Interpolant Generation without Constructing Resolution Graph

Chih-Jen Hsu, Shao-Lun Huang, Chi-An Wu and Chung-Yang (Ric) Huang
Graduate Institute of Electronics Engineering, National Taiwan University, Taiwan

ABSTRACT

In this paper, we proposed a novel interpolant generation algorithm without constructing the resolution graph of the unsatisfiability proof. Our algorithm generates the interpolant by building sub-interpolants from conflict analyses and then merges them based on the last decision conflict. The experimental results show that our algorithm has the advantages over the prior interpolant generation techniques in both memory usage and interpolation circuit size.

1. INTRODUCTION

Given two inconsistent Boolean formulae A and B ($A \wedge B = 0$)[1], we can find an interpolant I of A and B such that $A \rightarrow I$ and $I \wedge B = 0$. As shown in Figure 1, an interpolant I will be a formula that covers A and disjoins B. Obviously, there must exist such an I and it is generally not unique. Any formula that is "larger" (in terms of Boolean space) than A and "smaller" than B is a valid interpolant.

The Craig's Interpolation proposed in [1] showed that we can find an interpolant which is composed by the common variables of A and B only. Pudlák [2] and McMillan [3] further proved that the Craig interpolant can be derived in linear time complexity from the resolution graph of the unsatisfiability proof. Since then, interpolation has become a very popular technique and many researchers have successfully utilized it to push the envelope in various areas. For example, the authors in [4] and [5] have applied interpolation to improve the identification of the functional dependency and the computation of functional decomposition. Other applications of the interpolation technique also include constraint solving [6] and software verification [7][8], etc.

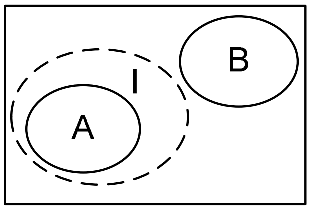
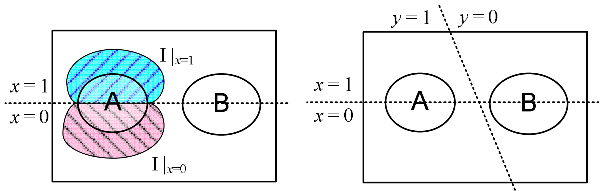


Figure 1: The conceptual illustration of interpolation

[1] In this paper, we will use the symbols $\wedge$, $\vee$ and $\neg$ for logical AND, OR and INVERSION, 0 and 1 for false and true, and $\rightarrow$ for implication, respectively.

However, current interpolation algorithms are very memory-consuming and unpredictable. In addition, the size of the interpolant can be much bigger than the original function. This is mainly due to the inefficient construction and representation of the interpolant function.

In this paper, we proposed a novel interpolant generation algorithm which *does not require the recording of the resolution graph*. An intuitive and high-level explanation of our idea can be illustrated as in Figure 2, where x and y are two decision variables in the SAT process. Intuitively, the interpolant I is the disjunction of $I|_{x=1}$ and $I|_{x=0}$, respectively (Figure 2(a)). As the SAT proof goes on, the search space will be further divided into smaller subspaces until the terminal case occurs[2]. As shown in Figure 2(b), the corresponding sub-interpolant for the terminal cases can be easily derived as 1 (e.g. for $x = 1$ and $y = 1$) or 0 (e.g. for $x = 1$ and $y = 0$). As the different subspaces are exploited by the decision procedure, the sub-interpolants can be computed with respect to the learned clauses. Finally, we can conclude the complete interpolant when the SAT returns unsatisfiability.



(a) One decision variable *x* **(b) Two decisions *x* and *y***

Figure 2: An intuitive and high-level illustration of our interpolant generation algorithm

The remainder of the paper is organized as follows. In Section 2, we present our algorithm and the major theorems behind them. The experimental results will be shown in Section 3, and Section 4 concludes the paper.

2. OUR INTERPOLANT GENERATION ALGORITHM

In this section, we will formally define the meaning of the partial interpolants with respect to the cofactor subspaces and demonstrate the algorithm to construct the complete interpolant from them. We will further show that we can derive the interpolant from the implication graph during the SAT conflict analysis process.

