A Proof-Sensitive Approach for Small Propositional Interpolants

Leonardo Alt, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina

Formal Verification Lab, USI, Switzerland

Abstract. The labeled interpolation system (LIS) is a framework for Craig interpolation widely used in propositional-satisfiability-based model checking. Most LIS-based algorithms construct the interpolant from a proof of unsatisfiability and a fixed labeling determined by which part of the propositional formula is being over-approximated. While this results in interpolants with fixed strength, it limits the possibility of generating interpolants of small size. This is problematic since the interpolant size is a determining factor in achieving good overall performance in model checking. This paper analyses theoretically how labeling functions can be used to construct small interpolants. In addition to developing the new labeling mechanism guaranteeing small interpolants, we also present its versions managing the strength of the interpolants. We implement the labeling functions in our tool PeRIPLO and study the behavior of the resulting algorithms experimentally by integrating the tool to a variety of model checking applications. Our results suggest that the new proof-sensitive interpolation algorithm performs consistently better than any of the standard interpolation algorithms based on LIS.

1 Introduction

In SAT-based model checking, a widely used workflow for obtaining an interpolant for a propositional formula A is to compute a proof of unsatisfiability for the formula $\phi = A \wedge B$, use a variety of standard techniques for compressing the proof (see, e.g., [17]), construct the interpolant from the compressed proof, and finally simplify the interpolant [4]. The *labeled interpolation system* (LIS) [9] is a commonly used, flexible framework for computing the interpolant from a given proof that generalizes several interpolation algorithms parameterized by a *labeling function*. Given a labeling function and a proof, LIS uniquely determines the interpolant. However, the LIS framework allows significant flexibility in constructing interpolants from a proof through the choice of the labeling function.

Arguably, the suitability of an interpolant depends ultimately on the application [17], but there is a wide consensus that small interpolants lead to better overall performance in model checking [3,21,17]. However, generating small interpolants for a given partitioning is a non-trivial task. This paper presents, to the best of our knowledge, the first thorough, rigorous analysis on how labeling in the LIS framework affects the size of the interpolant. The analysis is backed

up by experimentation showing also the practical significance of the result. We believe that the results reported here will help the community working on interpolation in designing interpolation algorithms that work well independent of the application. Based on the analysis we present the *proof-sensitive interpolation algorithm* PS that produces small interpolants by adapting itself to the proof of unsatisfiability. We prove under reasonable assumptions that the resulting interpolant is always smaller than those generated by any other LIS-based algorithms, including the widely used algorithms M_s (McMillan [13]), P (Pudlák [16]), and M_w (dual to M_s [9]).

In some applications it is important to give guarantees on the logical strength of the interpolants. Since the LIS framework allows us to argue about the resulting interpolants by their logical strength [9], we know that for a fixed problem $A \wedge B$ and a fixed proof of unsatisfiability, an interpolant constructed with M_s implies one constructed with P which in turn implies one constructed with M_w. While PS is designed to control the interpolant size, we additionally define two variants controlling the interpolant strength: the strong and the weak proof-sensitive algorithms computing, respectively, interpolants that imply the ones constructed by P and that are implied by the ones constructed by P.

We implemented the new algorithms in the PeRIPLO interpolation framework [17] and confirm the practical significance of the algorithms with an experimentation. The results show that when using PS, both the sizes of the interpolants and the run times when used in a model-checking framework compare favorably to those obtained with M_s, P, and M_w, resulting occasionally in significant reductions.

1.1 Related Work

Interpolants can be compacted through applying transformations to the resolution refutation. For example, [17,18] compare the effect of such compaction on the interpolation algorithms M_s, P, and M_w in connection with function-summarization-based model checking [10,20]. A similar approach is studied in [9] combined with an analysis on the strength of the resulting interpolant. Different size-based reductions are further discussed in [4,11]. While often successful, these approaches might produce a considerable overhead in large problems. Our approach is more light-weight and uses directly the flexibility of LIS to perform the compression. An interesting analysis similar to ours, presented in [3], concentrates on the effect of identifying subsumptions in the resolution proofs. A significant reduction in the size of the interpolant can be obtained by considering only CNF-shaped interpolants [21]. However, the strength of these interpolants is not as easily controllable as in the LIS interpolants, making the technique harder to apply in certain model checking approaches. A light-weight interpolant compaction can be performed by specializing through simplifying the interpolant with a truth assignment [12].

In many verification approaches using counter-examples for refinement it is possible to abstract an interpolant obtained from a refuted counter-example. For instance, [19,2] present a framework for generalizing interpolants based on

templates. A related approach for generalizing interpolants in unbounded model-checking through abstraction is presented in [5] using incremental SAT solving. While this direction is orthogonal to ours, we believe that the ideas presented here and addressing the interpolation back-end would be useful in connection with the generalization phase.

Linear-sized interpolants can be derived also from resolution refutations computed by SMT solvers, for instance in the combined theory of linear inequalities and equalities with uninterpreted functions [14] and linear rational arithmetic [1]. These approaches have an interesting connection to ours since they also contain a propositional part. It is also possible to produce interpolants without a proof [7]. However, this method gives no control over the relative interpolant strength and reduces in the worst case to enumerating all models of a SAT instance. Finally, conceptually similar to our work, there is a renewed interest in interpolation techniques used in connection with modern ways of organizing the high-level model-checking algorithm [15,6].

