
 6:         if left(u) ≠ v and left(u) is marked then
 7:           REFACTOR(M_i, u, left(u))
 8:         else if right(u) ≠ v and right(u) is marked then
 9:           REFACTOR(M_i, u, right(u))
10:   Unmark nodes in cut(M_i)
```

Algorithm 3. MACROGATEREFACTOR*(*$\mathcal{G}$*)*

IV. SAT-BASED WEAKENING

Previously described reductions follow the trend of fast circuit-based optimizations. We now present a novel approach combining the ideas of interpolant compaction and weakening.

Given an interpolant $I = \text{ITP}(A, B)$, a weaker (resp. stronger) interpolant I_w (resp. I_s) is another interpolant, such

that $I \rightarrow I_w$ ($I_s \rightarrow I$). Interpolant weakness and strength are dual concepts. Considering an interpolant I for A, B, its complement $\neg I$ is an interpolant for B, A. A weaker interpolant for A, B corresponds to a stronger interpolant for B, A. As mentioned in section I, interpolant strength and/or weakness can be related to the quality of the interpolant itself [13]. State-of-the-art approaches to interpolant strengthening/weakening are based on SAT proof transformations [28]. Interpolant re-computation is another straightforward and practical way to compact an interpolant and change its strength. Given $I = \text{ITP}(A, B)$, we can generate a weaker interpolant $I_w = \text{ITP}(I, B)$ or a stronger one $I_s = \text{ITP}(A, \neg I)$. Empirically, we spend extra time, performing an additional interpolant computation, in order to obtain a better interpolant, where *better* could mean weaker/stronger and possibly more compact. Unfortunately, compaction is not guaranteed, as the size of the final interpolant depends on a SAT solver run. Experimentally, we have observed both increases and decreases in terms of interpolant size.

Our strategy is to spend extra time by re-running a SAT solver query (either $A \wedge \neg I$ or $I \wedge B$), while computing the new interpolant in a different way, that guarantees compaction. In the following, we outline the main steps of our weakening approach (strengthening is dual):

- I is encoded as NNF, producing I_{NNF}
- A Gate-Level Abstraction of I_{NNF} is performed, using a PBA approach:
  - SAT query $I_{NNF} \wedge B$, guaranteed UNSAT, is solved and used to generate the UNSAT core $C(I_{NNF} \wedge B)$, the full proof is not necessary
  - Using the UNSAT core, a proof-based abstraction of I_{NNF} is computed: $I^{pba} = PBA(I_{NNF}, C)$
- As a result of PBA, fresh new variables Δ at all cut (abstraction) points are introduced. So, $supp(I^{pba}) = \Gamma \cup \Delta$, with $\Gamma = supp(A) \cap supp(B)$. The presence of these extra variables prevents I^{pba} from being a correct interpolant. Efficient existential quantification of Δ variables can be performed exploiting NNF encoding. In particular, $\exists \Delta\, I^{pba}$ is performed by replacing all variables in Δ with a $\top$ constant: $I_{w,NNF} = I^{pba}|_{\Delta=\{\top,\top,\ldots\top\}}$.
- The compacted interpolant $I_{w,NNF}$ is converted back to the (non NNF) AIG encoding.

Encoding a circuit as NNF implies a certain cost in terms of size. However, we experimentally observed (see section V) that this cost is negligible for interpolants, since they originate as pure AND-OR circuits with negations limited at input boundaries. Conversely, we have the advantage of *quantification by substitution*. Given a Boolean function $f(X, \Delta)$ in NNF form, with Δ appearing only in non-negated form, Δ can be existentially (resp. universally) quantified by substitution:

$$\begin{aligned} \exists \delta\, f(X, \Delta) &= f(X, \top) \\ \forall \delta\, f(X, \Delta) &= f(X, \bot) \end{aligned}$$

The top-level procedure is described in Algorithm 4. Given a node v, the function $CNF(v)$ is used to retrieve the CNF representation of f_v.

```
ITPWEAKEN(I, B)
 1: I_NNF ←AIG2NNF(I)
 2: C ← SATWITHUNSATCORE(I_NNF ∧ B)
 3: for all nodes v in I_NNF do
 4:    if CNF(v) ∉ C then
 5:       REPLACE(v, ⊤)
 6: I_w,NNF ← RECOMPUTECIRCUIT(I_NNF)
 7: I_w ←NNF2AIG(I_w,NNF)
Return I_w
```

Algorithm 4. ITPWEAKEN*(I, B)*

The algorithm shows weakening of I w.r.t. B, being strengthening with A dual. Furthermore, we use PBA-based abstraction, whereas a CBA-based approach is possible as well. The proposed code unifies GLA (Gate-Level Abstraction) with existential quantification, as, given the UNSAT core (C), circuit nodes with a corresponding CNF variable not in C are immediately abstracted and replaced with the $\top$ constant.