2.1 Overview of Our Algorithm

We proposed an interpolant generation algorithm which can be seamlessly integrated with the modern SAT solvers. Figure 3 illustrates our algorithm in pseudo code. This modified procedure, *InterpolantConflictAnalysis*, is called whenever a

[2] That is, some clause in A or B becomes false, which makes A or B false (empty set in the subspace).

decision conflict arises. It will terminate the proof process and return "*unsatisfiable*" when the conflict occurs in decision level 0 (i.e. the same level with the satisfiability target).

Our Interpolant Generation Algorithm

```
1.  InterpolantConflictAnalysis ( $A$, $B$ )
2.    Var $x$;  Clause $c_{imp}$, $c_{learn}$;
3.    initial($x$, $c_{imp}$, $c_{learn}$);
4.    Interpolant $I_{curr}$ = initInterpolant($c_{imp}$);
5.    repeat:  // imp nodes in the last decision level
6.      if (checkTerminatingCond($x$) is true) do
7.        recordInterpolant($c_{learn}$, $I_{curr}$);
8.        return ($c_{learn}$, $I_{curr}$);
9.      end if
10.     $x$ = $x$.prevNodeInImpGraph();
11.     $c_{imp}$ = clause that triggers implication x;
12.     if ($x$.isConflictRelated() is false)
          goto repeat;
13.     Interpolant $I_{imp}$ = getInterpolant($c_{imp}$);
14.     $I_{curr}$ = computeInterpolant($x$,$c_{imp}$,$c_{learn}$,$I_{imp}$,$I_{curr}$);
15.     $c_{learn}$ = updateLearnCls($x$, $c_{imp}$, $c_{learn}$);
16.   end repeat
```

Figure 3: Our interpolant generation algorithm

In the initialization step of line 3, x is set to the last implied variable of the last decision level, that is, the conflict variable. The clause c_{imp} is the clause that produces the implication on x, and the clause c_{learn} is initialized to 0.

I_{curr} is the current interpolant that is iteratively computed in the "repeat" loop (lines 5 to 16). It is initialized by the routine "*initInterpolant*()", which will be described at the end of the next subsection.

In the beginning of the repeated loop, we first check if the current variable x meets the terminating condition (line 6). That is, either it is the UIP variable in the conflict analysis, or we have encountered the conflict at decision level 0 (i.e. the last decision conflict). In the former case, we will record the partial interpolant (I_{curr}) with respect to the learned clause (c_{learn}) and exit the loop. In the latter case, we will traverse all the way to the beginning of the satisfiability target and construct the final interpolant.

In lines 10 and 11, x is updated to the previous node in the implication graph and c_{imp} is now the clause that triggers the implication. If x is not related to the conflict (i.e. not a implication source to the conflict), the algorithm will go to line 5 (repeat) for the next x. Otherwise, a sub-interpolant for the implication clause c_{imp} is generated and the current interpolant I_{curr} can be computed from the resolution operation on c_{imp} and the current learned clause c_{learn}. We will present the detail algorithms and theorems behind them in the coming subsections.

2.2 Interpolant under Cofactor Subspace

In this subsection we will introduce a novel view in explaining the partial interpolants created during the conflict analysis process.

A (positive) cofactor of a function f with respect to a variable x, denoted as $f\,|_x$, is a new function obtained by setting x to 1 in f. It can also be treated as the projection of f on the "x = 1" hyper plane of the Boolean space, or say, the projection of f under the "x = 1" cofactor subspace.

Likewise, we define the interpolant of two Boolean formulae under the cofactor subspace as:

[Definition 1] Interpolant under cofactor subspace
A formula is an interpolant of A and B under the "x = 1" cofactor subspace if it is an interpolant for $A\,|_x$ *and* $B\,|_x$. □

The following theorem provides a way to compose the interpolant from the interpolants under positive and negative cofactor subspaces.