2 Preliminaries

Given a finite set of propositional variables, a *literal* is a variable p or its negation $\neg p$. A *clause* is a finite set of literals and a formula ϕ in *conjunctive normal form* (CNF) is a set of clauses. We also refer to a clause as the disjunction of its literals and a CNF formula as the conjunction of its clauses. A variable p occurs in the clause C, denoted by the pair (p, C), if either $p \in C$ or $\neg p \in C$. The set $var(\phi)$ consists of the variables that occur in the clauses of ϕ. We assume that double negations are removed, i.e., $\neg\neg p$ is rewritten as p. A *truth assignment* σ assigns a Boolean value to each variable p. A clause C is satisfied if $p \in C$ and $\sigma(p)$ is true, or $\neg p \in C$ and $\sigma(p)$ is false. The propositional satisfiability problem (SAT) is the problem of determining whether there is a truth assignment satisfying each clause of a CNF formula ϕ. The special constants $\top$ and $\bot$ denote the empty conjunction and the empty disjunction. The former is satisfied by all truth assignments and the latter is satisfied by none. A formula ϕ *implies* a formula ϕ', denoted $\phi \rightarrow \phi'$, if every truth assignment satisfying ϕ satisfies ϕ'. The *size* of a propositional formula is the number of logical connectives it contains. For instance the unsatisfiable CNF formula

$$\phi = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1) \qquad (1)$$

of size 14 consists of 4 variables and 5 clauses. The occurrences of the variable x_4 are $(x_4, \neg x_2 \vee x_4)$ and $(x_4, \neg x_2 \vee \neg x_3 \vee \neg x_4)$.

For two clauses C^+, C^- such that $p \in C^+$, $\neg p \in C^-$, and for no other variable q both $q \in C^- \cup C^+$ and $\neg q \in C^- \cup C^+$, a *resolution step* is a triple C^+, C^-, $(C^+ \cup C^-) \setminus \{p, \neg p\}$. The first two clauses are called the *antecedents*, the latter is the *resolvent* and p is the *pivot* of the resolution step. A *resolution refutation* R of an unsatisfiable formula ϕ is a directed acyclic graph where the nodes are clauses and the edges are directed from the antecedents to the resolvent. The nodes of a refutation R with no incoming edge are the clauses of ϕ, and the rest

of the clauses are resolvents derived with a resolution step. The unique node with no outgoing edges is the empty clause. The *source clauses* of a refutation R are the clauses of ϕ from which there is a path to the empty clause.

Given an unsatisfiable formula $A \wedge B$, a *Craig interpolant* I for A is a formula such that $A \rightarrow I$, $I \wedge B$ is unsatisfiable and $var(I) \subseteq var(A) \cap var(B)$. An interpolant can be seen as an over-approximation of A that is still unsatisfiable when conjoined with B. In the rest of the paper we assume that A and B only consist of the source clauses of R.

The *labeled interpolation system* [9] (LIS) is a framework that, given propositional formulas A, B, a refutation R of $A \wedge B$ and a *labeling function* L, computes an interpolant I for A based on R. The refutation together with the partitioning A, B is called an *interpolation instance* (R, A, B). The labeling function L assigns a label from the set $\{a, b, ab\}$ to every variable occurrence (p, C) in the clauses of the refutation R. A variable is *shared* if it occurs both in A and B; otherwise it is *local*. For all variable occurrences (p, C) in R, $L(p, C) = a$ if p is local to A and $L(p, C) = b$ if p is local to B. For occurrences of shared variables in the source clauses the label may be chosen freely. The label of a variable occurrence in a resolvent C is determined by the label of the variable in its antecedents. For a variable occurring in both its antecedents with different labels, the label of the new occurrence is ab, and in all other cases the label is equivalent to the label in its antecedent or both antecedents.

An interpolation algorithm based on LIS computes an interpolant with a dynamic algorithm by annotating each clause of R with a *partial interpolant* starting from the source clauses. The partial interpolant of a source clause C is

$$I(C) = \begin{cases} \bigvee\{l \mid l \in C \text{ and } L(var(l), C) = b\} & \text{if } C \in A, \text{ and} \\ \bigwedge\{\neg l \mid l \in C \text{ and } L(var(l), C) = a\} & \text{if } C \in B, \end{cases} \tag{2}$$

The partial interpolant of a resolvent clause C with pivot p and antecedents C^+ and C^-, where $p \in C^+$ and $\neg p \in C^-$, is

$$I(C) = \begin{cases} I(C^+) \vee I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = a, \\ I(C^+) \wedge I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = b, \text{ and} \\ (I(C^+) \vee p) \wedge (I(C^-) \vee \neg p) & \text{otherwise.} \end{cases} \tag{3}$$

The interpolation algorithms M_s, P, and M_w mentioned in the introduction can be obtained as special cases of LIS by providing a labeling function returning b, ab, and a for the shared variables, respectively.

In some applications it is useful to consider different interpolants constructed from a fixed interpolation instance, but using different interpolation algorithms [10]. For such cases the LIS framework provides a convenient tool for analyzing whether the interpolants generated by one interpolation algorithm always imply the interpolants generated by another algorithm. If we order the three labels so that $b \leq ab \leq a$, it can be shown that given two labeling functions L and L' resulting in the interpolants I_L and $I_{L'}$ in LIS and having the property that $L(p, C) \leq L'(p, C)$ for all occurrences (p, C), it is true that $I_L \rightarrow I_{L'}$. In this case

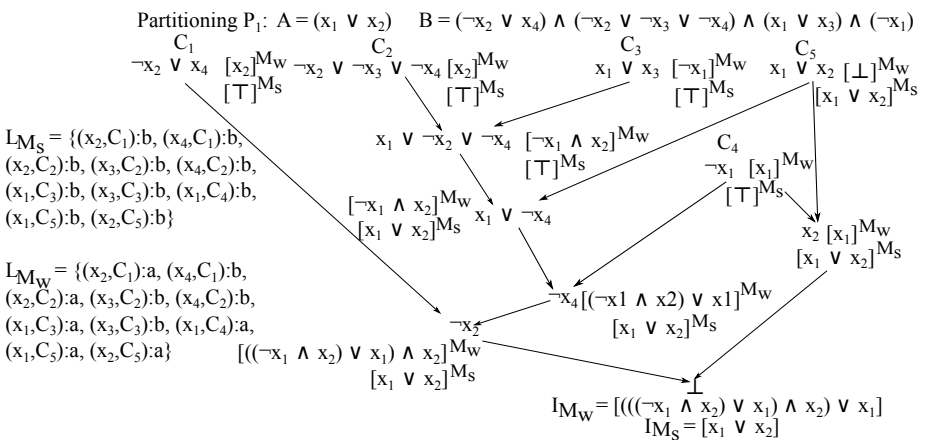


Fig. 1. Different interpolants obtained from the refutation using the partitioning P_1.

we say that the interpolation algorithm obtained from LIS using the labeling L' is *weaker* than the interpolation algorithm that uses the labeling L.