V. EXPERIMENTAL RESULTS

We implemented a prototype version of our interpolant compaction procedures on top of the PdTRAV tool [29], a state-of-the-art verification framework. Experimental data in this section provide an evaluation of the techniques proposed. Experiments were run on an Intel Core $i7-3770$, with 8 CPUs running at 3.40 GHz, 16 GBytes of main memory DDR III 1333, and hosting a Ubuntu 12.04 LTS Linux distribution. We set memory limits to 900 seconds (3600 for the weakening experiments) and 8 GB, respectively.

We performed an extensive experimentation on a selected subset of interpolants used in [11]. These interpolants are extracted from publicly available benchmarks from the past HWMCC [20] suites and are represented as AIGs. We took into account also interpolants derived from software verification problems [12]. The former set is composed of 2472 instances, ranging from 1.1×10^5 to 8.5×10^6 nodes. The latter set is composed of 1872 instances, ranging from 4×10^2 to 6×10^4 nodes[3].

We gathered initial data from the first set of interpolants in order to purge *easy* instances. We considered easy those instances with less than 1.5×10^4 nodes and for which our logic synthesis procedure was able to reach a fix-point within 150 seconds. The purged set of benchmarks, comprising 87 instances ranging from 4×10^5 to 8.5×10^6 nodes, was used to conduct a more in-depth experimentation.

Figures 5 and 6 show the results obtained for compaction with logic synthesis (section III) and GLA-based weakening (section IV), respectively. Compaction techniques are applied incrementally, i.e., we always apply simplifications described in [11][4], followed by the techniques described in this paper.

[3]The interpolant circuits are available at http://fmgroup.polito.it/index.php/download.

[4]With the exception of the most time-consuming, and less scalable, ITE-based decomposition.

A. Compaction by Logic Synthesis

In our experiments, we evaluated techniques of section III by applying them as follows. First the circuit is partitioned into clusters and macrogates. A trivial simplification is performed by removing each duplicated input from macrogates. Then DIRECTODCSIMPLIFY, MACROGATEREFACTOR and TRANSITIVEODCSIMPLIFY are iterated in this order, recomputing the circuit partition between each call, until two consecutive iterations reduce the circuit size for less than 1%.

For each benchmark, we first apply the AIG balancing procedure of ABC prior to applying any of the aforementioned techniques. We consider the size of interpolants after balancing as baseline for the following experimentation. In order to test individual contributions of the proposed techniques we performed an initial run with all simplifications enabled, we call this run ITPSIMPLIFY, followed by a set of runs in which we selectively disabled them one at a time: NODIRECTODCSIMPLIFY, NOMACROGATEREFACTOR and NOTRANSITIVEODCSIMPLIFY respectively. As a last test, we disabled our techniques altogether and performed ITP compaction using only standard logic synthesis (rewriting/refactoring, using the state-of-the-art ABC [30] tool).

Figures 5a and 5b illustrate the cumulative size and execution time, respectively, over all the benchmarks. In both cases, the closer a line is to the x axis, the better the result.

The two figures easily illustrate the compromise between execution time and potential size reduction obtained. On the one hand the purely ABC-based simplification is the best performing one, but it requires a significant amount of time. Different compaction rates are achievable with less computational effort adopting less aggressive approaches. We excluded timeouts from the visual representation.

As mentioned in section II-D, the size of implication lists could be a limit to the scalability of the proposed methods, as well. Although such lists could theoretically grow quadratically in the number of nodes, experimentally we noticed at worst a multiplicative factor of 20.

B. Compaction by Weakening

In order to characterize the rate of ITP compaction achievable through SAT-based weakening/strengthening, we raised the time limits to 3600 seconds. Such an approach is conceived to be used when ITP size reduction is crucial, and/or weakening/strengthening are actually the target, which motivates a bigger effort in terms of total execution time.

A preliminary step for all the proposed techniques requires to convert a given interpolant into NNF form. This step could lead to an increase in circuit size up to a factor of 2, in the general case. Given the nature and structure of interpolants themselves the increase in size is almost negligible. Taking into account all the experiments conducted, the biggest experienced increase was below 0.5%, confirming the intuitive arguments in section IV.

