Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Exploiting Partial Variable Assignment in Interpolation-based Model Checking

Pavel Jančík[1] · **Jan Kofroň**[1] ·
Leonardo Alt[2] · **Grigory Fedyukovich**[2] ·
Antti E. J. Hyvärinen[2] ·
Natasha Sharygina[2] ·

Received: date / Accepted: date

Abstract Craig interpolation has been successfully employed in symbolic program verification as a means of abstraction for sets of program states. In this article, we present the Partial Variable Assignment Interpolation System, an extension of the Labeled Interpolation System, enriched by partial variable assignments. It allows for both generation of smaller interpolants as well as for their faster computation. We present proofs of important properties of the interpolation system as well as a set of experiments proving its usefulness.

Keywords Craig interpolant, refutation, heuristics, reduction, variable assignment

1 Introduction

Craig interpolants are heavily applied in symbolic model checking techniques, typically used as a means of abstraction. The basic idea of using interpolants in these settings is to over-approximate the reachable set of states, while not intersecting with the undesired (faulty) states. The main benefit of not representing a set of states precisely is usually a much simpler representation of the corresponding over-approximation than of the set itself. The techniques employing interpolation differ in the way the interpolants are utilized as well as in additional properties the interpolants have to satisfy, e.g., the path-interpolation [31], the state-transition-interpolation [2], and the tree-interpolation property [22]. These properties are essential for the safety of the corresponding approaches. The size and the logical strength of interpolants are other important attributes [5, 19], which are orthogonal to the aforementioned ones. In this work, we present a technique to reduce the size of interpolants while preserving the ability to control their logical strength.

This work was partially supported by the Czech Science Foundation project number 18-17403S.

[1]Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
E-mail: jancik@d3s.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz
[2]University of Lugano, Lugano, Switzerland
E-mail: antti.hyvarinen@gmail.com, natasha.sharygina@usi.ch

When verifying properties of a program, a possible approach is to transform it to an Abstract Reachability Graph (ARG) [1] and consequently to encode this representation as a (propositional) formula. The formula expressing the desired property (e.g., the argument of an *assert* statement in a programming language) is negated and conjoined with the formula representing the program. Satisfiability of the conjunction is checked; finding a satisfying assignment of the formula then corresponds to reaching an error state (a state satisfying negation of the property), while unsatisfiability proves validity of the property. To improve efficiency of this process, over-approximation of sets of states in the form of Craig interpolants can be used. Here, each node in ARG is connected with an (over)-approximation of reachable program states at that node; the over-approximation needs to be strong enough to block all the traces via a given node to the faulty states, given they are not actually reachable in the program.

In many cases, the techniques permit computing interpolants under some *assumptions*; these can be expressed in the form of variable assignments. The assignment then represents an additional assumption, which serves to exclude from consideration a set of paths. To give an example, consider the code in Fig. 1 and the corresponding ARG in Fig. 2. If it is clear from the context that $i > j$, we can exclude node 4 from consideration by setting the corresponding variable to False. This way, also the over-approximation (interpolant) can be more precise, i.e., focused to this particular *sub-problem*. We call it then a *focused interpolant*.

Focused interpolants can be used also in cases when parallel computational resources are available; then no additional assumptions are needed to exploit the benefits of partial variable assignments. Here, we can used a partial variable assignment to make a case split at conditional branches, where assigning *true* to the statement condition corresponds to the case when the *then* branch is executed, while assigning *false* corresponds to the *else* branch. This way, the program can be split into two (or more, depending on the number of assigned variables) sub-programs, each one referring to a particular situation of the condition validity. Parallel computation of focused interpolants can save the overall time of the verification, as our preliminary experiments suggest.

Craig interpolation [6] is a process for computing over-approximations of propositional formulas that has proven useful in both program verification and automatic abstraction refinement [15]. For an unsatisfiable formula $A \wedge B$, formula I is called Craig interpolant if and only if (1) $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) I contains only variables common to both A and B. The idea of applying Craig interpolation in model checking is to reduce the over-approximation process into finding a compact interpolant I such that I is satisfied by all models of the part being over-approximated (the reachable states up to a given state forming the A part of the input formula), but still entails the properties of interest with respect to the rest of the formula (unreachability of the faulty states—the rest of states).

In this article, we describe *Partial Variable Assignment Interpolants (PVAI)*—a generalization of Craig interpolants—which, in addition to the standard division of an unsatisfiable formula (the *interpolation problem*) into the A and B parts, is parametrized by a *partial variable assignment (PVA)*—an assumption. PVAI have been originally introduced in [13] and their experimental evaluation was provided in [12] describing the tool support. This article, in addition to those papers, (i) provides proofs of various PVAI properties important to program verification, and

```
1: int max(int i, int j) {
2:  if (i > j)
3:   return i;
4:  else
5:   return j;
6: }
   // The main function
7: assert(max(random(), 0) >= 0);
```

Fig. 1 Motivating example

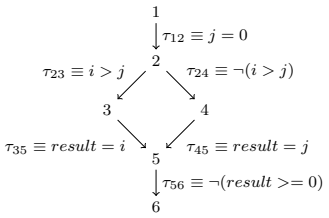

Fig. 2 Abstract reachablity graph

(ii) generalizes the main theorem guaranteeing the path interpolation property of the interpolants generated by the proposed approach.

A PVA defines the *sub-problem* on which a PVAI is focused. A sub-problem is obtained from the interpolation problem by removing the clauses (constraints) satisfied by the assignment. Due to this specialization, (1) the interpolants for the sub-problem can be of smaller size, compared to Craig interpolants computed for the interpolation problem. Moreover, since the satisfied constraints (those not occurring in the sub-problem) need not to be considered by interpolation, (2) it is possible to restrict the variables occurring in an interpolant to those relevant to the sub-problem, i.e. those shared between the A and B parts of the sub-problem, and (3) computing the interpolant can potentially be more efficient, since the computation can exploit the variable assignment, too, by pruning a whole refutation sub-tree.

We also present a framework of *Labeled Partial Assignment Interpolation Systems* (LPAIS)—a generalization of LIS [7], which computes PVAIs for propositional logic. We define the notion of logical strength for LPAISs and show how introducing a partial order over LPAISs allows to systematically compare the strength of the computed interpolants (a feature intuitively relevant to verification since it affects the precision of the over-approximations realized by interpolants [19]). We also show how LPAISs can be used to generate collections of interpolants, which feature the path interpolation property. We evaluated the approach to demonstrate how the assignments (and in particular LPAIS) can be used to reduce the size of interpolants.

1.1 Applications of Craig interpolants

Craig interpolants are usually used in verification to over-approximate the states reachable at the ARG nodes at the boundary between nodes corresponding to the A and B parts. As an example, consider Fig. 2 again. Here, the A part can be formed by, e.g., formulas corresponding to the transitions up to nodes 3 and 4, while the B part by formulas corresponding to the transitions starting at nodes 3 and 4 up to node 6. Let us assume that we want to compute an interpolant for node 3. Depending on the ARG structure, the boundary may or may not include just the considered node (3 in our example), but also many additional ones (4). Moreover, these additional nodes can introduce shared variables that are out of scope at the considered node, that is, they do not exist at the program point corresponding to the considered node (3). Such variables can occur in the Craig

interpolant and need to be subsequently eliminated. Based on the way the ARG is encoded into a formula, a variable assignment can be used to block paths via particular (out-of-scope) ARG nodes. This can be achieved by creating an assignment blocking all paths within the ARG that do not pass a particular node; such paths in ARG cannot influence the reachable states at that node. The assignment is then used as an assumption under which the interpolant (i.e., an over-approximation of reachable states) is computed. This process and the way assignments are generated is described in more details in Sect. 2.

A very similar idea can be applied to function summaries (e.g., FunFrog [25]). A function summary is a formula over input and output variables of the function such that the formula holds if the summarized function for a given input values returns the specified output. Function summaries in FunFrog are computed as follows: first, the BMC formula that encodes all possible (bounded) executions of the program is created. The BMC formula is passed to a SAT (or SMT) solver; if the formula is unsatisfiable, the program is safe (w.r.t. considered assertions) and function summaries are computed as Craig interpolants from the resolution proof of unsatisfiability. To compute the summary for the function f, the BMC formula is partitioned such that the body of summarized function f and the bodies of all functions (resp. their representation in the BMC formula) called from within f belong to the A partition. The rest of the BMC formula belongs to the B partition. Again, based on the way the BMC formula is created, the assignment can be used to eliminate all the traces that do not call the summarized function and added to assumptions when the summary is computed.

Other techniques to reduce the size of interpolants and the size of refutations from which the interpolants are computed utilize inefficiencies in interpolant formulae and in the resolution proofs, respectively. In contrast to them, our technique exploits another type of inefficiency—it depends on inefficiencies of a particular technique resp. in the encoding the technique uses. On one hand, the sensitivity to encoding implies that our technique cannot be used in general (in contrast to other reduction techniques); we require the assumptions (i.e., variable assignment) that need to be created in a technique-dependent way. On the other hand, since the origin of the reduction is orthogonal to other reduction techniques, these techniques complement each other, so it is beneficial to use them together.

2 Motivation

In this section, we illustrate a possible application of PVAI, which originally motivated our work; nonetheless, the proposed PVAIs are not limited to this context. As an example, consider the source code in Fig. 1 and the corresponding ARG in Fig. 2. Node i is associated with location i in the program. We say that a variable is *in-scope* in a node if there is a path through the node where the variable is used both before and after the node. In our example, node 1 is the initial node, while node 6 represents an error location. The *edge constraints* τ_{ij} encode the semantics of the corresponding program statements. Note that τ_{12} originates from the call to the `max` function in `main`, at line 6. Further, in node 3, parameter i is the only in-scope variable; similarly in node 4, parameter j is the only in-scope variable.

In the context of software verification, an important question is whether an error location is actually reachable from the initial location of a program—this is

$$
\begin{array}{ll}
\mu_1 \equiv (n_1 \Rightarrow n_2) & \wedge ((n_1 \wedge n_2) \Rightarrow \tau_{12}) \\
\mu_2 \equiv (n_2 \Rightarrow (n_3 \vee n_4)) & \wedge ((n_2 \wedge n_3) \Rightarrow \tau_{23}) \\
 & \wedge ((n_2 \wedge n_4) \Rightarrow \tau_{24}) \\
\mu_3 \equiv (n_3 \Rightarrow n_5) & \wedge ((n_3 \wedge n_5) \Rightarrow \tau_{35}) \\
\mu_4 \equiv (n_4 \Rightarrow n_5) & \wedge ((n_4 \wedge n_5) \Rightarrow \tau_{45}) \\
\mu_5 \equiv (n_5 \Rightarrow n_6) & \wedge ((n_5 \wedge n_6) \Rightarrow \tau_{56})
\end{array}
$$

$$\mathsf{Cond} \equiv n_1 \wedge \mu_1 \wedge \mu_2 \wedge \mu_3 \wedge \mu_4 \wedge \mu_5$$

Fig. 3 The Cond formula

$$\pi_3 \equiv n_3 \wedge \overline{n_4}$$

$$
\begin{array}{l}
A_3 \equiv\ n_1 \\
\quad \wedge (n_1 \Rightarrow n_2) \wedge\ ((n_1 \wedge n_2) \Rightarrow j = 0) \\
\quad \wedge\ ((n_2 \wedge n_3) \Rightarrow i > j)
\end{array}
$$

$$
\begin{array}{l}
B_3 \equiv (n_3 \Rightarrow n_5) \\
\quad \wedge\ ((n_3 \wedge n_5) \Rightarrow result = i) \\
\quad \wedge (n_5 \Rightarrow n_6) \\
\quad \wedge\ ((n_5 \wedge n_6) \Rightarrow \neg(result >= 0))
\end{array}
$$

Fig. 4 The A and B parts of the sub-problem for node 3

known as the *reachability problem*. The question can be answered by computing, for each node i, the set of states reachable at i via paths in the program ARG [3, 16]. Typically, it is enough to compute an over-approximation of these states, i.e. a *node interpolant*. To compute it, the ARG is converted into a propositional formula (Cond), which represents all execution paths in the ARG. An auxiliary *structure-encoding* Boolean variable n_i is introduced for each node i in the ARG; for each i (except for the error node), a *node formula* μ_i is created, which encodes the labels on the outgoing edges (Fig. 3).

For illustration, we describe the meaning of μ_2. The first conjunct $n_2 \Rightarrow (n_3 \vee n_4)$ expresses that after reaching node 2, a path has to proceed to a successor node (3 or 4). The second conjunct $(n_2 \wedge n_3) \Rightarrow \tau_{23}$ guarantees that if a path goes via the edge $2 \rightarrow 3$, the semantics of the edge is preserved (i.e., the constraint τ_{23} is satisfied). Similarly, the third conjunct enforces the semantics of the edge $2 \rightarrow 4$.