We define here two concepts that will be useful in the next section: the class of *uniform* labeling functions, and the *internal size* of an interpolant.

Definition 1. *A labeling function is* uniform *if for all pairs of clauses $C, D \in R$ containing the variable p, $L(p, C) = L(p, D)$, and no occurrence is labeled ab. Any interpolation algorithm with uniform labeling function is also called uniform.*

An example of non-uniform labeling function is D_{min}, presented in [8]. D_{min} is proven to produce interpolants with the least number of distinct variables.

Definition 2. *The* internal size *$IntSize(I)$ of an interpolant I is the number of connectives in I excluding the connectives contributed by the partial interpolants associated with the source clauses.*

Typically, an interpolant constructed by a LIS-based algorithm will contain a significant amount of subformulas that are syntactically equivalent. The *structural sharing*, i.e., maintaining a unique copy of the syntactically equivalent subformulas, while completely transparent to the satisfiability, is of critical practical importance. Similarly important for performance is the *constant simplification*, consisting of four simple rewriting rules: $\top \wedge \phi \rightsquigarrow \phi$, $\bot \wedge \phi \rightsquigarrow \bot$, $\top \vee \phi \rightsquigarrow \top$, and $\bot \vee \phi \rightsquigarrow \phi$, where ϕ is an arbitrary Boolean formula.

The following example illustrates the concepts discussed in this section by showing how LIS can be used to compute interpolants with two different uniform algorithms M_s and M_w.

Example 1. Consider the unsatisfiable formula $\phi = A \wedge B$ where ϕ is from Eq. (1) and $A = (x_1 \vee x_2)$ and $B = (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_3) \wedge (\neg x_1)$. Figure 1 shows a resolution refutation for ϕ and the partial interpolants computed by the interpolation algorithms M_s and M_w. Each clause in the refutation is associated with a partial interpolant ψ generated by labeling L_{M_s} (denoted by $[\psi]^{\mathrm{M}_s}$)

and a partial interpolant ψ' generated by labeling L_{M_w} (denoted by $[\psi']^{\mathrm{M}_w}$). The generated interpolants are $I_{\mathrm{M}_s} = x_1 \vee x_2$ and $I_{\mathrm{M}_w} = (((\neg x_1 \wedge x_2) \vee x_1) \wedge x_2) \vee x_1$. Now consider a different partitioning $\phi' = A' \wedge B'$ for the same formula where the partitions have been swapped, that is, $A' = B$ and $B' = A$. Using the same refutation (figure omitted for lack of space), we get the interpolants $I'_{\mathrm{M}_s} = (((x_1 \vee \neg x_2) \wedge \neg x_1) \vee \neg x_2) \wedge \neg x_1 = \neg I_{\mathrm{M}_w}$ and $I'_{\mathrm{M}_w} = \neg(x_1 \vee x_2) = \neg I_{\mathrm{M}_s}$ We use both structural sharing and constant simplification in the example. The internal size of I_{M_s} is 0, whereas the internal size of I_{M_w} is 4.

The two partitionings illustrate a case where the interpolation algorithm M_s, in comparison to M_w, produces a small interpolant for one and a large interpolant for another interpolation instance. Since the goal in this work is to develop LIS-based interpolation algorithms that consistently produce small interpolants, the labeling function of choice cannot be L_{M_s} or L_{M_w}. Note that while in this case the interpolants I_{M_s} and I_{M_w} are equivalent, the representation of I_{M_w} is considerably smaller than the representation of I_{M_s}. Since minimizing a propositional formula is an NP-complete problem, producing interpolants that are small in the first place is a very important goal.

3 Labeling Functions for LIS

This section studies the algorithms based on the labeled interpolation system in an analytic setting. Our main objective is to provide a basis for developing and understanding labeling functions that construct interpolants having desirable properties. In particular, we will concentrate on three syntactic properties of the interpolants: the number of distinct variables; the number of literal occurrences; and the internal size of the interpolant. In most of the discussion in this section we will ignore the two optimizations on structural sharing and constraint simplification. While both are critically important for practicality of interpolation, our experimentation shows that they mostly have similar effect on all the interpolation algorithms we studied, and therefore they can be considered orthogonally (see Sec. 4.4). The exception is that *the non-uniform labeling functions allow a more efficient optimization compared to the uniform labeling functions* through constraint simplification. More specifically, the main results of the section are the following theorems.

(i) If an interpolation instance is not p-annihilable (see Def. 3), which in our experimentation turns out almost always to be the case, then all LIS interpolants constructed from the refutation have the same number of distinct variables (Theorem 1);
(ii) For a given interpolation instance, the interpolants I_n obtained with any non-uniform labeling function and I_u obtained with any uniform labeling function satisfy $\mathit{IntSize}(I_u) \leq \mathit{IntSize}(I_n)$. (Theorem 2); and
(iii) Among uniform labeling functions, the *proof-sensitive* labeling function (see Def. 4) results in the least number of variable occurrences in the partial interpolants associated with the source clauses (Theorem 3).

From the three theorems we immediately have the following:

Corollary 1. *For not p-annihilable interpolation instances, the proof-sensitive labeling function will result in interpolants that have the smallest internal size, the least number of distinct variables, and least variable occurrences in the source partial interpolants.*

The proof-sensitive interpolant strength can only be given the trivial guarantees: it is stronger than I_{M_w} and weaker than I_{M_s}. At the expense of the minimality in the sense of the above corollary, we introduce in Equations (6) and (7) the weak and strong versions of the proof-sensitive labeling functions.