[Theorem 1] Compose interpolant from cofactor subspaces
Let I_x *and* $I_{\neg x}$ *be interpolants of A and B under the "x = 1" and "x = 0" cofactor subspaces. Then the formula "*$(x \wedge I_x) \vee (\neg x \wedge I_{\neg x})$*" will be an interpolant of A and B.*
Proof: Due to the space limit, we omit the proof here. □

For simple cases, it is doable to recursively apply cofactor operations and then compose the interpolant from ground up. However, for any reasonably larger cases, it is not feasible due to the exponential complexity. Therefore, we will need a more efficient algorithm to correlate the cofactor interpolant concept with the conflict analysis process and devise a composition rule for the sub-interpolants[3] of the learned clauses.

During the conflict analysis process, a cut on the implication graph corresponds to a learned clause that is sufficient to refute the conflict under analysis. Let the learned clause be c_{learn}. By adding c_{learn} to the SAT problem, we can prevent the SAT engine from searching the "$c_{learn} = 0$" subspace. In other words, if $c_{learn} = (l_1 \vee l_2 \vee \dots \vee l_n)$, where l_1 to l_n are literals of this clause, then the learned clause will block the SAT search in the $(\neg l_1 \wedge \neg l_2 \wedge \dots \wedge \neg l_n)$ cofactor subspace. This leads to our next definition:

[Definition 2] Interpolant under clause refuted subspace
A formula is an interpolant of A and B under the clause cls refuted subspace if it is an interpolant for $A\,|_{\neg cls}$ *and* $B\,|_{\neg cls}$. □

In general, Definition 2 can be extended to clauses that are not learned during the conflict analysis process. Given any clause *cls*, if we can derive an interpolant for the "$\neg cls$" cofactor subspace, we can say that it is an interpolant under the "*cls*" refuted subspace.

In the next theorem, we will demonstrate the terminal cases where the sub-interpolants can be obtain without further decomposing into smaller cofactor subspaces.

[Theorem 2] Terminal case for the sub-interpolant
Let A and B be inconsistent formulae in CNF and the clause "cls" be one of the clause in A. Then the constant '0' will be an interpolant of A and B under the "cls" refuted subspace.
Proof:
Since *cls* is a clause in A, we have: $A \rightarrow cls$ (1)
By contraposition, we have: $\neg cls \rightarrow \neg A$ (2)
In other words, the projection of A in the $\neg cls$ cofactor subspace will be null. Therefore, $A\,|_{\neg cls} = 0$. (3)
Since, $0 \subseteq \neg B\,|_{\neg cls}$ (4)
From (3) and (4), we know '0' is an interpolant of A and B under the $\neg cls$ cofactor subspace (i.e. *cls* refuted subspace). □

It is worthwhile to know that the interpolant '0' for the above terminal case is not unique. In fact, any function between 0 and "$\neg B\,|_{\neg cls}$" will be a valid sub-interpolant. However, since we do not compute "$\neg B\,|_{\neg cls}$" here, we can only use '0' in our interpolant generation algorithm and leave it as an open question for the future research.

Intuitively, Theorem 3 will have a counterpart for the clause in formula B.

[3] For convenience, we call the interpolant under certain cofactor subspace "sub-interpolant" without explicitly clarifying what the cofactor variables are.

[Corollary 2.1] Terminal case for the sub-interpolant

Let A and B be inconsistent formulae in CNF and the clause "cls" be one of the clause in B. Then the constant '1' will be an interpolant of A and B under the "cls" refuted subspace.

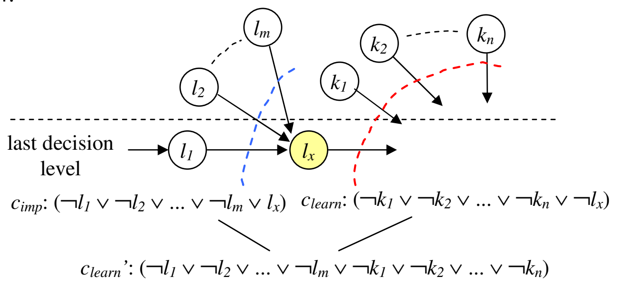
Proof:

The proof of this Corollary is quite similar to Theorem 3. Due to the space limit, we omit it here. □

In line 4 of our interpolant generation algorithm in Figure 3, the sub-interpolant `I_curr` is initialized from the clause `c_imp`, which is the clause that produces the conflict. With proper arrangement of the implication order, the decision conflict should occur only in the clause of formula A or B, and thus we can return the sub-interpolant as constant '0' or '1', respectively, according to Theorem 2. However, in case the conflict arises from a learned clause, we will return its recorded sub-interpolant which is computed in the previously corresponding conflict analysis routine. We will further illustrate the composition rules for the interpolant under the learned clause refuted subspace in the next subsection.