The Cond formula is satisfiable if and only if a feasible path exists that leads from node 1 to node 6 in the ARG. Suppose now that Cond is unsatisfiable; then a node interpolant for each node i can be computed. First, the ARG needs to be partitioned into A and B—so that A corresponds to the antecedents of i, B to all the other nodes in the ARG—and then a Craig interpolant I is generated as an over-approximation of the states reachable at i. For instance, in the case of node 3, A would be set to $n_1 \wedge \mu_1 \wedge \mu_2$ and B to $\mu_3 \wedge \mu_4 \wedge \mu_5$. However, employing standard Craig interpolation in this manner to compute a node interpolant I is not sufficient; out-of-scope variables might in fact belong to both A and B, they could therefore appear in I, and should be consequently eliminated. Variable j, in particular, could appear in the interpolant for node 3. Even though out-of-scope variables can be eliminated by resorting to quantification followed by a quantifier-elimination phase, this phase is a well-known bottleneck in verification [28].

Computing node interpolants using PVAIs effectively solves the problem of out-of-scope program variables. Assume that a node interpolant is to be computed for a node k; a suitable PVA assigns False to all the structure-encoding variables corresponding to the nodes not lying on the paths through k. By setting a variable n_j to False, the paths via node j are blocked; moreover, the whole node formula μ_j is satisfied and thus μ_j is not a part of the sub-problem for node k. On the other hand, the PVA can assign n_k to True to express that each considered path has to pass through k (the node for which the interpolant is computed). In particular, to compute an interpolant for node 3, consider Fig. 4; we assign n_3 to True and n_4 to False to block the path through node 4 (π_3); the rest of variables remain unassigned. This assignment satisfies (and thus removes) $n_2 \Rightarrow (n_3 \vee n_4)$, $(n_2 \wedge n_4) \Rightarrow \tau_{24}$ and

μ_4 (Fig. 3) from the sub-problem (see Fig. 4). In the A part, the sub-problem for node 3 contains the edge labels (and consequently the program state variables) related to the path from node 1 to node 3, and in the B part, information related to the path from node 3 to node 6. The program state variables shared by the A and B parts of the sub-problem are the in-scope variables, which are exactly those that may appear in PVA interpolants.

3 Preliminaries

This section fixes the notation that we will use throughout the paper related to propositional logic, resolution, and interpolation.

A *literal* is a Boolean variable x or its negation $\overline{x}$. We identify the double negation $\overline{\overline{x}}$ with x. A *clause* is a finite set of literals $\{l_1, \ldots, l_n\}$ interpreted as a disjunction $l_1 \vee \ldots \vee l_n$. The empty clause is denoted by $\bot$. Given two clauses Θ, Θ', we write Θ, Θ' to denote the clause $\Theta \cup \Theta'$, and for convenience write also Θ, l to denote the clause $\Theta \cup \{l\}$ for a literal l. In the following, we consider propositional formulas in *Conjunctive Normal Form* (CNF), i.e., as conjunctions (or equivalently sets) of clauses.

We use $\mathsf{var}(l)$ to denote the variable of a literal l and $\mathsf{var}(A)$ for the variables occurring in the set of clauses A. For a finite set X of variables, a *partial variable assignment* (PVA) is a set of literals $\pi \subseteq \{x, \overline{x} \mid x \in X\}$ such that for no $x \in X$ both $x \in \pi$ and $\overline{x} \in \pi$. For a literal l, we say that l is assigned True or $\top$ if $l \in \pi$ and False or $\bot$ if $\bar{l} \in \pi$. When convenient, we interpret the variable assignment as a conjunction of literals. We will often use PVAs as assumptions over a set of clauses C, writing $\pi \models C$, to represent the set of models of C restricted to those satisfying π.

Definition 1 (Clauses under assignment) Let A be a set of clauses and π a PVA over $\mathsf{var}(A)$. We define the sets of

satisfied clauses	$A_\pi = \{\Theta \mid \Theta \in A \text{ and } \pi \models \Theta\}$ and
non-satisfied clauses	$A_\pi^C = \{\Theta \mid \Theta \in A \text{ and } \pi \not\models \Theta\}$.

Satisfied clauses contain at least one literal assigned to $\top$ under π, while for non-satisfied clauses, every literal is either unassigned or falsified.

Example 1 Let $\Phi = (l_1 \vee l_3) \wedge (\bar{l}_2 \vee \bar{l}_6) \wedge (\bar{l}_4 \vee \bar{l}_5) \wedge (\bar{l}_2 \vee l_4) \wedge (\bar{l}_1 \vee l_2)$, and $\pi = \bar{l}_2$. Given the assignment, Φ can be split into $\Phi_\pi = (\bar{l}_2 \vee \bar{l}_6) \wedge (\bar{l}_2 \vee l_4)$ and $\Phi_\pi^C = (l_1 \vee l_3) \wedge (\bar{l}_4 \vee \bar{l}_5) \wedge (\bar{l}_1 \vee l_2)$.

For a set of literals ρ, we write $\neg\rho = \{\bar{l} \mid l \in \rho\}$ for the set consisting of the negations of the literals of ρ. Given a formula F, a PVA π, and a clause $C \in F$ such that $C \cap \pi = \emptyset$, we define the *unit propagation* operator $\mathrm{UP}_C(\pi)$ as

$$\mathrm{UP}_C(\pi) = \begin{cases} \{l\} & \text{if } C \setminus \neg\pi = \{l\} \\ \emptyset & \text{if } |C \setminus \neg\pi| \geq 2, \text{ and} \\ \{x, \overline{x} \mid x \in \mathsf{var}(F \wedge \pi)\} & \text{if } C \subseteq \neg\pi \end{cases}$$

The *unit propagation closure* U is the smallest set of literals over $\mathsf{var}(F \wedge \pi)$ that is closed under $\mathrm{UP}_C(U)$ for all $C \in F$. The *simplification* of F over π, denoted

by $F[\pi]$, is the set of clauses obtained from F by first removing from each clause $C \in F$ the literals l such that $\bar{l} \in U$, and then removing all clauses $C \in F$ such that $U \cap C$ is nonempty.

Let Θ, x and $\Theta', \overline{x}$ be clauses. The *resolvent* clause $Res(\Theta, x, \Theta', \overline{x}, x) = \Theta, \Theta'$ is the result of applying *resolution* on the *antecedent* clauses Θ, x and $\Theta', \overline{x}$ and the *pivot* x. We adopt the definition of a *resolution derivation* from [7]: a *resolution derivation* R for a CNF formula Φ is a tuple (V, E, cl, piv, s), where V is a set of vertices in the derivation, $E \subset V \times V$ is a set of edges forming a full binary DAG (i.e., all the vertices except for the leaves have the in-degree 2). The sink vertex s has the out-degree 0. Each vertex $v \in V$ is associated with a vertex-clause specified by the function $cl(v)$. Each vertex clause of a leaf vertex v corresponds to a clause from input formula Φ (i.e., $cl(v) \in \Phi$). Each inner vertex v represents a resolvent of its antecedent vertex-clauses (specified by cl) using the pivot $piv(v)$; formally, for each inner vertex v there exist edges $(v_1, v), (v_2, v) \in E$ such that $cl(v) = Res(cl(v_1), cl(v_2), piv(v))$. A *refutation* is a resolution derivation such that $cl(s) = \bot$.

Since the refutations take the set of clauses as input, the input formula is first converted into a conjunction of clauses. Therefore, in the following we use the terms formula and set of clauses interchangeably.

Given an unsatisfiable formula F partitioned into two disjoint parts A, B, a *Craig interpolant* [6] is a formula I such that $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) $\mathsf{var}(I) \subseteq \mathsf{var}(A) \cap \mathsf{var}(B)$. An interpolant can be computed from a refutation R of F using the *labeled interpolation system* (LIS) [7]. The system is parametrized by a labeling function L that assigns a label (color) from the set $\{a, b, ab\}$ to each variable occurrence (x, C) of the clauses C in the refutation R. A variable is called *shared* if it occurs both in A and in B; and *local* otherwise. Given an occurrence (x, C) of R, $L(x, C) = a$ if x is local to A and $L(x, C) = b$ if x is local to B. The color of shared variables can be chosen freely for different L. The occurrences in the resolvents are labeled according to the labels in the antecedent clauses. If a variable appears in antecedents with different labels, the label for the new occurrence is ab, and otherwise the label will be the same as in both the antecedents. LIS computes an interpolant by annotating each clause of R with a *partial interpolant* starting with the leaves such that the partial interpolant for a leaf clause C is

$$I(C) = \begin{cases} \bigvee\{l \mid l \in C \text{ and } L(\mathsf{var}(l), C) = b\} & \text{if } C \in A, \text{ and} \\ \bigwedge\{\neg l \mid l \in C \text{ and } L(\mathsf{var}(l), C) = a\} & \text{if } C \in B, \end{cases} \tag{1}$$

and for resolvent clause C with pivot p and antecedents C^+ and C^-, where $p \in C^+$ and $\overline{p} \in C^-$ is

$$I(C) = \begin{cases} I(C^+) \vee I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = a, \\ I(C^+) \wedge I(C^-) & \text{if } L(p, C^+) = L(p, C^-) = b, \text{ and} \\ (I(C^+) \vee p) \wedge (I(C^-) \vee \neg p) & \text{otherwise.} \end{cases} \tag{2}$$

We build the labeled partial variable interpolation system based on LIS, using an additional parameter in the form of a partial variable assignment, in Sec. 5, where we will give the proof of correctness of the new system as a generalization of the correctness of LIS.

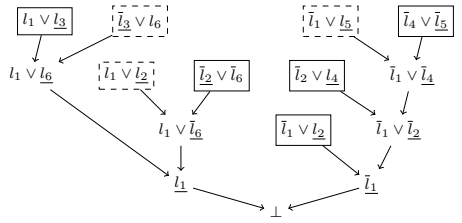

Fig. 5 Resolution refutation; the clauses from A-part and B-part are in dashed and full boxes, respectively.

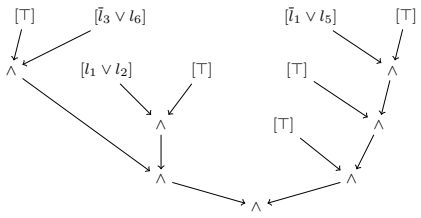

Fig. 6 Derivation of McMillan's interpolant $(l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$.

Example 2 Fig. 5 shows a refutation for formula $\Phi = A \wedge B$ where $A = (l_1 \vee l_2) \wedge (\bar{l}_1 \vee l_5) \wedge (\bar{l}_3 \vee l_6)$, and B as given in Ex. 1. Fig. 6 shows how LIS with the labeling function that labels all occurrences of shared variables as b can be used to construct the interpolant $I_1 = (l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ (after constant propagation) from the refutation in Fig. 5. The interpolants computed with this labeling function are called *McMillan's interpolants*. Note that for convenience, we write the partial interpolant associated to a particular node of the refutation into brackets. Formula $I_2 \equiv \big(l_1 \vee \big((l_6 \vee \bar{l}_3) \wedge (\bar{l}_6 \vee l_2)\big)\big) \wedge (\bar{l}_1 \vee l_5)$ is another interpolant that can be computed by LIS from the refutation; Fig. 9 shows the labels used to compute I_2 (minimal labeling).

Many applications naturally require several interpolants that are computed from a given formula by partitioning the formula differently. Often the resulting interpolants are furthermore required to satisfy certain interdependencies. In particular, an *interpolant sequence* for the unsatisfiable formula $A_1 \wedge \ldots \wedge A_n$ is a tuple of formulas $(I_0, \ldots, I_n)$, where $I_0 = \top$, and for $i \geq 1$, I_i is an interpolant for partitioning $(A_1 \wedge \ldots \wedge A_i, A_{i+1} \wedge \ldots \wedge A_n)$. If for all i, $I_i \wedge A_i \Rightarrow I_{i+1}$, then $(I_0, I_1, \ldots, I_n)$ satisfies the *path interpolation* (PI) property. It can be shown that the path interpolation property holds for any LIS [10], including the well-known McMillan's and Pudlák's systems, whenever the interpolant sequence is computed from the same refutation.

4 Partial Variable Assignment Interpolants

In this section, we formally define *Partial Variable Assignment Interpolation*, which, in addition to the division of an unsatisfiable formula into A and B parts, requires specification of a PVA π defining the sub-problem. Given a verification problem represented by a formula $F \equiv A \wedge B$, a sub-problem w.r.t. assignment π is represented by formula $F^C_\pi \equiv A^C_\pi \wedge B^C_\pi$; i.e., a sub-problem consists only of the unsatisfied part of the original formula. In other words, the choice of the assignment defines to which part of the verification problem to focus. Looking on the motivation example, various assignments can be used to define sub-problems considering e.g., a single ARG path, or all paths via an arbitrary ARG node or all paths via an ARG edge.