We conducted a set of experiments taking into account the same subset of 87 interpolants, iterating sequences of weakening (labelled B) and/or strengthening (labelled A) steps in different patterns. We propose an experimental evaluation for six different sequences: A, B, AB, BA, $ABAB$ and $BABA$. We run our logic synthesis compaction procedure before any weakening/strengthening attempt (baseline). Figures 6a and 6b illustrate the cumulative size and execution time, respectively, over all the benchmarks. It is fairly noticeable the impact on the choice of the first kind of chosen compaction: starting with B tends to produce better results, related to the fact that most of the interpolants proposed have more room for weakening than strengthening.

Overall, it is fairly clear that SAT-based abstraction leads to dramatic compaction, though paid in terms of time.

VI. CONCLUSIONS

We addressed the problem of optimizing interpolants size for SAT-based UMC. Our main contribution is to provide an integrated approach, that targets interpolation compaction, providing different tradeoffs between time and memory according the proper context of application. We work both at the level of logic synthesis and at SAT level, proposing different techniques aimed at interpolant size reduction. Overall, our main target is to increase the scalability of existing UMC approaches, taking into account resource limitations and compromising between optimal results and applicability of the proposed methods. We experimentally observed that the proposed optimizations can be beneficial to existing reachability schemes, based on interpolation.

VII. ACKNOWLEDGEMENTS

We thank prof. Natasha Sharygina, dr. Antti E. J. Hyvärinen and Leonardo Alt from Università della Svizzera Italiana (USI), Switzerland, for the benchmarks generated from software verification problems.

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] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV*, ser. LNCS, vol. 2725, Boulder, USA, 2003, pp. 1–13.

[3] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *Proc. of CAV*, London, UK, UK, 1997, pp. 72–83.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," *SIGPLAN Not.*, vol. 39, pp. 232–244, Jan. 2004.

[5] J. Marques-Silva, "Improvements to the implementation of Interpolant–Based Model Checking," in *Proc. of CHARME*, ser. LNCS, vol. 3725. Edinburgh, Scotland, UK: Springer, 2005, pp. 367–370.

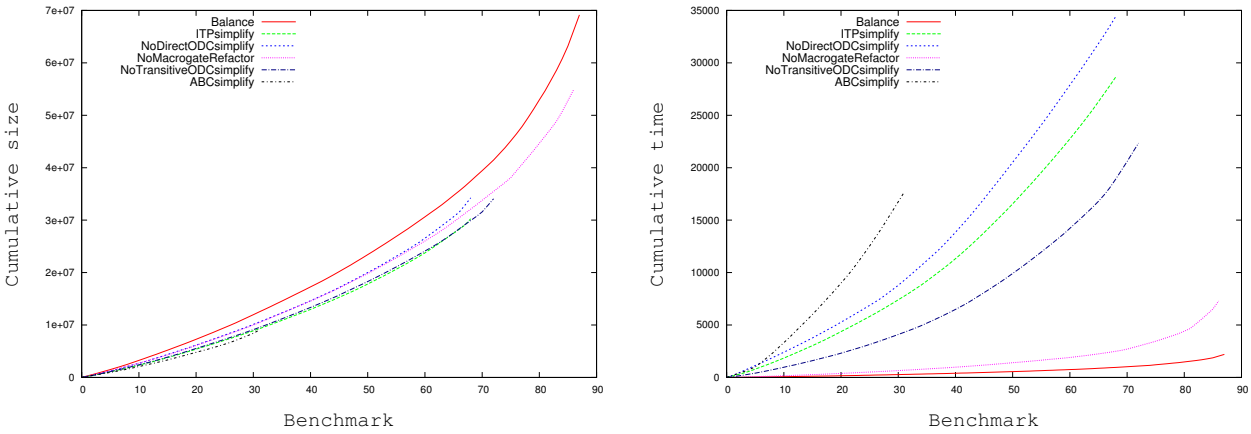
[6] V. D'Silva, M. Purandare, and D. Kroening, "Approximation Refinement for Interpolation-Based Model Checking," in *Verification, Model Checking and Abstract Interpretation*, vol. 4905, 2008, pp. 68–82.

[7] 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.

[8] G. Cabodi, P. Camurati, and M. Murciano, "Automated Abstraction by Incremental Refinement in Interpolant-based Model Checking," in *Proc. of ICCAD*. San Jose, California: ACM Press, Nov. 2008, pp. 129–136.