3.1 Analysing Labeling Functions

An interesting special case in LIS-based interpolation algorithms is when the labeling can be used to reduce the number of distinct variables in the final interpolant. To make this explicit we define the concepts of a p-pure resolution step and a p-annihilable interpolation instance.

Definition 3. *Given an interpolation instance (R, A, B), a variable $p \in var(A) \cup var(B)$ and a labeling function L, a resolution step in R is* p-pure *if at most one of the antecedents contain p, or both antecedents C, D contain p but $L(p, C) = L(p, D) = a$ or $L(p, C) = L(p, D) = b$. An interpolation instance (R, A, B) is* p-annihilable *if there is a non-uniform labeling function L such that $L(p, C) = a$ if $C \in A$, $L(p, C) = b$ if $C \in B$, and all the resolution steps are p-pure.*

The following theorem shows the value of p-annihilable interpolation instances in constructing small interpolants.

Theorem 1. *Let (R, A, B) be an interpolation instance, $p \in var(A) \cap var(B)$, and I an interpolant obtained from (R, A, B) by means of a LIS-based algorithm. If $p \notin var(I)$, then (R, A, B) is p-annihilable.*

Proof. Assume that (R, A, B) is not p-annihilable, $p \in var(A) \cap var(B)$, but there is a labeling L which results in a LIS-based interpolation algorithm that constructs an interpolant not containing p. The labeling function cannot have $L(p, C) = b$ if $C \in A$ or $L(p, C) = a$ if $C \in B$ because p would appear in the partial interpolants associated with the sources by Eq. (2). No clause C in R can have $L(p, C) = ab$ since all literals in the refutation need to be used as a pivot on the path to the empty clause, and having an occurrence of p labeled ab in an antecedent clause would result in introducing the literal p to the partial interpolant associated with the resolvent by Eq. (3) when used as a pivot. Every resolution step in the refutation R needs to be p-pure, since if the antecedents contain occurrences (p, C) and (p, D) such that $L(p, C) \neq L(p, D)$ either the label of the occurrence of p in the resolvent clause will be ab, violating the condition that no clause can have $L(p, C) = ab$ above, or, if p is pivot on the resolution step, the variable is immediately inserted to the partial interpolant by Eq. (3). □

While it is relatively easy to artificially construct an interpolation instance that is p-annihilable, they seem to be rare in practice (see Section 4.4). Hence, while instances that are p-annihilable would result in small interpolants, it has little practical significance at least in the benchmarks available to us. However, we have the following practically useful result which shows the benefits of labeling functions producing p-pure resolution steps in computing interpolants with low number of connectives.

Theorem 2. *Let (R, A, B) be an interpolation instance. Given a labeling function L such that the resolution steps in R are p-pure for all $p \in var(A \wedge B)$, and a labeling function L' such that at least one resolution step in R is not p-pure for some $p \in var(A \wedge B)$, we have $IntSize(I_L) \leq IntSize(I_{L'})$.*

Proof. For a given refutation R, the number of partial interpolants will be the same for any LIS-based interpolation algorithm. By Eq. (3) each resolution step will introduce one connective if both occurrences in the antecedents are labeled a or b and three connectives otherwise. The latter can only occur if the labeling algorithm results in a resolution step that is not p-pure for some p. □

Clearly, p-pure steps are guaranteed with uniform labeling functions. Therefore we have the following corollary:

Corollary 2. *Uniform labeling functions result in interpolants with smaller internal size compared to non-uniform labeling functions.*

The main result of this work is the development of a labeling function that is uniform, therefore producing small interpolants by Corollary 2, and results in the smallest number of variable occurrences among all uniform labeling functions. This *proof-sensitive labeling function* works by considering the refutation R when assigning labels to the occurrences of the shared variables.

Definition 4. *Let R be a resolution refutation for $A \wedge B$ where A and B consist of the source clauses, $f_A(p) = |\{(p, C) \mid C \in A\}|$ be the number of times the variable p occurs in A, and $f_B(p) = |\{(p, C) \mid C \in B\}|$ the number the variable p occurs in B. The proof-sensitive labeling function L_{PS} is defined as*

$$L_{\mathrm{PS}}(p, C) = \begin{cases} a & \text{if } f_A(p) \geq f_B(p) \\ b & \text{if } f_A(p) < f_B(p). \end{cases} \tag{4}$$

Note that since L_{PS} is uniform, it is independent of the clause C. Let Sh_A be the set of the shared variables occurring at least as often in clauses of A as in B and Sh_B the set of shared variables occurring more often in B than in A:

$$\begin{aligned} Sh_A &= \{p \in var(A) \cap var(B) \mid f_A(p) \geq f_B(p)\} \text{ and} \\ Sh_B &= \{p \in var(A) \cap var(B) \mid f_A(p) < f_B(p)\} \end{aligned} \tag{5}$$

Theorem 3 states the optimality with respect to variable occurrences of the algorithm PS among uniform labeling functions.

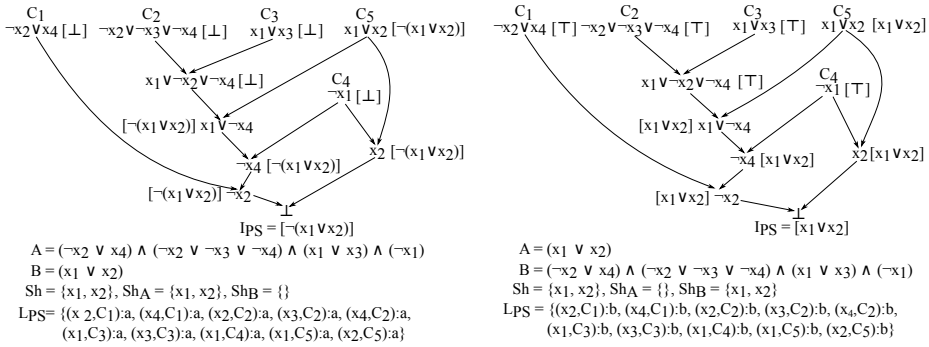


Fig. 2. Interpolants obtained by PS.