Definition 2 (Partial Variable Assignment Interpolant) Let R be a refutation of formula $A \wedge B$ and π be a partial variable assignment over $\mathsf{var}(A \wedge B)$. A *partial variable assignment interpolant* (PVAI) is a formula I such that:

(D2.1) $\pi \models A \Rightarrow I$, or equivalently $\pi \models A^C_\pi \Rightarrow I$
(D2.2) $\pi \models B \wedge I \Rightarrow \bot$, or equivalently $\pi \models B^C_\pi \wedge \neg I \Rightarrow \bot$
(D2.3) $\mathsf{var}(I) \subseteq \mathsf{var}(A^C_\pi) \cap \mathsf{var}(B^C_\pi)$
(D2.4) $\mathsf{var}(I) \cap \mathsf{var}(\pi) = \emptyset$

Since for any set of clauses X, $\pi \models (X \Leftrightarrow X^C_\pi)$, D2.1 and D2.2 can be equivalently rewritten as $\pi \models A^C_\pi \Rightarrow I$ and $\pi \models B^C_\pi \wedge I \Rightarrow \bot$, respectively. Rules D2.3 and D2.4 restrict the variables that may appear in PVAI to the ones shared by the A and B parts of the sub-problem.

In other words, PVAI is a Craig interpolant for the sub-problem, i.e., a single PVAI can be computed using standard interpolation techniques (given a refutation proof of the sub-problem). The above definitions are beneficial to us, since they permit us to reason about interpolants for different sub-problems (i.e., different input formula). Thanks to this property, it is possible to show various properties of PVA (such as the path interpolation property) among interpolants for various sub-problems. This is a unique property, since in other works, only the interpolants computed from same input formula (i.e., verification sub-problem) are considered.

Note that a PVAI cannot be obtained from a standard Craig interpolant of the verification problem AB by simplification ($I[\pi]$); the sub-problem needs to be created (and its refutation proof constructed). The reason is that, in addition to assigned variables (disallowed by D2.4), rule D2.3 also excludes also all unassigned (out-of-scope) variables that occur in satisfied clauses only, which can still appear in $I[\pi]$.

Example 3 Craig and PVA interpolants differ in the variables that can occur in the interpolant. The shared variables between A and B (i.e., those that can appear in a Craig interpolant) are l_1, l_2, l_3, l_5, and l_6. Since PVAI considers (for the shared variables) only non-satisfied parts of A resp. B (i.e., A^C_π and B^C_π), fewer variables are shared; in our example, assuming $\pi \equiv \bar{l}_2$, only l_1, l_3, and l_5 can appear in a PVA interpolant, which are those that can appear in a Craig interpolant for the sub-problem.

Given an assignment ($\pi \equiv \bar{l}_2$) and a Craig interpolant, an alternative way to reduce the interpolant size is to assign the values inside the interpolant formula and propagate the Boolean constants. In this case, the interpolants from the example above result in $I_1[\pi] \equiv l_1 \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ and $I_2[\pi] \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge \bar{l}_6]) \wedge (\bar{l}_1 \vee l_5)$. None of them is a PVA interpolant since each one contains variable l_6. Both of them can be further simplified (i.e., equivalently rewritten) as $I^s_1[\pi] \equiv l_1 \wedge l_5 \wedge (\bar{l}_3 \vee l_6)$ resp. $I^s_2[\pi] \equiv l_1 \vee (\bar{l}_3 \wedge \bar{l}_6) \wedge (\bar{l}_1 \vee l_5)$. In general, such a transformation requires a complex analysis and sometimes the out-of-scope variables can be eliminated from the interpolant by this technique. However, as shown above, variable l_6 cannot be eliminated from the interpolants by these transformations.

The aforementioned techniques can be used to reduce the size of the formula, but it cannot guarantee producing interpolants without the variables appearing just in satisfied clauses, since the information is not present in the interpolant formula.

SAT/SMT calls are resource demanding. A refutation is independent of PVAs; this important fact allows us to call the solver only once on the overall problem Φ, and, later, to compute various PVAs (representing relevant sub-problems) for which the PVAI can be efficiently computed. This follows the idea of having a refutation and several partitionings, for which several (related) interpolants are computed.

Although Craig interpolation has many applications in program verification, verification tools often require interpolant sequences with specific properties [10]. The PVAI for all the sub-problems are computed from the same refutation, thus they are related to each other. The existence of a single refutation permits the application of a standard proving technique in the area of interpolation—structural induction over a refutation—to show various properties of PVA interpolant sequences. All the techniques where interpolants for different sub-problems are computed using different refutations (e.g., applying a solver directly on each sub-problem, or incremental solving with assumptions) do not, per se, guarantee any properties of their sequences. The price to pay is an additional assumption in the form of a partial assignment.

To be complete, there are several ways to generate PVAs, which can be divided into four groups: (1) shrinking the input formula based on the assignment, (2) shrinking (cutting) the refutation proof based on the assignment, (3) quantification (such as in the UFO tool [3]), i.e., use Craig interpolants, apply the assignment onto the interpolant, and quantify out remaining out-of-scope variables, and (4) our PVA approach. The first two options do not guarantee the path interpolation property between related interpolants[1], while the third approach works also for higher-level theories (not just for propositional logic), but the interpolants are quantified formulae. Our approach generates quantifier-free interpolants interrelated by the path interpolation property.

5 Labeled Partial Assignment Interpolation System

In this section, we present the framework of *Labeled Partial Assignment Interpolation Systems*, a generalization of LISs [7], which computes PVAIs for propositional logic, and prove its soundness. Next, in order to prove the path interpolation property, we introduce the concept of logical strength on LPAISs, which allows one to systematically compare the strength of the generated interpolants.

In order to define LPAISs, first we have to extend the definitions of labeling functions and locality from LISs to take variable assignments into account. Note that if no variable is assigned, LPAISs are equivalent to LISs.

A labeling function assigns labels to literals in a refutation; the labeling drives the computation of an interpolant from the refutation and determines its strength (Fig. 7). Note that in the following, if not stated otherwise, we assume just a single refutation being re-used for computing many interpolants.

[1] Our intuition leads us to the conclusion that the path interpolation property cannot be guaranteed in a general case of different assignments, but we have no counterexample demonstrating it.

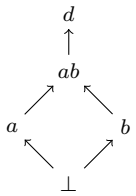

Fig. 7 Lattice of labels ($\sqcup$)

5.1 Definitions

Definition 3 (Labeling function) Let $L = (S, \sqsubseteq, \sqcap, \sqcup)$ be the lattice in Fig. 7, where $S = \{\bot, a, b, ab, d\}$ and $\bot$ is the least element, and let $R = (V, E, cl, piv, s)$ be a refutation over a set of literals Lit. Function $\mathsf{Lab}_{R,L} : V \times \mathsf{Lit} \to S$ is called *labeling function* for refutation R iff $\forall v \in V$ and $\forall l \in \mathsf{Lit}, \mathsf{Lab}_{R,L}$ satisfies the following conditions:

(D3.1) $\mathsf{Lab}_{R,L}(v, l) = \bot$ if and only if $l \notin cl(v)$, and
(D3.2) $\mathsf{Lab}_{R,L}(v, l) = \mathsf{Lab}_{R,L}(v_1, l) \sqcup \mathsf{Lab}_{R,L}(v_2, l)$, where v_1, v_2 are the antecedent vertices.

From condition D3.2 it follows that the labeling function is fully determined once the labels in the leaves have been specified. We omit subscripts R and L if clear from the context.

Naming conventions. Let us assume a pair of sets of clauses (A, B) and a PVA π. The clause sets are split into four groups, the non-satisfied clauses A^C_π and B^C_π, which specify the sub-problem and are taken into account during interpolation, and the satisfied clauses A_π and B_π, which are disregarded.

We distinguish among the following kinds of variables, depending on the standard notions of locality and sharedness, as well as on where the variables appear in the four groups of clauses. We say that a variable k is *unassigned* if $k \notin \mathsf{var}(\pi)$. An unassigned variable k is:

A^C_π*-local*	if $k \in \mathsf{var}(A^C_\pi)$ and $k \notin \mathsf{var}(B^C_\pi)$
B^C_π*-local*	if $k \notin \mathsf{var}(A^C_\pi)$ and $k \in \mathsf{var}(B^C_\pi)$
$A^C_\pi B^C_\pi$*-shared*	if $k \in \mathsf{var}(A^C_\pi)$ and $k \in \mathsf{var}(B^C_\pi)$
$A^C_\pi B^C_\pi$*-clean*	if $k \notin \mathsf{var}(A^C_\pi)$ and $k \notin \mathsf{var}(B^C_\pi)$

The properties above are independent of the occurrence of k in $\mathsf{var}(A_\pi)$ and $\mathsf{var}(B_\pi)$. The "clean" variables occur only in the satisfied clauses, thus are out-of-scope and cannot appear in PVA interpolants.

Definition 4 We say that variable k is *McMillan-labeled* if, whenever k is $A^C_\pi B^C_\pi$-shared or $A^C_\pi B^C_\pi$-clean, it is labeled b.

Note that the labels of the other variables are not limited to b. If all variables are McMillan-labeled, LIS reduces to McMillan's interpolation system [7], which yields the strongest interpolant that LISs (and LPAISs) can produce from a given refutation.

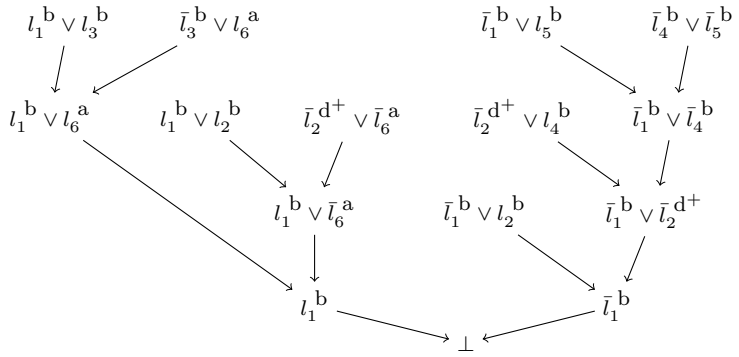

Fig. 8 McMillan's labeling for (A, B, π), where $\pi \equiv \overline{l_2}$.

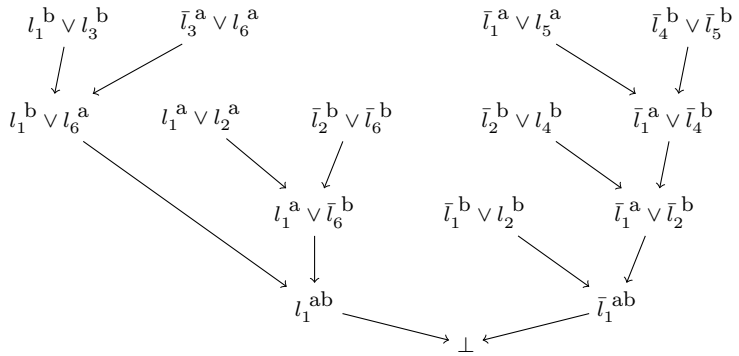

Fig. 9 Minimal labeling for (A, B) and empty assignment.

Definition 5 Variable k is labeled *consistently* if all occurrences of k in a refutation have the same label:
$\forall x, x' \in V, l \in cl(x), l' \in cl(x') : \mathsf{var}(l) = \mathsf{var}(l') \Rightarrow \mathsf{Lab}(v, l) = \mathsf{Lab}(v', l')$.

Example 4 Fig. 8 shows how a labeling function assigns labels to literals; the label of a literal is shown in superscript. Again, we assume $\pi \equiv \overline{l_2}$. We choose the strongest possible labeling (which for an empty assignment would produce McMillan's interpolant I_1); in particular $A_\pi^C B_\pi^C$-shared and $A_\pi^C B_\pi^C$-clean variables are labeled b. Note that variable l_6 is A_π^C-local and thus has to be labeled a, the $A_\pi^C B_\pi^C$-shared variables are l_1, l_3, and l_5, no variable is $A_\pi^C B_\pi^C$-clean, and variable l_4 is B_π^C-local. Fig. 9 shows another example of labeling; note the clauses l_1 and $\bar{l}_1$ illustrating how the labels are merged from the labels in antecedents.

Not all labeling functions can be used to generate interpolants; in LPAIS, interpolants are computed if a locality preserving labeling is used.

Definition 6 Let R be a refutation of formula $A \wedge B$ and π be a PVA. Labeling function Lab for refutation R is *locality preserving* iff $\forall v \in V, \forall l \in cl(v)$ all the following locality constraints are satisfied:

(D6.1) $\mathsf{Lab}(v, l) = d \Leftrightarrow l \in \pi$
(D6.2) $l \notin \pi$ and A^C_π-local $\Rightarrow \mathsf{Lab}(v, l) = a$
(D6.3) $l \notin \pi$ and B^C_π-local $\Rightarrow \mathsf{Lab}(v, l) = b$
(D6.4) $l \notin \pi$ and $A^C_\pi B^C_\pi$-clean $\Rightarrow$ it is consistently labeled a or b.

Locality constraints provide freedom in labeling $A^C_\pi B^C_\pi$-shared and $A^C_\pi B^C_\pi$-clean variables; the choice of labels directly affects the strength of the computed interpolants. The label of $A^C_\pi B^C_\pi$-shared variables can be set freely to a, b, or ab. The same holds for falsified literals; their labels are irrelevant since they are removed by the assignment filter (defined below).