2.3 Sub-interpolant Composition

As we mentioned in Subsection 2.1, our interpolant generation algorithm is embedded in the conventional conflict analysis process. The repeated loop in Figure 3 goes through the implications in the last decision level, and in the middle of the iterations, we should have the implied literal l_x, implication clause c_{imp}, and current learned clause c_{learn}, as shown in Figure 4.



l_m
k_2
k_n
l_2
k_1
last decision level
l_1
l_x
c_{imp}: $(\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m \vee l_x)$
c_{learn}: $(\neg k_1 \vee \neg k_2 \vee \ldots \vee \neg k_n \vee \neg l_x)$
c_{learn}': $(\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m \vee \neg k_1 \vee \neg k_2 \vee \ldots \vee \neg k_n)$

Figure 4: Resolution of currently learned and implication clauses.

Let I_{learn} be the sub-interpolant computed up to the current learned clause c_{learn}. From Definition 2, I_{learn} can also be treated as the interpolant under the c_{learn} refuted subspace. The next step in conflict analysis will be performing the resolution on clauses c_{imp} and c_{learn}, and the resolved clause, c_{learn}', will include all the literals in c_{imp} and c_{learn} except for the pivot literal l_x.

Note that the sub-interpolant I_{imp} for the implication clause c_{imp} should be ready when the resolution takes place (i.e. line 13 of our algorithm in Figure 3).

Given the interpolants under the c_{imp} and c_{learn} refuted subspaces, the interpolant under the c_{learn}' refuted subspace can be generated by the following theorem:

[Theorem 3] Sub-interpolant composition

Let c_{imp} and c_{learn} *be the implication and learned clauses during the conflict analysis process and* l_x *be the resolved literal appearing as* l_x *and* $\neg l_x$ *in* c_{imp} and c_{learn}, *respectively. Let* I_{imp} *and* I_{learn} *be the interpolants under the* c_{imp} and c_{learn} *refuted subspace. Then "*$(\neg l_x \wedge I_{imp}) \vee (l_x \wedge I_{learn})$*" will be an interpolant of A and B under the new resolved clause* c_{learn}*' refuted subspace.*

Proof:

Let c_{imp} be $(\neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_m \vee l_x)$. I_{imp} will be an interpolant under the "$l_1 \wedge l_2 \wedge \ldots \wedge l_m \wedge \neg l_x$" cofactor subspace. (1)

Let c_{learn} be $(\neg k_1 \vee \neg k_2 \vee \ldots \vee \neg k_n \vee \neg l_x)$. I_{learn} will be an interpolant under the "$k_1 \wedge k_2 \wedge \ldots \wedge k_n \wedge l_x$" cofactor subspace. (2)

According to (1), I_{imp} will also be an interpolant under the "$l_1 \wedge l_2 \wedge \ldots \wedge l_m \wedge k_1 \wedge k_2 \wedge \ldots \wedge k_n \wedge \neg l_x$" cofactor subspace. (3)

Similarly, according to (2), I_{learn} will also be an interpolant under the "$l_1 \wedge l_2 \wedge \ldots \wedge l_m \wedge k_1 \wedge k_2 \wedge \ldots \wedge k_n \wedge l_x$" cofactor subspace. (4)

From (3), (4) and Theorem 1, "$(\neg l_x \wedge I_{imp}) \vee (l_x \wedge I_{learn})$" will be an interpolant under the "$l_1 \wedge l_2 \wedge \ldots \wedge l_m \wedge k_1 \wedge k_2 \wedge \ldots \wedge k_n$" cofactor subspace, or by Def. 2, c_{learn}' refuted subspace. □

With the above theorem, we can derive the sub-interpolant under the refuted subspace of the learned clause in the conflict analysis process. We will record the mapping from this sub-interpolant I_{learn} to the learned clause c_{learn}, and it will be used in later interpolant generation calls.