Theorem 3. *For a fixed interpolation instance (R, A, B), the interpolation algorithm* PS *will introduce the smallest number of variable occurrences in the partial interpolants associated with the source clauses of R among all uniform interpolation algorithms.*

Proof. The interpolation algorithm PS is a uniform algorithm labeling shared variables either as a or b. Hence, the shared variables labeled a will appear in the partial interpolants of the source clauses from B of R, and the shared variables labeled b will appear in the partial interpolants of the source clauses from A of R. The sum of the number of variable occurrences in the partial interpolants associated with the source clauses by PS is

$$n_{\mathrm{PS}} = \sum_{v \in Sh_B} f_A(v) + \sum_{v \in Sh_A} f_B(v).$$

We will show that swapping uniformly the label of any of the shared variables will result in an increase in the number of variable occurrences in the partial interpolants associated with the source clauses of R compared to n_{PS}. Let v be a variable in Sh_A. By (4) and (5), the label of v in PS will be a. Switching the label to b results in the size $n' = n_{\mathrm{PS}} - f_B(v) + f_A(v)$. Since v was in Sh_A we know that $f_A(v) \geq f_B(v)$ by (5), and therefore $-f_B(v) + f_A(v) \geq 0$ and $n' \geq n_{\mathrm{PS}}$. An (almost) symmetrical argument shows that swapping the label for a variable $v \in Sh_B$ to a results in $n' > n_{\mathrm{PS}}$. Hence, swapping uniformly the labeling of PS for any shared variable will result in an interpolant having at least as many variable occurrences in the leaves. Assuming no simplifications, the result holds for the final interpolant. □

Example 2. Fig. 2 shows the interpolants that PS would deliver if applied to the same refutation R of ϕ and partitionings $A \wedge B$ and $A' \wedge B'$ given in Example 1. Notice that PS adapts the labeling to the best one depending on the refutation and partitions, and gives small interpolants for both cases.

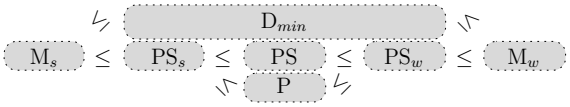
Because of the way L_{PS} labels the variable occurrences, we cannot beforehand determine the strength of PS relative to, e.g., the algorithms M_s, P, and M_w.

Although it is often not necessary that interpolants have a particular strength, in some applications this has an impact on performance or even soundness [17]. To be able to apply the idea in applications requiring specific interpolant strength, for example tree interpolation, we propose a weak and a strong version of the proof-sensitive interpolation algorithm, PS_w and PS_s. The corresponding labeling functions L_{PS_w} and L_{PS_s} are defined as

$$L_{\mathrm{PS}_w}(p, C) = \begin{cases} a & \text{if } p \text{ is not shared and } C \in A \text{ or } p \in Sh_A \\ b & \text{if } p \text{ is not shared and } C \in B \\ ab & \text{if } p \in Sh_B \end{cases} \tag{6}$$

$$L_{\mathrm{PS}_s}(p, C) = \begin{cases} a & \text{if } p \text{ is not shared and } C \in A \\ b & \text{if } p \text{ is not shared and } C \in B, \text{ or } p \in Sh_B \\ ab & \text{if } p \in Sh_A \end{cases} \tag{7}$$

Finally, it is fairly straightforward to see based on the definition of the labeling functions that the strength of the interpolants is partially ordered as shown in the diagram below.



4 Experimental Results

We implemented the three interpolation algorithms within the PeRIPLO [17] toolset and compare them with the D_{min} algorithm, as well as with the popular algorithms M_s, P and M_w in the context of three different model-checking tasks: (*i*) incremental software model checking with function summarization using FunFrog [20]; (*ii*) checking software upgrades with function summarization using eVolCheck [10]; and (*iii*) pre-image overapproximation for hardware model checking with PdTRAV [5]. The wide range of experiments permits the study of the general applicability of the new techniques. In experiments (*i*) and (*ii*) the new algorithms are implemented within the verification process allowing us to evaluate their effect on the full verification run. Experiment (*iii*) focuses on the size of the interpolant, treating the application as a black box. Unlike in the theory presented in Section 3, all experiments use both structural sharing and constraint simplification, since the improvements given by these practical techniques are important. Experiments (*i*) and (*ii*) use a large set of benchmarks each containing a different call-tree structure and assertions distributed on different levels of the tree. For (*iii*), the benchmarks consisted of a set of 100 interpolation problems constructed by PdTRAV. All experiments use PeRIPLO both as the interpolation engine and as the SAT solver.

Fig. 3 shows a generic verification framework employing the new labeling mechanism for interpolation. Whenever the application needs an interpolant for the problem $A \wedge B$, it first requests the refutation from the SAT solver. After

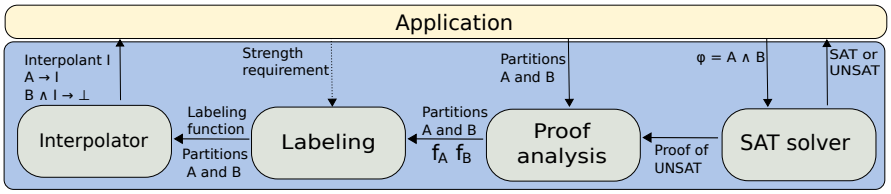


Fig. 3. Overall verification/interpolation framework.

the refutation is generated, the application provides the partitioning to the proof analyser, which will generate functions f_A and f_B (Def. 4). The labeling engine then creates a labeling function based on the partitions A and B, the functions f_A and f_B, and a possible strength requirement from the application, and then passes it to the interpolator. The latter will finally construct an interpolant and return it to the application.