D6.2 and D6.3 are equivalent to the locality requirements of LIS, where A-local and B-local variables must be labeled a and b, respectively. D6.1 concerns the satisfied literals. Label d is used in the interpolation process to identify resolutions with an assigned pivot and parts of the refutation that are not relevant to the sub-problem. D6.4 is specific to PVAI and deals with variables that occur in the satisfied clauses only. The requirement guarantees that such variables do not occur in the interpolant, because Res-ab (see Tab. 1) cannot be applied. Further, note that for the empty assignment, the locality constraints reduce to those of LISs, since D6.1 and D6.4 do not apply to any literal.

Filters. For a clause Θ, a labeling function Lab, a refutation vertex $v \in V$, and a label c, we define the *match filter* $|$, which preserves only the literals with the specified label, as follows:

$$\Theta|_{c,v,\mathsf{Lab}} = \{l \in \Theta \mid c = \mathsf{Lab}(v, l)\}$$

Similarly, we define the *upward filter* $\restriction$, which preserves the literals with labels greater than c (Fig. 7), as:

$$\Theta\restriction_{c,v,\mathsf{Lab}} = \{l \in \Theta \mid c \sqsubseteq \mathsf{Lab}(v, l)\}$$

The subscripts Lab, v are omitted if clear from the context. Given a partial assignment π and a clause Θ, we also define the *assignment filter*, which removes all the assigned literals (satisfied and falsified ones), as follows:.

$$\Theta[\pi] = \{l \in \Theta \mid \mathsf{var}(l) \notin \mathsf{var}(\pi))\}$$

Moreover, in our notation, we assume that filters have a higher precedence than negation. E.g., $\neg\Theta[\pi]\restriction_a$ can be equivalently rewritten as $\neg((\Theta[\pi])\restriction_a)$.

Interpolation system. An interpolation system is a procedure for computing an interpolant given a refutation. It assigns a partial *vertex-interpolant* to each vertex of the refutation, yielding the final interpolant at the sink vertex.

Definition 7 Let R be a refutation of $A \wedge B$, π be a PVA, and Lab be a corresponding locality preserving labeling function. Then, Tab. 1 defines the *Labeled Partial Assignment Interpolation System* $\mathsf{Lpaltp}(\mathsf{Lab}, R, A, B, \pi)$.

LPAIS produces interpolants in the following way: First, the vertex-interpolants for leaves of the refutation are computed using the rules in the upper part of Tab. 1 (*Hyp*othesis rules). Depending on the occurrence of the vertex-clause Θ in the A

Leaf v:	$\Theta, [I]$	
$I = \begin{cases} \Theta[\pi]\rVert_{b,v,\mathsf{Lab}} \\ \neg\Theta[\pi]\rVert_{a,v,\mathsf{Lab}} \\ \top \end{cases}$	$\begin{array}{l} \text{if } \Theta \in A_\pi^C \\ \text{if } \Theta \in B_\pi^C \\ \text{if } \Theta \in A_\pi \cup B_\pi \end{array}$	$\begin{array}{l} \text{Hyp-}A_\pi^C \\ \text{Hyp-}B_\pi^C \\ \text{Hyp-}A_\pi, \text{ Hyp-}B_\pi \end{array}$
Inner vertex v:	$\dfrac{v_1 : p, \Theta_1, [I_1] \qquad v_2 : \overline{p}, \Theta_2, [I_2]}{\Theta_1, \Theta_2, [I]}$	
$I = \begin{cases} I_1 \vee I_2 \\ I_1 \wedge I_2 \\ (I_1 \vee p) \wedge (I_2 \vee \overline{p}) \\ I_2 \\ I_1 \end{cases}$	$\begin{array}{l} \text{if } \mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a \\ \text{if } \mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b \\ \text{if } \mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab \\ \text{if } \mathsf{Lab}(v_1, p) = d \\ \text{if } \mathsf{Lab}(v_2, \overline{p}) = d \end{array}$	$\begin{array}{l} \text{Res-}a \\ \text{Res-}b \\ \text{Res-}ab \\ \text{Res-}d \\ \text{Res-}d \end{array}$

Table 1 Hypothesis and resolution rules for Labeled Partial Variable Assignment Interpolation System

or B sets, the corresponding rule describes the transformation of the vertex-clause into a partial vertex-interpolant. Later, going down through the refutation from the leaves to the sink, the vertex-interpolants for inner vertices are computed using the resolution rules in the lower part of Tab. 1. The labels assigned to the pivots determine how vertex-interpolants of both antecedents are combined. This process ends at the sink vertex where the PVAI is derived. The interpolants are computed in time linear to the size of the refutation.

The main difference compared to LISs are the additional d rules. For instance, consider the last rule $\mathsf{Lab}(v_2, \overline{p}) = d$ in Tab. 1. In contrast to the original LIS rules, the partial vertex-interpolant is simpler, because it does not contain I_2, omitted due to the variable assignment. Generally, these rules *cut out* the satisfied subtree of the refutation. Usually, the later in the refutation the assigned variable is resolved, the larger sub-tree is pruned and the smaller the resulting interpolant is.

The differences between LPAISs and LISs are motivated by the way variable assignments work. The new d rules can be seen as a specialization of the ab resolution rule if PVA π is assumed. A similar relationship holds for the hypothesis rules in the leaves of a refutation. These rules are equivalent to LIS hypothesis rules if applied on a clause under the assumed assignment. The changes we introduce w.r.t. LISs are of two kinds: (i) those in LPAIS rules force specialization of the interpolant on a sub-problem, and (ii) the changes in the locality constraints remove unassigned out-of-scope variables from the interpolant.

Example 5 Figs. 10 and 11 show the rules that apply to a vertex; the labels are taken from Figs. 8 resp. 9. Fig. 12 shows how LPAIS produces interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$ for our example using labeling of Fig. 10. Note the dotted arrows at vertices corresponding to Res-d resolutions; they highlight the antecedents whose partial vertex-interpolants are ignored and their sub-trees do not contribute to the final PVA interpolant. Also note that the PVA interpolant I_π is smaller compared to both $I_1[\pi]$ and $I_2[\pi]$ from examples above.

5.2 Correctness

Theorem 1 (Correctness) *Let R be a refutation of $A \wedge B$, π be a PVA, and* Lab *be a locality preserving labeling function. Then,* $\mathsf{Lpaltp}(\mathsf{Lab}, R, A, B, \pi)$ *generates a partial variable assignment interpolant at the sink vertex s.*

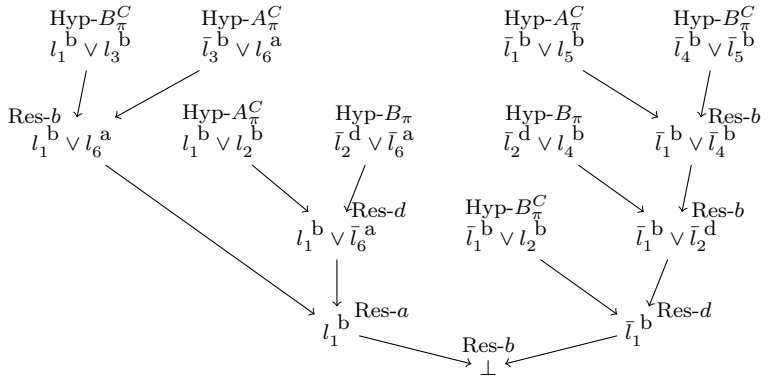

Fig. 10 Rules applied at refutation vertices if McMillan's labeling of Fig. 8 is used.

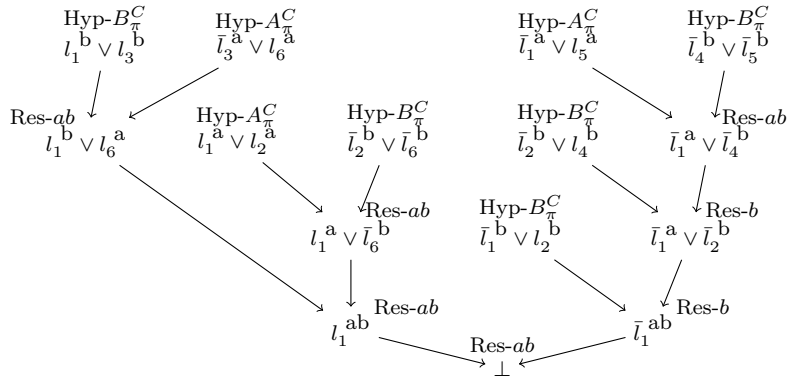

Fig. 11 Rules applied at refutation vertices if minimal labeling of Fig. 9 is used.

Proof (Theorem 1—Correctness). In the proof, we follow the proof idea of LIS. By structural induction, we show that for each vertex v of a resolution proof the following invariants hold:

(T1.Inv1) $\pi \models A \wedge \neg\Theta\restriction_{a,v,\mathsf{Lab}} \Rightarrow I$
(T1.Inv2) $\pi \models B \wedge \neg\Theta\restriction_{b,v,\mathsf{Lab}} \Rightarrow \neg I$
(T1.Inv3) $\mathsf{var}(I) \subseteq \mathsf{var}(A_\pi^C) \cap \mathsf{var}(B_\pi^C)$

where I is the partial interpolant of vertex v and $cl(v) = \Theta$.

These invariants are equivalent to the PVAI constraints for the sink node (where the $\neg\Theta = \top$). We omit the labeling function Lab from subscripts (since it is unique in the proof) and the vertex if clear.

Base cases. The base cases apply to the leaf vertices of the proof where the hypotheses operations are applied.

Hyp-A_π^C: $\Theta \in A_\pi^C$ so $I = \Theta[\pi]\!\|_b$

(T1.Inv1) $\pi \models A \wedge \neg\Theta\restriction_a \Rightarrow \Theta[\pi]_b$ holds because $A \Rightarrow \Theta$ and $\Theta \Leftrightarrow (\Theta\restriction_a \vee \Theta|_b)$, so $\Theta \wedge \neg\Theta\restriction_a \Rightarrow \Theta|_b$. Moreover, it holds that $\pi \models \Theta|_b \Leftrightarrow \Theta[\pi]_b$ because the clause Θ (thus even $\Theta|_b$) is not satisfied by the partial assignment π, so all the assigned literals (i.e., those removed by the filter $[\pi]$) evaluate to $\bot$.

(T1.Inv2) $\pi \models B \wedge \neg\Theta\restriction_b \Rightarrow \neg\Theta[\pi]_b$ holds because $\neg\Theta\restriction_b \Rightarrow \neg\Theta|_b$. Moreover, it holds that $\pi \models \neg\Theta|_b \Leftrightarrow \Theta[\pi]_b$; the reason is the same as above, all the assigned literals evaluate to $\bot$.

(T1.Inv3) $\mathsf{var}(\Theta[\pi]_b) \subseteq \mathsf{var}(A^C_\pi) \cap \mathsf{var}(B^C_\pi)$. Label b implies that such variables are $A^C_\pi B^C_\pi$-shared. Otherwise, the locality-preserving requirement D6.2 is violated. Moreover, the assignment filter is applied, so the partial vertex-interpolant does not contain any assigned variable.

Hyp-B^C_π: $\Theta \in B^C_\pi$ so $I = \neg\Theta[\pi]_a$. The situation is symmetric to Hyp-A^C_π case.

(T1.Inv1) $\pi \models A \wedge \neg\Theta\restriction_a \Rightarrow \neg\Theta|_a$ holds because $\neg\Theta\restriction_a \Rightarrow \neg\Theta|_a$. Moreover, $\pi \models \Theta|_a \Leftrightarrow \Theta[\pi]_a$, because all the assigned literals in the clause Θ evaluate to $\bot$ under the assignment π.

(T1.Inv2) $\pi \models B \wedge \neg\Theta\restriction_b \Rightarrow \Theta|_a$ holds because $B \Rightarrow \Theta$ and $\Theta \Leftrightarrow (\Theta\restriction_b \vee \Theta|_a)$ so $\Theta \wedge \neg\Theta\restriction_b \Rightarrow \Theta|_a$. Moreover, as shown above, $\pi \models \Theta|_a \Leftrightarrow \Theta[\pi]_a$.

(T1.Inv3) $\mathsf{var}(\neg\Theta[\pi]_a) \subseteq \mathsf{var}(A^C_\pi) \cap \mathsf{var}(B^C_\pi)$. The label a implies that these variables are $A^C_\pi B^C_\pi$-shared. Otherwise, the locality preserving requirements D6.3 is violated. Moreover, the assignment filter is applied, so the partial vertex-interpolant does not contain any assigned variable.

Hyp-A_π, Hyp-B_π: $\Theta \in A_\pi \cup B_\pi$ so $I = \top$.

(T1.Inv1) $\pi \models A \wedge \neg\Theta\restriction_a \Rightarrow \top$ holds trivially.