2.4 Optimization Rules

Theorem 3 constitutes the key operation of our algorithm (i.e. line 14) and enables the interpolant generation process to be conducted without constructing the resolution graph. However, it is more than necessarily complicated in most of the cases. Table 1 summarizes the potential situations when we need to apply the theorem and the corresponding simplified forms.

Table 1: Optimization rules of our algorithm

	l_x is global	l_x is local to A	l_x is local to B
$c_{imp} \in A$	② $l_x \wedge I_{learn}$	④ I_{learn}	⑦ N/A
$c_{imp} \in B$	③ $\neg l_x \vee I_{learn}$	⑥ N/A	⑤ I_{learn}
c_{imp} is learned	① $(\neg l_x \wedge I_{imp}) \vee (l_x \wedge I_{learn})$	⑧ $I_{imp} \vee I_{learn}$	⑨ $I_{imp} \wedge I_{learn}$

While case ① is the default case, cases ② and ③ can be easily obtained by setting I_{imp} to be 0 and 1, respectively (Theorem 3). Due to the space limit, we skip the explanations of other cases here.

2.5 Flexibility in the Interpolant Generation

Since the interpolant of two inconsistent formulae is not unique, there should be flexibility in the interpolant generation process that can create different interpolation circuits for different objectives and applications.

Based on our algorithm in Figure 3 and the theorems in this section, the potential flexibilities in interpolant generation include:

(1) Sub-interpolant for terminal cases: See Theorem 2.

(2) Sub-interpolant composition: Subsection 2.4 lists several optimization rules under different special situations. This implies the flexibility in choosing different sub-interpolants in the generation process.

(3) Resolution steps in the conflict analysis: Different traversals on the implication graph will result in different resolution graphs and thus different interpolants.

(4) Interpolation circuit minimization: Since we generate the sub-interpolants during the proof process, the corresponding sub-interpolation circuits can also be constructed on the fly.

(5) Extension to consistent formulae: We can extend our algorithm to generate "interpolant" for consistent formulae (i.e. $A \wedge B$ satisfiable). The interpolant will be a formula between $(A \wedge \neg B)$ and $(\neg B)$.

Other than the above, flexibility in interpolant generation can also be explored by the decision order or by computing the inverse of interpolation(B, A), etc. These are very interesting research topics and should have great impact on the applicability of the interpolation techniques.

3. EXPERIMENTAL RESULTS

In this section, we will present two sets of experimental results: (1) Quantifying the memory usage overhead in recording the resolution graph, and (2) Comparing the interpolation circuit sizes between our, Pudlák's and McMillan's algorithms.

We conducted all of our experiments on a Linux workstation with 16GB RAM and 2GHz quad-core Intel Xeon CPU. The testcases are from ISCAS'85, ISCAS'89, and ITC'99 benchmarks. In addition, we utilized the equivalent checking tool in ABC[9] framework to verify the correctness.

3.1 Memory Overhead Comparison

We first conducted experiments on "circuit resubstitution [10]" to quantify the memory usage overhead in recording the resolution graph. In this experiment, we randomly pick a gate g in a circuit and then call the interpolation engine to resynthesize the function of its fanin cone. The corresponding SAT problem is to assert a miter (i.e. adding an XOR gate) of the function and its inverse (i.e. $\neg g$). Obviously, the XOR output must be unsatisfiable and thus an interpolant can be generated as a substitution of the original function.