As mentioned in Section 1, different verification tasks may require different kinds of interpolants. For example, [17] reports that the FunFrog approach works best with strong interpolants, whereas the eVolCheck techniques rely on weaker interpolants that have the tree-interpolation property. As shown in [18], only interpolation algorithms stronger than or equal to P are guaranteed to have this property. Therefore, we evaluated only M_s, P and PS_s for (ii), and M_s, P, M_w, PS, PS_w and PS_s for (i) and (iii). D_{min} was evaluated against the other algorithms for (i), but couldn't be evaluated for (ii) because it does not preserve the tree interpolation property. For (iii), D_{min} was not evaluated due to its poor performance in (i).

In the experiments (i) and (ii), the overall verification time of the tools and average size of interpolants were analysed. For (iii) only the size was analysed. In all the experiments the size of an interpolant is the number of connectives in its DAG representation.

The tool and experimental data are available at `http://verify.inf.usi.ch/periplo`.

4.1 Incremental Verification with Function Summarization

FunFrog is a SAT-based bounded-model-checker for C designed to incrementally check different assertions. The checker works by unwinding a program up to some predefined bound and encoding the unwound program together with the negation of each assertion to a *BMC* formula which is then passed to a SAT solver. If the result is unsatisfiable, FunFrog reports that the program is safe with respect to the provided assertion. Otherwise, it returns a counter-example produced from the model of the BMC formula.

Craig interpolation is applied in FunFrog to extract *function summaries* (relations over input and output parameters of a function that over-approximate its behavior) to be reused between checks of different assertions with the goal of improving overall verification efficiency. Given a program P, and an assertion π, let $\phi_{P,\pi}$ denote the corresponding BMC formula. If $\phi_{P,\pi}$ is unsatisfiable,

FunFrog uses Craig Interpolation to extract function summaries. This is an iterative procedure for each function call f in P. Given f, the formula $\phi_{P,\pi}$ is partitioned as $\phi_{P,\pi} \equiv A_f \wedge B_\pi$, where A_f encodes f and its nested calls, B_π the rest of the program and the negated assertion π. FunFrog then calls PeRIPLO to compute an interpolant $I^{f,\pi}$ for the function f and assertion π.

While checking the program with respect to another assertion π', FunFrog constructs the new BMC formula $\phi_{P,\pi'}$, $\equiv I^{f,\pi} \wedge B_{\pi'}$; where $I^{f,\pi}$ is used to over-approximate f. If $\phi_{P,\pi'}$ is unsatisfiable then the over-approximation was accurate enough to prove that π' holds in P. On the other hand, satisfiability of $\phi_{P,\pi'}$ could be caused by an overly weak over-approximation of $I^{f,\pi}$. To check this hypothesis, $\phi_{P,\pi'}$ is refined to $\phi^{ref}_{P,\pi'}$, in which $I^{f,\pi}$ is replaced by the precise encoding of f and the updated formula is solved again. If $\phi^{ref}_{P,\pi'}$ is satisfiable, the error is real. Otherwise, the unsatisfiable formula $\phi^{ref}_{P,\pi'}$ is used to create new function summaries in a similar manner as described above.

In our previous work [20,17] FunFrog chooses the interpolation algorithm from the set $\{\mathrm{M}_s, \mathrm{P}, \mathrm{M}_w\}$ and uses it to create summaries for all function calls in the program. In this paper, we add the algorithms PS, PS_w and PS_s to the portfolio of the interpolation algorithms and show that in particular the use of PS and PS_s improves quality of function summaries in FunFrog and therefore makes overall model checking procedure more efficient.

Experiments. The set of benchmarks consists of 23 C programs with different number of assertions. FunFrog verified the assertions one-by-one incrementally traversing the program call tree. The main goal of ordering the checks this way is to maximize the reuse of function summaries and thus to test how the labeling functions affect the overall verification performance. To illustrate our setting, consider a program with the chain of nested function calls

$$main()\{f()\{g()\{h()\{\}assert_g\}assert_f\}assert_{main}\},$$

where $assert_F$ represents an assertion in the body of function F. In a successful scenario, (a) $assert_g$ is detected to hold and a summary I^h for function h is created; (b) $assert_f$ is efficiently verified by exploiting I^h, and I^g is then built over I^h; and (c) finally $assert_{main}$ is checked against I^g.

Figure 4 (left) shows FunFrog's performance with each interpolation algorithm. Each curve represents an interpolation algorithm, and each point on the curve represents one benchmark run using the corresponding interpolation algorithm, with its verification time on the vertical axis. The benchmarks are sorted by their run time. The PS and PS_s curves are mostly lower than those of the other interpolation algorithms, suggesting they perform better. Table 1 (left) shows the sum of FunFrog verification time for all benchmarks and the average size of all interpolants generated for all benchmarks for each interpolation algorithm. We also report the relative time and size increase in percents. Both PS and PS_s are indeed competitive for FunFrog, delivering interpolants smaller than the other interpolation algorithms.

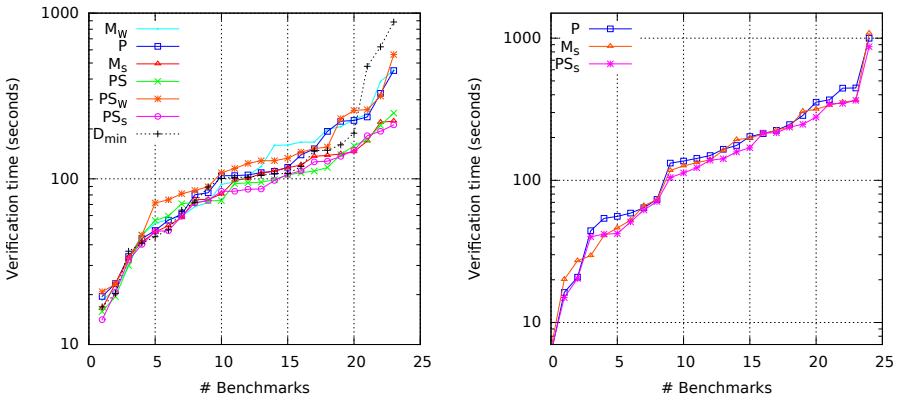


Fig. 4. Overall verification time of FunFrog (*left*) and eVolCheck (*right*) using different interpolation algorithms.