(T1.Inv2) $\pi \models B \wedge \neg\Theta\restriction_b \Rightarrow \bot$.
We show that the antecedents of the implication are unsatisfied. The reason is that $\neg\Theta\restriction_b$ evaluates (is equivalent) to $\bot$ under assignment π.
From $\Theta \in A_\pi$ (resp. $\Theta \in B_\pi$) it follows that exists literal $l \in \Theta$ such that $\pi \models l$; the literal l makes the clause Θ satisfied under π. The label of l is d (locality of labeling function—D6.1) so the literal is preserved by the upward-filter $\restriction_b$. Thus $\pi \models \neg\Theta\restriction_b \Leftrightarrow \bot$.

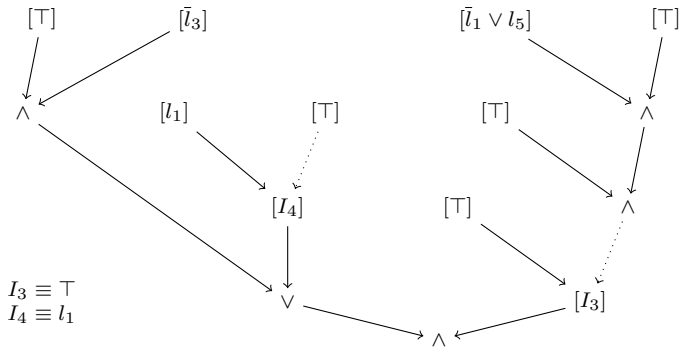

Fig. 12 PVA interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$, when using labeling of Fig. 10.

(T1.Inv3) $\mathsf{var}(\top) \subseteq \mathsf{var}(A) \cap \mathsf{var}(B)$ holds trivially.

□

Before the proof of Theorem 1 continues (i.e., moves from leaves to inner vertices), we introduce auxiliary lemmas. The first one introduces upward-filter for pivot variables. The second lemma connects the antecedents of the invariant implications of the current vertex and the antecedent vertices.

Lemma 1 (Introducing upward-filters) *Let p be a variable, v be a vertex, and c be a label ($c \in L$). It holds:*

$$\models p \Rightarrow \neg\overline{p}\restriction_{c,v} \qquad \textit{and} \qquad \models \overline{p} \Rightarrow \neg p\restriction_{c,v}$$

Proof (Lemma 1). The upward-filter $\restriction_{c,v}$ can either preserve the literal $\overline{p}$ or filter it out. In the first case, the filter evaluates to $\neg\overline{p}$, which is equivalent to p and the implication $\models p \Rightarrow p$ holds trivially. In the second case, the filter evaluates to the empty clause, i.e., $\bot$ and the implication $\models p \Rightarrow \neg\bot$ holds trivially.

The same reasoning applies to the second formula.

□

Lemma 2 (Filters in antecedent vertices) *Let $R \equiv (V, E, cl, piv, s)$ be a resolution proof and $\mathsf{Lab}_{R,L}$ be a labeling function for proof R. Let $v \in V$ be inner vertex of the proof with vertex clause $cl(v) = \Theta_1, \Theta_2$. Let vertices v_1 and v_2 be the antecedents of vertex v and their vertex clauses be $cl(v_1) = p, \Theta_1$ resp. $cl(v_2) = \overline{p}, \Theta_2$. Let c be a label ($c \in L$). Then the following holds:*

$$\neg p\restriction_{c,v_1} \wedge \neg\Theta_1, \Theta_2\restriction_{c,v} \Rightarrow \neg p, \Theta_1\restriction_{c,v_1} \qquad \textit{and}$$
$$\neg\overline{p}\restriction_{c,v_2} \wedge \neg\Theta_1, \Theta_2\restriction_{c,v} \Rightarrow \neg\overline{p}, \Theta_2\restriction_{c,v_2}$$

Proof (Lemma 2). The upward-filter $\restriction$ preserves all the literals whose label equals to or is greater than the given label (e.g., $\restriction_a$ preserves literals with labels a, ab, d). From the definition of labeling function (in particular from the conditions D3.1 and D3.2) it follows that $\forall l \in \Theta_1, \Theta_2 : \mathsf{Lab}(v_1, l) \sqsubseteq \mathsf{Lab}(v, l)$. So, the literals preserved by the upward filter in the vertex v_1 (excluding the pivot) are also preserved by the upward filter in the successor vertex v. Thus, it follows that $\Theta_1, \Theta_2\restriction_{c,v_1} \Rightarrow \Theta_1, \Theta_2\restriction_{c,v}$, which can be equivalently rewritten into contrapositive implication $\neg\Theta_1, \Theta_2\restriction_{c,v} \Rightarrow \neg\Theta_1, \Theta_2\restriction_{c,v_1} \Rightarrow \neg\Theta_1\restriction_{c,v_1}$.

The implication $\neg p\restriction_{c,v_1} \wedge \neg\Theta_1, \Theta_2\restriction_{c,v} \Rightarrow \neg p, \Theta_1\restriction_{c,v_1}$ holds, because the same filter is applied onto literal p (it is either filtered out or preserved by both filters).

The same reasoning applies to the second formula.

□

Proof (Theorem 1—Correctness—cont.).

Induction hypothesis. Now, we will focus on the inductive step. Let v be an inner vertex of the proof and let variable p be the pivot of the resolution at vertex v (i.e., $p = piv(v)$). Let vertex v_1 be the antecedent of v with the vertex-clause containing the pivot positively (i.e., $cl(v_1) = p, \Theta_1$) and let vertex v_2 be the antecedent of v

having negated pivot in its vertex-clause (i.e., $cl(v_2) = \overline{p}, \Theta_2$). From the induction hypothesis, we know that for the antecedent vertices, the following invariants hold:

$$\begin{array}{l} \pi \models A \wedge \neg p, \Theta_1\restriction_{a,v_1} \Rightarrow I_1 \quad \text{and} \quad \pi \models B \wedge \neg p, \Theta_1\restriction_{b,v_1} \Rightarrow \neg I_1 \quad \text{and} \\ \pi \models A \wedge \neg \overline{p}, \Theta_2\restriction_{a,v_2} \Rightarrow I_2 \quad \text{and} \quad \pi \models B \wedge \neg \overline{p}, \Theta_2\restriction_{b,v_2} \Rightarrow \neg I_2 \end{array} \tag{IH}$$

For each type of the resolution, we establish the induction invariants for vertex v.

Res-a: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$ so $I = I_1 \vee I_2$.
In this case, pivot variable p has label a in both antecedents v_1 and v_2.

(T1.Inv1) It follows that:

$$\begin{aligned} \pi \models \overline{p} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} &\overset{\text{(L1)}}{\Rightarrow} \neg\overline{p}\restriction_{a,v_1} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \overset{\text{(L2)}}{\Rightarrow} \\ &\Rightarrow A \wedge \neg p, \Theta_1\restriction_{a,v_1} \overset{\text{(IH)}}{\Rightarrow} I_1 \\ \pi \models p \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} &\overset{\text{(L1)}}{\Rightarrow} \neg\overline{p}\restriction_{a,v_2} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \overset{\text{(L2)}}{\Rightarrow} \\ &\Rightarrow A \wedge \neg\overline{p}, \Theta_2\restriction_{a,v_2} \overset{\text{(IH)}}{\Rightarrow} I_2 \end{aligned}$$

The first implication is application of Lemma 1. The second implication is application of Lemma 2 and the last one is the induction hypothesis.
From the previous implications, it directly follows that:

$$\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Leftrightarrow (\overline{p} \vee p) \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Rightarrow (I_1 \vee I_2)$$

The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above.

(T1.Inv2) Because the label of the pivot in the antecedents is a, it follows that $\neg p\restriction_{b,v_1} \Leftrightarrow \neg\overline{p}\restriction_{b,v_2} \Leftrightarrow \top$. Thus, Lemma 2 can be applied directly without any additional assumptions:

$$\begin{aligned} \pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Leftrightarrow \neg p\restriction_{b,v_1} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L2)}}{\Rightarrow} \\ &\Rightarrow B \wedge \neg p, \Theta_1\restriction_{b,v_1} \overset{\text{(IH)}}{\Rightarrow} \neg I_1 \qquad (1) \\ \pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Leftrightarrow \neg\overline{p}\restriction_{b,v_2} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L2)}}{\Rightarrow} \\ &\Rightarrow B \wedge \neg\overline{p}, \Theta_2\restriction_{b,v_2} \overset{\text{(IH)}}{\Rightarrow} \neg I_2 \qquad (2) \end{aligned}$$

$$\pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Rightarrow (\neg I_1 \wedge \neg I_2) \Leftrightarrow \neg(I_2 \vee I_2)$$

The first implication follows from the equations (1) and (2). The second equivalence factors out the negation.

(T1.Inv3) The third requirement (shared variables only) holds trivially. No new variable is added into the partial vertex-interpolant.

Res-b: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$ so $I = I_1 \wedge I_2$.
The proof is symmetric to the Res-a case. In this case the pivot variable has the label b in both antecedents v_1 and v_2.

(T1.Inv1) The label of the pivot in the antecedents is b so it holds:
$\neg p\restriction_{a,v_1} \Leftrightarrow \neg\overline{p}\restriction_{a,v_2} \Leftrightarrow \top$.

$$\begin{aligned}
\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Leftrightarrow \neg p\restriction_{a,v_1} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} &\overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow A \wedge \neg p, \Theta_1\restriction_{a,v_1} &\overset{\text{(IH)}}{\Rightarrow} I_1 \\
\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Leftrightarrow \neg\overline{p}\restriction_{a,v_2} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} &\overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow A \wedge \neg\overline{p}, \Theta_2\restriction_{a,v_2} &\overset{\text{(IH)}}{\Rightarrow} I_2
\end{aligned}$$

These equations directly yield the result:

$$\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Rightarrow (I_1 \wedge I_2)$$

(T1.Inv2) It follows that:

$$\begin{aligned}
\pi \models \overline{p} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L1)}}{\Rightarrow} \neg p\restriction_{b,v_1} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow B \wedge \neg p, \Theta_1\restriction_{b,v_1} &\overset{\text{(IH)}}{\Rightarrow} \neg I_1 \\
\pi \models p \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L1)}}{\Rightarrow} \neg\overline{p}\restriction_{b,v_2} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow B \wedge \neg\overline{p}, \Theta_2\restriction_{b,v_2} &\overset{\text{(IH)}}{\Rightarrow} \neg I_2
\end{aligned}$$

From the previous implications, it directly follows that:

$$\begin{aligned}
\pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Leftrightarrow (\overline{p} \vee p) \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\Rightarrow \\
\Rightarrow (\neg I_1 \vee \neg I_2) \Leftrightarrow \neg(I_1 \wedge I_2)&
\end{aligned}$$

The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above, while the last equivalence factors out the negation.

Res-ab: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$, so $I = (p \vee I_1) \wedge (\overline{p} \vee I_2)$.

(T1.Inv1) It follows that:

$$\begin{aligned}
\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Rightarrow p \vee (\overline{p} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v}) &\overset{\text{(L1)}}{\Rightarrow} \\
\Rightarrow p \vee (\neg p\restriction_{a,v_1} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v}) &\overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow p \vee (A \wedge \neg p, \Theta_1\restriction_{a,v_1}) &\overset{\text{(IH)}}{\Rightarrow} (p \vee I_1) \\
\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Rightarrow \overline{p} \vee (p \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v}) &\overset{\text{(L1)}}{\Rightarrow} \\
\Rightarrow \overline{p} \vee (\neg\overline{p}\restriction_{a,v_2} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v}) &\overset{\text{(L2)}}{\Rightarrow} \\
\Rightarrow \overline{p} \vee (A \wedge \neg\overline{p}, \Theta_2\restriction_{a,v_2}) &\overset{\text{(IH)}}{\Rightarrow} (\overline{p} \vee I_2)
\end{aligned}$$

The first implication is a logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication is application of Lemma 1. The third implication is application of Lemma 2 and the last one is the induction hypothesis. From the implications above, it directly follows that:

$$\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v}) \Rightarrow (p \vee I_1) \wedge (\overline{p} \vee I_2)$$

(T1.Inv2) Similarly to the previous case:

$$\begin{aligned}
\pi \models \overline{p} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L1)}}{\Rightarrow} \overline{p} \wedge (\neg\overline{p}\restriction_{b,v_1} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v}) \overset{\text{(L2)}}{\Rightarrow} \\
&\Rightarrow \overline{p} \wedge (B \wedge \neg p, \Theta_1\restriction_{b,v_1}) \overset{\text{(IH)}}{\Rightarrow} \\
&\Rightarrow \overline{p} \wedge (\neg I_1) \Leftrightarrow \neg(p \vee I_1) \\
\pi \models p \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} &\overset{\text{(L1)}}{\Rightarrow} p \wedge (\neg\overline{p}\restriction_{b,v_2} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v}) \overset{\text{(L2)}}{\Rightarrow} \\
&\Rightarrow p \wedge (B \wedge \neg\overline{p}, \Theta_2\restriction_{b,v_2}) \overset{\text{(IH)}}{\Rightarrow} \\
&\Rightarrow p \wedge (\neg I_2) \Leftrightarrow \neg(\overline{p} \vee I_2)
\end{aligned}$$

In the first implication, the conjunct $\overline{p}$ is duplicated and then, Lemma 1 is applied. The last implication is simple logical equality.

$$\begin{aligned}
\pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Leftrightarrow (\overline{p} \vee p) \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Rightarrow \\
\Rightarrow \neg(p \vee I_1) \vee \neg(\overline{p} \vee I_2) \Leftrightarrow \neg((p \vee I_1) \wedge (\overline{p} \vee I_2))
\end{aligned}$$

The same reasoning as in the Res-a (T1.Inv1) case is used. The first equivalence is a simple logical consequence of $p \vee \overline{p} \Leftrightarrow \top$. The second implication follows from the two equations above. The last equivalence just factors out the negation.