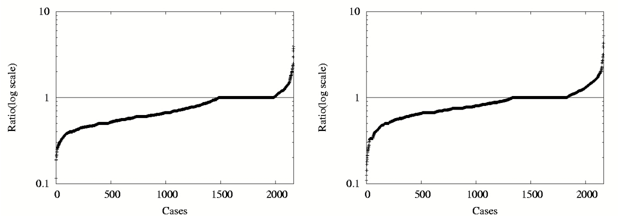
Table 2 outlines the experimental result on the memory usage comparison. The "#gates", "#confs(K)" and "avg lits" columns are the number of gates in the circuit, the total number of decision conflicts (in thousands), and the average literal counts of the conflicts. For each circuit, we randomly pick 1000 gates for circuit resubsitutions. The fanin cones are limited to 15 levels of gates. The memory overhead of our and Pudlák's algorithms are listed in 5th and 6th columns, respectively. It can be shown that our method consumes only about 24% of memory in interpolant generation when compared to Pudlák's algorithm. In other words, the memory overhead in recording resolution graphs can be as high as 76/24 = 3.17 times.]

Table 2. Memory overhead comparison for resubstitution problem

Circuit	#gates	#confs(K)	avg lits	ours(MB)	[2] (MB)	Ratio
c3540	1038	372.1	39.62	27.94	127.3	0.22
c5315	1733	63.76	14.13	3.545	13.00	0.27
c6288	2337	1125	30.76	115.0	464.0	0.25
b14	6070	499.8	34.00	43.32	174.2	0.25
b15	8993	157.1	40.09	14.00	75.44	0.19
b17	27647	139.0	34.31	12.21	63.96	0.19
b18	81710	1073	44.18	108.5	425.27	0.26
b19	163520	1132	46.03	96.20	421.6	0.23
s13207	2719	13.69	27.41	1.086	5.386	0.20
s38417	9219	86.80	14.16	6.284	21.43	0.29
average						0.24

3.2 Interpolation Circuit Size Comparison

In this section, we compare the circuit sizes of interpolants generated by our, Pudlák's and McMillan's algorithms. The Pudlák's and McMillan's algorithms are on top of the proof-logging version of MiniSat[11] solver. We conducted the same experiments for the different interpolant generation engines. Figure 5 demonstrates the results. Figures 5(a) and (b) are the circuit size ratios of ours compared to the Pudlák's and McMillan's algorithms, respectively. For most of the cases, our algorithm can generate interpolant circuit with smaller sizes.



(a) Ours / Pudlák's Algorithm **(b) Ours / McMillan's Algorithm**

Figure 5: Interpolation circuit size comparison of our, Pudlák's and McMillan's algorithms

4. CONCLUSION

In this paper, we proposed a novel interpolant generation algorithm without constructing the resolution graph of the unsatisfiability proof. The experimental results show that our algorithm can generate interpolants with smaller memory overhead and circuit sizes. In addition, we formally define the meaning of the sub-interpolants created during the conflict analysis process. We believe that our novel interpolant generation algorithm will significantly enhance the performance of the SAT-based logic synthesis and verification techniques.

5. REFERENCES

[1] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," *J. Symbolic Logic*, 22(3):250-268, 1957.

[2] P. Pudlák, "Lower bounds for resolution and cutting plane proofs and monotone computations," *J. Symbolic Logic*, 62(3):981-998, 1997.

[3] K. L. McMillan, "Interpolation and SAT-based model checking," in *Proc. CAV*, pp. 1-13, 2003.

[4] C.-C. Lee, J.-H. R. Jiang, C.-Y. R Huang, and A. Mishchenko, "Scalable Exploration of Functional Dependency by Interpolation and Incremental SAT Solving," in *Proc. ICCAD*, 2007.

[5] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-Decomposing Large Boolean Functions via Interpolation and Satisfiability Solving," in *Proc. DAC*, 2008.

[6] C. Scholl, S. Disch, F. Pigorsch, and S. Kupferschmid, "Using an SMT solver and Craig interpolation to detect and remove redundant linear constraints in representations of non-convex polyhedral," in *Proc. SMT/BPR*, 2008.

[7] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Proc. CAV*, 2006.

[8] R. Jhala, K. L. McMillan, "Interpolant-Based Transition Relation Approximation," in *Proc.CAV*, 2005.

[9] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. Release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/

[10] A. Mishchenko, R. K. Brayton, J-H. R. Jiang, and S. Jang, "Scalable Don't Care Based Logic Optimization and Resynthesis," in *Proc. FPGA'09*, pages 151-160, Feb. 2009

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