4.2 Upgrade Checking using Function Summarization

eVolCheck is an Upgrade Checker for C, built on top of FunFrog. It takes as an input an original program S and its upgrade T sharing the set of functions calls $\{f\}$. eVolCheck uses the interpolation-based function summaries $\{I^{S,f}\}$, constructed for S as shown in Sect. 4.1 to perform upgrade checking. In particular, it verifies whether for each function call f the summary $I^{S,f}$ over-approximates the precise behavior of T. This local check is turned into showing unsatisfiability of $\neg I^{S,f} \wedge A_{T,f}$, where $A_{T,f}$ encodes f and its nested calls in T. If proven unsatisfiable, eVolCheck applies Craig Interpolation to refine the function summary with respect to T.

Experiments. The benchmarks consist of the ones used in the FunFrog experiments and their upgrades. We only experiment with M_s, P and PS_s since eVolCheck requires algorithms at least as strong as P. Figure 4 (right) demonstrates that PS_s, represented by the lower curve, outperforms the other algorithms also for this task. Table 1 (right) shows the total time eVolCheck requires to check the upgraded versions of all benchmarks and average interpolant size for each of the three interpolation algorithms. Also for upgrade checking, the

Table 1. Sum of overall verification time and average interpolants size for the FunFrog (left) and eVolCheck (right) using the applicable labeling functions.

	FunFrog							eVolCheck		
	M_s	P	M_w	PS	PS_w	PS_s	D_{min}	M_s	PS_s	P
Time (s)	2333	3047	3207	2272	3345	**2193**	3811	4867	**4422**	5081
increase %	6	39	46	3	52	0	74	10	0	16
Avg size	48101	79089	86831	43781	95423	**40172**	119306	246883	**196716**	259078
increase %	20	97	116	9	137	0	197	26	0	32

Table 2. Average size and increase relative to the winner for interpolants generated when interpolating over A (top) and B (bottom) in $A \wedge B$ with PDTRAV.

	M_s	P	M_w	PS	PS_w	PS_s
Avg size	683233	724844	753633	**683215**	722605	685455
increase %	0.003	6	10	0	6	0.3
Avg size	699880	694372	649149	**649013**	650973	692434
increase %	8	7	0.02	0	0.3	7

interpolation algorithm PS_s results in smaller interpolants and lower run times compared to the other studied interpolation algorithms.

4.3 Overapproximating pre-image for Hardware Model Checking

PDTRAV [5] implements several verification techniques including a classical approach of unbounded model checking for hardware designs [13]. Given a design and a property, the approach encodes the existence of a counterexample of a fixed length k into a SAT formula and checks its satisfiability. If the formula is unsatisfiable, proving that no counterexample of length k exists, Craig interpolation is used to over-approximate the set of reachable states. If the interpolation finds a fixpoint, the method terminates reporting safety. Otherwise, k is incremented and the process is restarted.

Experiments. For this experiment, the benchmarks consist of interpolation instances generated by PDTRAV. We compare the effect of applying different interpolation algorithms on the individual steps of the verification procedure.[1]

Table 2 (top) shows the average size of the interpolants generated for all the benchmarks using each interpolation algorithm, and the relative size compared to the smallest interpolant. Also for these approaches the best results are obtained from M_s, PS and PS_s, with PS being the overall winner. We note that M_s performs better than M_w likely due to the structure of the interpolation instances in these benchmarks: the partition B in $A \wedge B$ is substantially larger than the partition A. This structure favors algorithms that label many literals as b, since the partial interpolants associated with the clauses in B will be empty while the number of partial interpolants associated with the partition A will be small. To further study this phenomenon we interchanged the partitions, interpolating this time over B in $A \wedge B$ for the same benchmarks resulting in problems where the A part is large. Table 2 (bottom) shows the average size of the interpolants generated for these benchmarks and the relative size difference compared to the winner. Here M_w and PS_w perform well, while PS remains the overall winner.

We conclude that the experimental results are compatible with the analysis in Sec. 3. In the FUNFROG and EVOLCHECK experiments, PS_s outperformed the

[1] The forthcoming research question is how interpolants generated using PS affect the convergence. This study is however orthogonal to ours and left for future work.

other interpolation systems with respect to verification time and interpolant size. PDTRAV experiments confirm in addition that PS is very capable in adapting to the problem, giving best results in both cases while the others work well in only one or the other.

4.4 Effects of Simplification

It is interesting to note that in our experiments the algorithm PS was not always the best, and the non-uniform interpolation algorithm PS_s sometimes produced the smallest interpolant, seemingly contradicting Corollary 1. A possible reason for this anomaly could be in the small difference in how constraint simplification interacts with the interpolant structure. Assume, in Eq. (3), that $I(C^+)$ or $I(C^-)$ is either constant true or false. As a result in the first and the second case respectively, the resolvent interpolant size decreases by one in Eq. (3). However in the third case, potentially activated only for non-uniform algorithms, the simplification if one of the antecedents' partial interpolants is false decreases the interpolant size by two, resulting in partial interpolants with smaller internal size. Therefore, in some cases, the good simplification behavior of non-uniform algorithms such as PS_s seems to result in slightly smaller interpolants compared to PS. We believe that this is also the reason why P behaves better than M_s and M_w in some cases.

We also observed (detailed data not shown) that in only five of the benchmarks a labeling function led to interpolants with less distinct variables, the difference between the largest and the smallest number of distinct variables being never over 3%, suggesting that p-annihilable interpolation instances are rare. Finally, we measured the effect of structural sharing. The results (see Appendix A) show that there is no noticeable, consistent difference between any of the algorithms, suggesting that the theory developed in Sec. 3 suffices to explain the experimental observations.