(T1.Inv3) Variable p is the only new variable added into the interpolant. Variable p is shared (because of its label ab), thus the requirements are met. Moreover, variable p is not assigned. If it would be assigned, it would be labeled d in one of the antecedents, which would lead to the Res-d resolution.

Res-d: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d$ so $I = I_1$ resp. $I = I_2$.

In this case, the pivot variable is assigned by PVA π. Labeling function Lab is locality preserving and constraint D6.1 give us that there is exactly one antecedent where the pivot is labeled d. Assume that $\mathsf{Lab}(v_1, p) = d$, so it holds that $\pi \models p$; the case $\mathsf{Lab}(v_2, \overline{p}) = d$ is symmetric.

(T1.Inv1) It follows that:

$$\begin{aligned}
\pi \models A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} \Leftrightarrow \neg\overline{p}\restriction_{a,v_2} \wedge A \wedge \neg\Theta_1, \Theta_2\restriction_{a,v} &\overset{\text{(L2)}}{\Rightarrow} \\
&\Rightarrow A \wedge \neg\overline{p}, \Theta_2\restriction_{a,v_2} \overset{\text{(IH)}}{\Rightarrow} I_2
\end{aligned}$$

The first equivalence holds because $\pi \models \neg\overline{p}\restriction_{a,v_2}$; the $\neg\overline{p}\restriction_{a,v_2}$ is either directly $\top$ if the $\overline{p}$ literal is not preserved by the upward-filter or it is p if the $\overline{p}$ literal is preserved by the filter $\restriction_{a,v_2}$. In the latter case, p is satisfied under π.

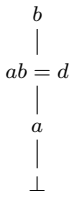

Fig. 13 Strength ordering ($\preceq$)

(T1.Inv2) Similarly to the previous case, it holds:

$$\pi \models B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \Leftrightarrow \neg\overrightarrow{p}\restriction_{b,v_2} \wedge B \wedge \neg\Theta_1, \Theta_2\restriction_{b,v} \overset{\text{(L2)}}{\Rightarrow}$$
$$\Rightarrow B \wedge \neg\overline{p}, \Theta_2\restriction_{b,v_2} \overset{\text{(IH)}}{\Rightarrow} \neg I_2$$

(T1.Inv3) This condition holds trivially from the induction hypothesis.

To summarize, we have shown that all the resolutions and hypotheses establish the inductive invariant for its partial vertex-interpolant. Thus, the inductive invariant also holds for the sink vertex s, where $cl(s) = \emptyset = \bot$; the inductive invariant establishes Theorem 1 for the sink vertex.

□

Symmetry. Notice that the locality constraints, as well as the way LPAISs compute interpolants, are *symmetric* in terms of presence of formulas in the A_π and B_π sets of satisfied clauses. It reflects the fact that these clauses are not a part of the sub-problem under consideration, thus irrelevant for PVAI interpolants. Given a fixed π, the satisfied clauses can be moved freely between the A and B sets; both computed interpolants and locality of the labeling functions are unaffected if satisfied clauses are moved. This fact allows us to articulate the *strength* theorem in an elegant way.

5.3 Interpolant strength

Interpolation systems based on labeling provide some freedom in the choice of labels (e.g., for shared variables); this choice affects the resulting interpolants, in particular their logical strength. In the following, we investigate this relationship in more detail.

Definition 8 (Strength order) Let $\preceq$ be a pre-order relation defined over the set of labels $S = \{\bot, a, b, ab, d\}$ as: $b \preceq ab = d \preceq a \preceq \bot$ (Fig. 13). Let Lab and Lab' be labeling functions for a refutation R. We say Lab *is stronger than* Lab', denoted as $\mathsf{Lab} \preceq \mathsf{Lab}'$, if for all vertices $v \in V$ and for all literals $l \in cl(v)$ it holds that $\mathsf{Lab}(v, l) \preceq \mathsf{Lab}'(v, l)$.

Note that labels ab and d are of the same strength and can be exchanged if the locality requirements permit it; b is the strongest label, while a is the weakest one a literal can get. The following theorem states that the introduced strength order on labeling functions also induces ordering of the produced interpolants by logical strength.

Theorem 2 (Interpolant strength) *Let R be a refutation of $A \wedge B$, π and π' be PVAs, and Lab and Lab' be corresponding locality preserving labeling functions. Let I be a partial variable assignment interpolant for $\mathsf{Lpaltp}(\mathsf{Lab}, R, A, B, \pi)$ and I' be a PVAI for $\mathsf{Lpaltp}(\mathsf{Lab}', R, A, B, \pi')$.*

If $\mathsf{Lab} \preceq \mathsf{Lab}'$ then $\pi, \pi' \models I \Rightarrow I'$.

Note that if π and π' are *empty* assignments, we obtain exactly the theorem on interpolant strength from [7]. Also, note that the theorem permits different variable assignments for the interpolants. Thus, it relates the interpolants generated for different sub-problems (e.g., interpolants considering different sets of paths through a given ARG node). Since both π and π' are assumptions of the formula $I \Rightarrow I'$, the theorem applies to cases common to both sub-problems (i.e., to the shared paths). Both interpolants (I and I') have to be computed using the same A, B partitioning, thus interpolants for different ARG nodes cannot be compared using this theorem; we present a generalization in this direction later.

Thm. 2 is a corollary of Thm. 3, articulated along with its proof below.

Weakened-labels filter. To be able to relate interpolants for different partitionings (i.e., for different labeling functions), we need to introduce a new type of filter, which preserves the literals whose label is weaker in Lab' than in Lab. Let Lab and Lab' be labeling functions. Let $v \in V$ be a vertex, Θ be a clause and $C_1, C_2 \subseteq L$ be sets of labels. The *label change filter* $||$ is defined as follows:

$$\Theta||^{\mathsf{Lab},\mathsf{Lab}'}_{v,C_1 \Rightarrow C_2} = \{l \in \Theta \mid \mathsf{Lab}(v,l) \in C_1 \text{ and } \mathsf{Lab}'(v,l) \in C_2\}$$

For a preserved literal, set C_1 specifies permitted labels for Lab and set C_2 specifies permitted labels for labeling function Lab'. We define *weakened-labels filter* $||\downarrow$ as follows:

$$\Theta||^{\mathsf{Lab},\mathsf{Lab}'}_{v} = \Theta||^{\mathsf{Lab},\mathsf{Lab}'}_{v,\{b,ab,d\} \Rightarrow \{ab,d,a\}}$$

The filter preserves all the literals whose label is weaker in the primed labeling function according to the strength ordering $\preceq$. Note that the weakened-labels filter also preserves some equally strong literals, i.e., those labeled ab or d by both labeling functions. E.g., the filter preserves a literal l if the strongest labels b (i.e., $\mathsf{Lab}(v,l) = b$) is weakened into label a or ab in $\mathsf{Lab}'(v,l)$, while it filters out a literal if both functions assign label a to it. We omit the vertex and labeling functions sub- and superscripts if clear from the context.

In [13], we proved the theorem above using the invariant shown below in the proof sketch. In this article, we choose a different approach; in the next section we show Theorem 3, which is stronger compared to the corresponding one in [13]. Theorem 2 directly follows from Theorem 3 (using the empty set of clauses S).

As in LISs, for a fixed variable assignment there is a lattice of LPAISs ordered according to the strength of labeling functions. The top element of the lattice involves the strongest labeling function, which assigns label b to $A^C_\pi B^C_\pi$-shared and $A^C_\pi B^C_\pi$-clean variables, while the labeling function of the bottom element assigns label a to them. Theorem 2 claims that LPAISs produce interpolants ordered by strength according to the lattice.

5.4 Path interpolation property

Several verification approaches such as [1,16,30] depend on the *path interpolation* property (PI). In [21], the authors show that LISs can be employed to generate path interpolants by providing a sequence of labeling functions that are decreasing in terms of strength. In this subsection, we generalize this property for LPAIS. We study conditions that the labeling functions have to satisfy in order to obtain a sequence of interpolants with the PI property. The following theorem states the main result:

Theorem 3 (PI property) *Let* Lab *and* Lab' *be locality preserving labeling functions such that* $\mathsf{Lab} \preceq \mathsf{Lab}'$ *(see Def. 8),* R *be a refutation of* $A \wedge S \wedge B$*, and* π *and* π' *be PVAs.*

Then for the two interpolants $I = \mathsf{LpaItp}(\mathsf{Lab}, R, A, S \cup B, \pi)$ *and* $I' = \mathsf{LpaItp}(\mathsf{Lab}', R, A \cup S, B, \pi')$*, it holds that* $\pi, \pi' \models I \wedge S \Rightarrow I'$*.*

Before we prove Thm. 3, we introduce auxiliary lemmas; the lemmas are of similar purpose as lemmas used in the proof of the correctness (i.e., the proof of Thm. 1). Lemma 3 about the weakened-labels filters is similar to Lemma 2. The latter one is used to introduce assignment filters.

Lemma 3 (Weakened-labels filters in antecedent vertices)
Let $R \equiv (V, E, cl, piv, s)$ *be a resolution proof and let* Lab *and* Lab' *be labeling functions for proof* R*. Let* $v \in V$ *be an inner vertex of the proof with vertex clause* $cl(v) = \Theta_1, \Theta_2$*. Let vertices* v_1 *and* v_2 *be the antecedents of vertex* v *and their vertex clauses be* $cl(v_1) = p, \Theta_1$ *resp.* $cl(v_2) = \overline{p}, \Theta_2$*. Then the following holds:*

$$\neg p|\downarrow_{v_1} \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow \neg p, \Theta_1|\downarrow_{v_1} \qquad \textit{and}$$
$$\neg \overline{p}|\downarrow_{v_2} \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow \neg \overline{p}, \Theta_2|\downarrow_{v_2}$$

Proof (Lemma 3). First, we show that if a literal $l \in \Theta_1, \Theta_2$ is preserved by the weakened-labels filter in antecedent vertex v_1, then it is also preserved by the weakened-labels filter in vertex v where its label is a result of the join (i.e., $\sqcup$) operation (see D3.2). The sets used by the filter $|\downarrow$, in particular $\{b, ab, d\}$ and $\{ab, d, a\}$, are closed under the join operation (i.e., $\sqcup$); formally, $\forall c \in L$ and $\forall c' \in \{b, ab, d\}$ it holds that $c \sqcup c' \in \{b, ab, d\}$.

If literal l is preserved by the weakened-labels filter in vertex v_1, the first labeling function assigns to literal l a label from the set $\{b, ab, d\}$ (formally, $\mathsf{Lab}(v_1, l) \in \{b, ab, d\}$); for the other labeling function, it holds that $\mathsf{Lab}'(v_1, l) \in \{ab, d, a\}$. Because these sets are closed under the join operation (which is used to compute the labels at vertex v from the labels at vertex v_1), the same holds even in vertex v

(formally $\mathsf{Lab}(v,l) \in \{b, ab, d\}$ and $\mathsf{Lab}'(v,l) \in \{ab, d, a\}$). It means that literal l is also preserved by the weakened-labels filter in vertex v.

It follows that $\Theta_1, \Theta_2{\restriction}_{v_1} \Rightarrow \Theta_1, \Theta_2{\restriction}_v$; the implication can be equivalently rewritten into the contrapositive form:

$$\neg\Theta_1, \Theta_2{\restriction}_v \Rightarrow \neg\Theta_1, \Theta_2{\restriction}_{v_1} \Rightarrow \neg\Theta_1{\restriction}_{v_1}$$

The claim of the lemma directly follows from the above implication:

$$\neg p{\restriction}_{v_1} \wedge \neg\Theta_1, \Theta_2{\restriction}_v \Rightarrow \neg p, \Theta_1{\restriction}_{v_1}$$

Note that the same filter is applied on pivot p; the pivot is either filtered or preserved in both cases. The same reasoning applies to the second formula.

□

Lemma 4 (Introducing assignment filter) *Let π be a partial variable assignment and Θ be a clause not satisfied by the partial assignment, i.e., $\pi \not\models \Theta$.*

Then the following holds: $\pi \models \Theta \Leftrightarrow \Theta[\pi]$.

Proof (Lemma 4). It is possible to split the set of literals Θ into two disjoint sets; set of literals Θ_1 containing the literals over the assigned variables (these literals will be filtered-out by the assignment filter) and set Θ_2 containing the remaining literals over the non-assigned variables. So $\Theta \Leftrightarrow \Theta_1 \vee \Theta_2$.