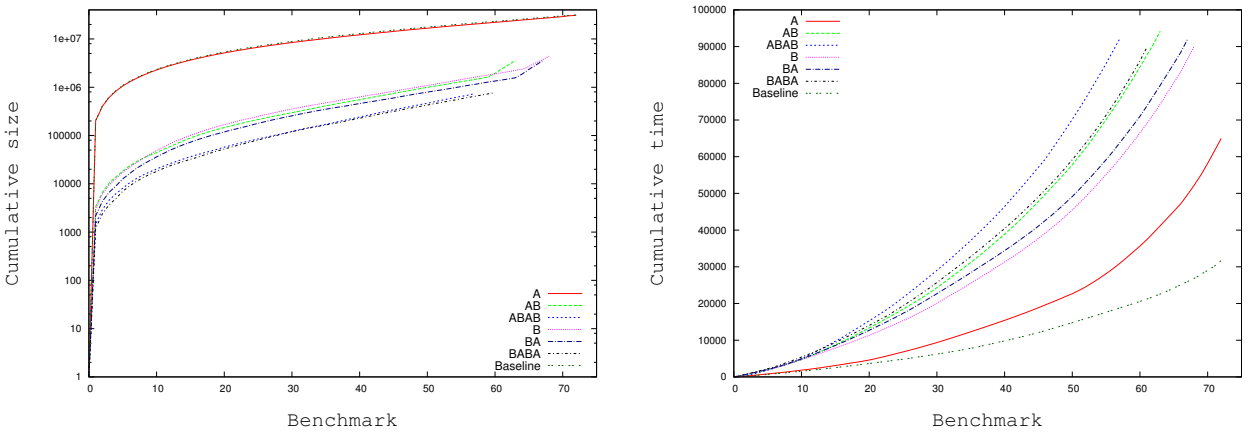
[9] B. Li and F. Somenzi, "Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3920, 2006, pp. 227–241.

[10] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proc. of DATE*, Washington, DC, USA, 2003.



(a) (b)

Fig. 5: Cumulative results of ITP compaction based on logic synthesis, in terms of size and execution time.



(a) (b)

Fig. 6: Cumulative results of ITP compaction based on SAT, in terms of size and execution time. Sizes are plotted on a log scale given the higher ratio of compaction achieved.

[11] G. Cabodi, C. Loiacono, and D. Vendraminetto, "Optimization techniques for craig interpolant compaction in unbounded model checking," *Formal Methods in System Design*, vol. 46, no. 2, pp. 135–162, 2015.

[12] L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "A proof-sensitive approach for small propositional interpolants," in *Verified Software: Theories, Tools, and Experiments - Revised Selected Papers*, San Francisco, CA, USA, Jul. 2015, pp. 1–18.

[13] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *Proc. of VMCAI*, vol. 5944, January 2010, pp. 129–145.

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

[15] D. B. neres, J. Cortadella, and M. Kishinevsky, "Dominator-based partitioning for delay optimization," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '06. New York, NY, USA: ACM, 2006, pp. 67–72.

[16] N. Eén, "Cut Sweeping," Cadence Research Labs, Berkeley, USA, Tech. Rep., May 2007.

[17] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. of DAC*, Anaheim, California, Jun. 1997, pp. 263–268.

[18] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Property Checking," in *Proc. of ICCAD*, San Jose, California, Nov. 2004, pp. 50–57.

[19] P. Bjesse and A. Boralv, "DAG-Aware Circuit Compression For Formal Verification," in *Proc. of ICCAD*, San Jose, California, Nov. 2004.

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

[21] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization." in *Proc. of ICCAD*, 1991, pp. 514–517.

[22] A. Mishchenko and R. K. Brayton, "Sat-based complete don't-care computation for network optimization," *CoRR*, vol. abs/0710.4695, 2007.

[23] K. L. McMillan, "Applying sat methods in unbounded symbolic model checking." in *Proc. of CAV*, vol. 2404, 2002, pp. 250–264.

[24] R. P. Kurshan, "Computer Aided Verification of Coordinating Processes," in *Princeton University Press*, Princeton, NJ, 1994.

[25] A. Mishchenko, N. Een, R. Brayton, J. Baumgartner, H. Mony, and P. Nalla, "Gla: Gate-level abstraction revisited," in *Proc. of DATE*, ser. DATE '13, San Jose, CA, USA, 2013, pp. 1399–1404.

[26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. of CAV*, 2000, pp. 154–169.

[27] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental sat formulation of proof- and counterexample-based abstraction," in *Proc. of FMCAD*, Oct 2010, pp. 181–188.

[28] K. L. McMillan and R. Jhala, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV*, ser. LNCS, vol. 3725, Edinburgh, Scotland, UK, 2005, pp. 39–51.

[29] 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.

[30] R. K. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.