5 Conclusion and Future Work

This paper studies the *labeled interpolation system* (LIS), a framework for constructing interpolation algorithms for propositional proofs. In particular, we study how different labeling functions influence the resulting interpolants by analyzing how the choice of labeling affects several size metrics. Based on the results we construct three new interpolation algorithms: the algorithm PS that decides the labeling based on the resolution refutation, and its strong and weak variants. We show that under certain practical assumptions PS results in the smallest interpolants among the framework. Experimentally, when fully integrated with two software model checkers, PS or its stronger variant outperforms widely used algorithms. The results are similarly encouraging when we overapproximate pre-image in unbounded model checking with PS. We believe that this result is due to the size reduction obtained by the new algorithms.

In the future we plan to study why p-annihilable proofs are rare and how to make them common. We also plan to integrate our framework more tightly with other model checkers through efficiently exchanging proofs and interpolants.

Acknowledgements. We thank our colleagues Professor Gianpiero Cabodi and Danilo Vendraminetto from the University of Turin, Italy for the benchmarks and instructions related to PdTRAV. This work was funded by the Swiss National Science Foundation (SNSF), under the project #200021_138078.

References

1. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: CAV. pp. 313–329 (2013)
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: LPAR. pp. 46–61 (2012)
3. Bloem, R., Malik, S., Schlaipfer, M., Weissenbacher, G.: Reduction of resolution refutations and interpolants via subsumption. In: HVC. pp. 188–203 (2014)
4. Cabodi, G., Lolacono, C., Vendraminetto, D.: Optimization techniques for Craig interpolant compaction in unbounded model checking. In: DATE. pp. 1417–1422 (2013)
5. Cabodi, G., Murciano, M., Nocco, S., Quer, S.: Stepping forward with interpolants in unbounded model checking. In: ICCAD. pp. 772–778 (2006)
6. Cabodi, G., Palena, M., Pasini, P.: Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking. In: FMCAD. pp. 43–50 (2014)
7. Chockler, H., Ivrii, A., Matsliah, A.: Computing interpolants without proofs. In: HVC, pp. 72–85 (2012)
8. D'Silva, V.: Propositional interpolation and abstract interpretation. In: ESOP. pp. 185–204 (2010)
9. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI. pp. 129–145 (2010)
10. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: TACAS. pp. 292–307 (2013)
11. Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: propositional resolution proofs via partial regularization. In: CADE. pp. 237–251 (2011)
12. Jancík, P., Kofron, J., Rollini, S.F., Sharygina, N.: On interpolants and variable assignments. In: FMCAD. pp. 123–130 (2014)
13. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV. pp. 1–13 (2003)
14. McMillan, K.L.: An interpolating theorem prover. In: TACAS. pp. 16–30 (2004)
15. McMillan, K.L.: Lazy annotation revisited. In: CAV. pp. 243–259 (2014)
16. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(3), 981–998 (1997)
17. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: LPAR. pp. 683 – 693 (2013)
18. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: CAV. pp. 193–209 (2012)
19. Rümmer, P., Subotic, P.: Exploring interpolants. In: FMCAD. pp. 69–76 (2013)
20. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded model checking with interpolation-based function summarization. In: ATVA. pp. 203–207 (2012)
21. Vizel, Y., Ryvchin, V., Nadel, A.: Efficient generation of small interpolants in CNF. In: CAV. pp. 330–346 (2013)

Appendix A Experiments on simplifications by structural sharing

To investigate the effect of structural sharing on simplifications, we analysed two parameters: the number of connectives in an interpolant on its pure tree representation ($Size_{Tree}$), and the number of connectives in an interpolant on its DAG representation ($Size_{DAG}$), which is the result of the application of structural sharing. Thus, we believe that the ratio $Size_{Tree}/Size_{DAG}$ is a good way to measure the amount of simplifications due to structural sharing.

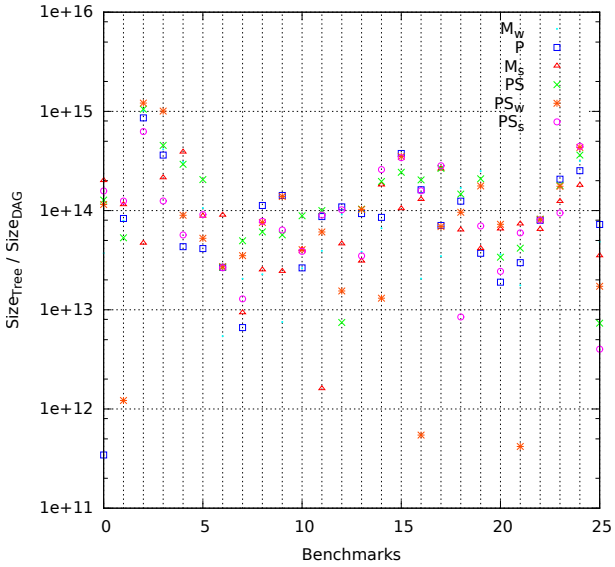


Fig. 5. Relation $Size_{Tree}/Size_{DAG}$ on FunFrog benchmarks for different interpolation algorithms

Figure 5 shows the results of this analysis on FunFrog benchmarks. Each vertical line represents a benchmark, and each point on this line represents the ratio $Size_{Tree}/Size_{DAG}$ of the interpolant generated by each of the interpolation algorithms for the first assertion of that benchmark. The reason why only the first assertion is considered is that from the second assertion on, summaries (that is, interpolants) are used instead of the original code, and therefore it is not guaranteed that the refutations will be the same when different interpolation algorithms are applied.

It is noticeable that the existence of more/less simplifications is not related to the interpolation algorithms, since all of them have cases where many/few simplifications happen. Therefore, there is no difference between any of the algorithms with respect to structural sharing.