From the assumption that $\pi \not\models \Theta$, it follows that all the literals over assigned variables evaluate to $\bot$ under the assignment π, thus: $\pi \models \Theta_1 \Leftrightarrow \bot$. From the definition of the assignment filter, it directly follows that $\Theta_2 \equiv \Theta_2[\pi]$ and $\Theta_1[\pi] \equiv \emptyset \Leftrightarrow \bot$. Therefore, the following holds:

$$\pi \models \Theta \Leftrightarrow \Theta_1 \vee \Theta_2 \Leftrightarrow \bot \vee \Theta_2 \Leftrightarrow \Theta_1[\pi] \vee \Theta_2[\pi] \Leftrightarrow \Theta[\pi]$$

□

Proof (Theorem 3—Path interpolation property). By structural induction over refutation R we show that for each vertex $v \in V$ of the refutation, the following invariant holds:

$$\pi, \pi' \models I_v \wedge S \wedge \neg\Theta{\restriction}_v \Rightarrow I'_v$$

where $cl(v) = \Theta$ is the vertex clause and I_v and I'_v are the partial vertex-interpolants for vertex v as generated by LPAIS using labeling functions Lab and Lab', respectively. In the proof, we show that the invariant holds for all possible combinations of the rules that can be used to define partial vertex-interpolants I_v and I'_v.

Bases cases. The base cases correspond to the leaves of the proof where the hypotheses operations are applied. Since there are four possible Hyp rules for each literal and each of the two labeling functions, there are sixteen possible combinations of hypotheses; however, not all of them are possible due to $\mathsf{Lab} \preceq \mathsf{Lab}'$. Below we discuss each of the combinations in more detail.

As to the naming conventions, we call each case either as Hyp or Res, followed by the kind of the rule used to compute the first partial vertex-interpolant I and by the kind of the rule used to compute the second partial vertex-interpolant I'. Note that for the first interpolant, the partitioning is $(A,\ S \cup B)$, while for the second interpolant the partitioning is $(A \cup S, B)$; we use these names of partitions in names of the Hyp rules.

Hyp-A^C_π*-*$(A^C_{\pi'} \cup S^C_{\pi'})$*:* $I_v = \Theta[\pi]\|_{b,v,\mathsf{Lab}}$ and $I'_v = \Theta[\pi']\|_{b,v,\mathsf{Lab}'}$.

First, we show the following:

$$\Theta|_{b,v,\mathsf{Lab}} \wedge \neg\Theta|\!\downarrow_v \Rightarrow \Theta|_{b,v,\mathsf{Lab}'}$$

Let literal l be labeled b by labeling function Lab (i.e., it is preserved by the match filter $|_{b,v,\mathsf{Lab}}$). It can either get a label b by labeling function Lab', thus l is preserved by the match filter $|_{b,v,\mathsf{Lab}'}$ in the consequent of the implication, or it gets a different label, which is necessarily weaker than b. In the latter case, the literal is preserved by the weakened-labels filter $|\!\downarrow_v$, which is negated in the antecedent of the implication above. This means that if clause $\Theta|_{b,v,\mathsf{Lab}}$ is satisfied due to literal l, then either the consequent of the implication is satisfied (the former case) and the implication holds, or the negation of the literal is in the antecedent of the implication (due to weakened-labels filter), so the antecedent of the implication is not satisfied (and the whole implication holds).

We can add the assignments as assumptions. Then the following holds:

$$\pi, \pi' \models \Theta|_{b,v,\mathsf{Lab}} \wedge \neg\Theta|\!\downarrow_v \Rightarrow \Theta|_{b,v,\mathsf{Lab}'}$$

Clause Θ is neither satisfied by π nor by π', so Lemma 4 can be used to remove the falsified literals. Then the following holds:

$$\pi, \pi' \models \quad \Theta[\pi]\|_{b,v,\mathsf{Lab}} \wedge \neg\Theta|\!\downarrow_v \Rightarrow \Theta[\pi']\|_{b,v,\mathsf{Lab}'}$$

$$\pi, \pi' \models \quad I_v \quad \wedge S \wedge \neg\Theta|\!\downarrow_v \Leftrightarrow$$
$$\Theta[\pi]\|_{b,v,\mathsf{Lab}} \wedge S \wedge \neg\Theta|\!\downarrow_v \Rightarrow \Theta[\pi']\|_{b,v,\mathsf{Lab}'} \Leftrightarrow I'_v$$

Hyp-A^C_π*-*$(A_{\pi'} \cup S_{\pi'})$*:* $I_v = \Theta[\pi]\|_{b,v,\mathsf{Lab}}$ and $I'_v = \top$.

Note that in contrast to the previous case, vertex clause Θ is satisfied under assignment π'. In this case, the invariant holds trivially, since anything implies $\top$.

Hyp-A^C_π*-*$B^C_{\pi'}$*:* $I_v = \Theta[\pi]\|_{b,v,\mathsf{Lab}}$ and $I'_v = \Theta[\pi']\|_{a,v,\mathsf{Lab}'}$.

This combination is impossible due to our partitionings; it would require clause Θ to move from the A-part of the first partitioning into the B-part of the second partitioning. However, the partitionings permit only moves of clauses from the B-part of the first partitioning into the A-part of the second partitioning; the moved clauses are those forming S.

Hyp-A^C_π*-*$B_{\pi'}$*:* $I_v = \Theta[\pi]\|_{b,v,\mathsf{Lab}}$ and $I'_v = \top$. The same reasoning as above applies; such combination is impossible due to our partitionings. Note that the reasoning above is independent of the assignment.

Hyp-$(B^C_\pi \cup S^C_\pi)$-$(A^C_{\pi'} \cup S^C_{\pi'})$: $I_v = \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}}$ and $I'_v = \Theta[\pi']\!\!]_{b,v,\mathsf{Lab}'}$. In this case, clause Θ is moved from the B-part of the first partitioning into the A-part of the second partitioning; it means the clause belongs to the set of clauses S ($\Theta \in S$). First, we show that in this case, the following holds:

$$\Theta \Leftrightarrow \Theta|_{a,v,\mathsf{Lab}} \vee \Theta|_{b,v,\mathsf{Lab}'} \vee \Theta|\!\downarrow_v$$

The direction from right to left (i.e., the implication $\Leftarrow$) is trivial, since filters only remove literals. So if the right-hand side of the equivalence holds, the unfiltered clause Θ must also hold. The direction from left to right (i.e., the implication $\Rightarrow$) is shown below. We consider all the combinations of labels the literal $l \in \Theta$ can get by labeling functions Lab and Lab'.

- If $\mathsf{Lab}(v,l) = a$ then the match filter $|_{a,v,\mathsf{Lab}}$ preserves the literal l.
- If $\mathsf{Lab}(v,l) \in \{ab, d\}$ and $\mathsf{Lab}'(v,l) \neq b$ then the weakened-label filter $|\!\downarrow_v$ preserves the literal l.
- If $\mathsf{Lab}(v,l) \in \{ab, d\}$ and $\mathsf{Lab}'(v,l) = b$ then the assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.
- If $\mathsf{Lab}(v,l) = b$ and $\mathsf{Lab}'(v,l) = b$ then the match filter $|_{b,v,\mathsf{Lab}'}$ preserves the literal l.
- If $\mathsf{Lab}(v,l) = b$ and $\mathsf{Lab}'(v,l) \neq b$ then the weakened-label filter $|\!\downarrow_v$ preserves the literal l.

The clause Θ is satisfied neither by π nor by π' so Lemma 4 can be used to remove the falsified literals. Then the following holds:

$$\pi, \pi' \models \Theta \Leftrightarrow \Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \vee \Theta[\pi']\!\!]_{b,v,\mathsf{Lab}'} \vee \Theta|\!\downarrow_v$$

The invariant is shown by the following:

$$\begin{aligned}
\pi, \pi' \models \quad & I_v \quad \wedge && S && \wedge \neg\Theta|\!\downarrow_v \equiv && \\
\equiv\ & \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \wedge && S && \wedge \neg\Theta|\!\downarrow_v \Rightarrow && (1)\\
\Rightarrow\ & \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \wedge && \Theta && \wedge \neg\Theta|\!\downarrow_v \Leftrightarrow && (2)\\
\Leftrightarrow\ & \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \wedge && (\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \vee \Theta[\pi']\!\!]_{b,v,\mathsf{Lab}'} && \vee\Theta|\!\downarrow_v) \wedge \neg\Theta|\!\downarrow_v \Leftrightarrow && (3)\\
\Leftrightarrow\ & \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}} \wedge && \Theta[\pi']\!\!]_{b,v,\mathsf{Lab}'} && \wedge \neg\Theta|\!\downarrow_v \Rightarrow && (4)\\
& && \Rightarrow \Theta[\pi']\!\!]_{b,v,\mathsf{Lab}'} && \equiv I'_v &&
\end{aligned}$$

The implication (1) follows from the fact that $\Theta \in S$ and S is a conjunction (or equivalently a set) of clauses, so $S \Rightarrow \Theta$. The equivalence (2) is shown above. The equivalence (3) is a logical consequence. The following pattern is used twice: $\neg A \wedge (A \vee B) \Leftrightarrow (\neg A \wedge A) \vee (\neg A \wedge B) \Leftrightarrow \neg A \wedge B$, where we use $A \equiv \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}}$ resp. $A \equiv \neg\Theta|\!\downarrow_v$. The implication (4) is a trivial logical consequence.

Hyp-$(B^C_\pi \cup S^C_\pi)$-$(A_{\pi'} \cup S_{\pi'})$: $I_v = \neg\Theta[\pi]\!\!]_{a,v,\mathsf{Lab}}$ and $I'_v = \top$.

As in all the other cases where the vertex clause is satisfied by the second assignment π', the invariant holds trivially, because anything implies $I'_v \equiv \top$.

Hyp-$(B^C_\pi \cup S^C_\pi)$*-*$B^C_{\pi'}$*:* $I_v = \neg\Theta[\pi]\!]_{a,v,\mathsf{Lab}}$ and $I'_v = \neg\Theta[\pi']\!]_{a,v,\mathsf{Lab}'}$.

This case is similar to the Hyp-A^C_π-$(A^C_{\pi'} \cup S^C_{\pi'})$ case. First, we show that:

$$\neg\Theta|_{a,v,\mathsf{Lab}} \wedge \neg\Theta|\!\downarrow_v \Rightarrow \neg\Theta|_{a,v,\mathsf{Lab}'}$$

Let literal l be labeled a by labeling Lab' (so it is preserved by the match filter $|_{a,v,\mathsf{Lab}'}$). The literal is either labeled a by labeling Lab or not (in which case its label can be b or ab). In the former case, the literal is preserved by the match filter $|_{a,v,\mathsf{Lab}}$. In the latter case, the literal is preserved by the weakened-labels filter $|\!|\downarrow_v$. To sum it up, all the literals in the consequent of the implication occur even in the antecedent of the implication; this shows that the implication holds.

Clause Θ is neither satisfied by π nor by π', so Lemma 4 can be used to remove the falsified literals. Then the following holds:

$$\pi, \pi' \models \neg\Theta[\pi]\!]_{a,v,\mathsf{Lab}} \wedge \neg\Theta|\!\downarrow_v \Rightarrow \neg\Theta[\pi']\!]_{a,v,\mathsf{Lab}'}$$

The implication above is even stronger than the invariant; it does not require S to be a part of the antecedent of the implication.

Hyp-$(B^C_\pi \cup S^C_\pi)$*-*$B_{\pi'}$*:* $I_v = \neg\Theta[\pi]\!]_{a,v,\mathsf{Lab}}$ and $I'_v = \top$.

As in all the other cases where the vertex clause is satisfied by the second assignment π' the invariant holds trivially, because anything implies $I'_v \equiv \top$.

Hyp-A_π*-*$(A^C_{\pi'} \cup S^C_{\pi'})$*:* $I_v = \top$ and $I'_v = \Theta[\pi']\!]_{b,v,\mathsf{Lab}'}$.

Vertex clause Θ is satisfied under π, thus there exists literal $l \in \Theta$ such that $\mathsf{Lab}(v, l) = d$; literal l makes the clause satisfied under π. From the assumption (of the theorem that) $\mathsf{Lab} \preceq \mathsf{Lab}'$, it follows that $\mathsf{Lab}'(v, l) \neq b$; thus literal l is preserved by weakened-labels filter $|\!|\downarrow_v$. It means that $\pi, \pi' \models \neg\Theta|\!|\downarrow_v \Leftrightarrow \bot$, so the antecedent of the invariant implication is falsified by the assumed assignments and the whole invariant implication holds.

Exactly the same reasoning applies to all the remaining hypotheses where the assignment π satisfies the vertex clause; in particular, to the Hyp-A_π-$(A_{\pi'} \cup S_{\pi'})$, Hyp-A_π-$B^C_{\pi'}$, Hyp-A_π-$B_{\pi'}$, Hyp-$(B_\pi \cup S_\pi)$-$(A^C_{\pi'} \cup S^C_{\pi'})$, Hyp-$(B_\pi \cup S_\pi)$-$(A_{\pi'} \cup S_{\pi'})$, Hyp-$(B_\pi \cup S_\pi)$-$B^C_{\pi'}$, and Hyp-$(B_\pi \cup S_\pi)$-$B_{\pi'}$.

Induction hypothesis. Now, we will focus on the inductive step. Let v be an inner vertex of the proof and let variable p be the pivot of the refutation at vertex v (i.e., $p = piv(v)$). Let vertex v_1 be the antecedent of v with the vertex-clause containing the pivot positively (i.e., $cl(v_1) = p, \Theta_1$) and let vertex v_2 be the antecedent of v having negated pivot in its vertex-clause (i.e., $cl(v_2) = \overline{p}, \Theta_2$). From the induction hypothesis, we know that for the antecedent vertices the following invariants hold:

$$\begin{aligned} &\pi, \pi' \models I_{v_1} \wedge S \wedge \neg\Theta|\!\downarrow_v \Rightarrow I'_{v_1} \\ &\pi, \pi' \models I_{v_2} \wedge S \wedge \neg\Theta|\!\downarrow_v \Rightarrow I'_{v_2} \end{aligned} \tag{PIH}$$

For each possible combination of the resolutions we establish the induction invariant for the vertex v. Theoretically, there are 16 possible combinations of resolutions, however, not all of them are possible due to the assumptions of the theorem; below we discuss each of the combinations in more detail.

Res-a-a′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that $I_v \equiv I_{v_1} \vee I_{v_2}$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

The label of pivot p in both antecedent vertices v_1 resp. v_2 must be a (in both labeling functions), so it is not preserved by the weakened-labels filters $||\lfloor_{v_1}$ and $||\lfloor_{v_2}$; thus the following holds $\neg p||\lfloor_{v_1} \Leftrightarrow \top$.

It holds that:

$$\begin{aligned}\pi, \pi' \models I_{v_1} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Leftrightarrow \neg p||\lfloor_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \overset{L3}{\Rightarrow} \\ \Rightarrow I_{v_1} \wedge S \wedge \neg p, \Theta_1||\lfloor_{v_1} \overset{PIH}{\Rightarrow} I'_{v_1} \\ \pi, \pi' \models I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Leftrightarrow \neg \overline{p}||\lfloor_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \overset{L3}{\Rightarrow} \\ \Rightarrow I_{v_2} \wedge S \wedge \neg \overline{p}, \Theta_2||\lfloor_{v_2} \overset{PIH}{\Rightarrow} I'_{v_2}\end{aligned}$$

The invariant for the vertex v follows directly from the implications above:

$$\pi, \pi' \models ((I_{v_1} \vee I_{v_2})) \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Rightarrow (I'_{v_1} \vee I'_{v_2})$$

Res-a-ab′, Res-a-d′ and Res-a-b′: All these cases violate the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. Pivot p gets a stronger label than a by Lab' in at least one of the antecedent vertices (i.e., v_1 and v_2); otherwise, this will become the previous Res-*a*-a' case. Variable p at that vertex is the witness that the assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.

Res-ab-ab′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = ab$. It means that $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$ and $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

Note that in this case, the proof is independent of the labels of the pivot variable. Also note that the proof works regardless of whether p is assigned by assignment π (resp. π') or not. So it can be safely used to show other cases, such as Res-*ab*-a', as well.

We have to show the following:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$

First, we introduce two auxiliary implications. The following holds:

$$\begin{aligned}\pi, \pi' \models I_{v_1} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Rightarrow p \vee (\overline{p} \wedge I_{v_1} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v) \Rightarrow \\ \Rightarrow p \vee (\neg p||\lfloor_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v) \overset{L3}{\Rightarrow} \\ \Rightarrow p \vee (I_{v_1} \wedge S \wedge \neg p, \Theta_1||\lfloor_{v_1}) \overset{PIH}{\Rightarrow} p \vee I'_{v_1} \\ \pi, \pi' \models I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v \Rightarrow \overline{p} \vee (p \wedge I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v) \Rightarrow \\ \Rightarrow \overline{p} \vee (\neg \overline{p}||\lfloor_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2||\lfloor_v) \overset{L3}{\Rightarrow} \\ \Rightarrow \overline{p} \vee (I_{v_2} \wedge S \wedge \neg \overline{p}, \Theta_1||\lfloor_{v_2}) \overset{PIH}{\Rightarrow} \overline{p} \vee I'_{v_2}\end{aligned}$$

The first implication stems from the fact that $p \vee \overline{p} \Leftrightarrow \top$. The second implication holds because $\overline{p} \Rightarrow \neg p||\lfloor_{v_1}$; either the literal $\overline{p}$ is preserved by the filter $||\lfloor_{v_1}$ and then it holds that $\overline{p} \Leftrightarrow \neg p||\lfloor_{v_1}$, or the literal $\overline{p}$ is not preserved by the weakened-labels filter and then it holds that $\neg p||\lfloor_{v_1} \Leftrightarrow \top$.

The proof can be split into two cases. It holds that:

$$(p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \Leftrightarrow (p \wedge I_{v_2}) \quad (1)$$
$$\vee$$
$$(\overline{p} \wedge I_{v_1}) \quad (2)$$

We show that each of the two cases (1) and (2) leads to $(p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

$$\pi, \pi' \models (\overline{p} \wedge I_{v_1}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \quad \Rightarrow \overline{p} \wedge (p \vee I'_{v_1}) \quad \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$
$$\pi, \pi' \models (p \wedge I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \quad \Rightarrow p \wedge (\overline{p} \vee I'_{v_2}) \quad \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$

The first implication comes from the auxiliary implications above. The second implication is a simple logical consequence.

We have shown that:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$$

Res-ab-a′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

It holds that:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow$$
$$\Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Rightarrow I'_{v_1} \vee I'_{v_2}$$

The first implication comes from Res-*ab*-*ab′*. The second one is a trivial logical consequence.

Res-ab-d′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = d$. So, partial vertex-interpolant I_v is defined as follows: $I_v \equiv (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$. Assume that $\mathsf{Lab}'(v_1, p) = d$ thus $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}'(v_2, \overline{p}) = d$.

It holds that:

$$\pi, \pi' \models (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Leftrightarrow I'_{v_2}$$

The implication comes from Res-*ab*-*ab′*. It holds that $\pi' \models p$ (the locality constraint D6.1 and $\mathsf{Lab}'(v_1, p) = d$); the equivalence is a trivial logical consequence of the fact that variable p is assigned $\top$ by π'.

Res-ab-b′: This case violates the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. The pivot variable p gets the strongest label b by Lab' in both antecedent vertices (i.e., v_1 and v_2). However, the first labeling function Lab has to assign a weaker label a resp. ab to the pivot variable in at least one of the antecedent vertices; otherwise this case will become Res-*b*-*b′*. Variable p at that vertex is the witness that the assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.

Res-b-a′: $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = b$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = a$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv I'_{v_1} \vee I'_{v_2}$.

It holds that:

$$\begin{aligned}\pi,\pi' \models\ & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow \\ \Rightarrow\ & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow \\ & \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Rightarrow I'_{v_1} \vee I'_{v_2}\end{aligned}$$

The first and third implications are simple logical consequences. The second implication comes from Res-*ab*-*ab′*.

Res-b-ab′: $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = b$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = ab$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$.

It holds that:

$$\begin{aligned}\pi,\pi' \models\ & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow \\ \Rightarrow\ & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})\end{aligned}$$

The first implication is simple logical consequence. The second implication comes from Res-*ab*-*ab′*.

Res-b-d′: $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = b$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = d$. So, partial vertex-interpolant I_v is defined as $I_v \equiv I_{v_1} \wedge I_{v_2}$. Assume that $\mathsf{Lab}'(v_1,p) = d$, thus $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}'(v_2,\overline{p}) = d$. We will use Res-*ab*-*ab′* in the same way as in the above cases.

It holds that:

$$\begin{aligned}\pi,\pi' \models\ & (I_{v_1} \wedge I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow \\ \Rightarrow\ & (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Rightarrow \\ & \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \Leftrightarrow I'_{v_2}\end{aligned}$$

The first implication is a simple logical consequence. The second implication comes from Res-*ab*-*ab′*. It holds that $\pi' \models p$ (the locality constraint D6.1 and $\mathsf{Lab}'(v_1,p) = d$); the second equivalence is a trivial logical consequence of the fact that variable p is assigned $\top$ by π'.

Res-b-b′: $\mathsf{Lab}(v_1,p) \sqcup \mathsf{Lab}(v_2,\overline{p}) = b$ and $\mathsf{Lab}'(v_1,p) \sqcup \mathsf{Lab}'(v_2,\overline{p}) = b$. It means that $I_v \equiv I_{v_1} \wedge I_{v_2}$ and $I'_v \equiv I'_{v_1} \wedge I'_{v_2}$.

The label of pivot p in both antecedent vertices v_1 and v_2 must be b (in both labeling functions), so it is not preserved by the weakened-labels filters $|\downarrow_{v_1}$ and $|\downarrow_{v_2}$; thus, it holds that $\neg p|\downarrow_{v_1} \Leftrightarrow \top$.

The same auxiliary implications as in the previous Res-*a*-*a′* case hold:

$$\begin{aligned}\pi,\pi' \models I_{v_1} \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Leftrightarrow \neg p|\downarrow_{v_1} \wedge I_{v_1} \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \overset{L3}{\Rightarrow} \\ \Rightarrow I_{v_1} \wedge S \wedge \neg p,\Theta_1|\downarrow_{v_1} \overset{PIH}{\Rightarrow} I'_{v_1} \\ \pi,\pi' \models I_{v_2} \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \Leftrightarrow \neg\overline{p}|\downarrow_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\Theta_1,\Theta_2|\downarrow_v \overset{L3}{\Rightarrow} \\ \Rightarrow I_{v_2} \wedge S \wedge \neg\overline{p},\Theta_2|\downarrow_{v_2} \overset{PIH}{\Rightarrow} I'_{v_2}\end{aligned}$$

The first equivalence is shown above. The following implication is application of Lemma 3, while the last one is the induction hypothesis.

The invariant for the vertex v follows directly from these implications:

$$\pi, \pi' \models (I_{v_1} \wedge I_{v_2}) \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Rightarrow (I'_{v_1} \wedge I'_{v_2})$$

Res-d-a′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = a$. It means that partial vertex-interpolant I'_v is defined as follows: $I'_v \equiv I'_{v_1} \vee I'_{v_2}$. Assume that $\mathsf{Lab}(v_1, p) = d$, thus $I_v \equiv I_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d$.

It holds that $\pi \models p$ (the locality constraint D6.1 and $\mathsf{Lab}(v_1, p) = d$); thus, the following equivalence holds: $\pi \models I_{v_2} \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$. So, the invariant for vertex v can be established using the Res-*ab*-*ab* resolution:

$$\begin{aligned} \pi, \pi' \models \quad & I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Leftrightarrow \\ \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v & \Rightarrow \\ \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) & \Rightarrow I'_{v_1} \vee I'_{v_2} \end{aligned}$$

Res-d-ab′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = ab$. It means that partial vertex-interpolant I'_v is defined as follows: $I'_v \equiv (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2})$. Assume that $\mathsf{Lab}(v_1, p) = d$, thus $I_v \equiv I_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d$.

It holds that $\pi \models p$ (the locality constraint D6.1 and $\mathsf{Lab}(v_1, p) = d$); thus the following equivalence holds: $\pi \models I_{v_2} \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2})$.

The invariant for vertex v can be established using the Res-*ab*-*ab* resolution:

$$\begin{aligned} \pi, \pi' \models \quad & I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Leftrightarrow \\ \Leftrightarrow (p \vee I_{v_1}) \wedge (\overline{p} \vee I_{v_2}) \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v & \Rightarrow (p \vee I'_{v_1}) \wedge (\overline{p} \vee I'_{v_2}) \end{aligned}$$

Res-d-d′: $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = d$ and $\mathsf{Lab}'(v_1, p) \sqcup \mathsf{Lab}'(v_2, \overline{p}) = d$.

This rule splits into two different sub-cases; the pivot variable gets assigned either the same value by π and π' or one assignment assigns $\top$, while the other assignment assigns $\bot$ to p.

In the latter case, the invariant for vertex v holds trivially. The assumptions (i.e., assignments) contradict, so any formula, e.g., the invariant, holds. Let us focus on the former case. Assume that $\mathsf{Lab}(v_1, p) = d$, thus it holds that $\mathsf{Lab}'(v_1, p) = d$ and $I_v \equiv I_{v_2}$, $I'_v \equiv I'_{v_2}$. The situation is symmetric if $\mathsf{Lab}(v_2, \overline{p}) = d$.

The invariant for vertex v can be established from the invariant of vertex v_2:

$$\begin{aligned} \pi, \pi' \models I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v \Leftrightarrow p \wedge I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v & \Leftrightarrow \\ \Rightarrow \neg\overline{p}|\downarrow_{v_2} \wedge I_{v_2} \wedge S \wedge \neg\Theta_1, \Theta_2|\downarrow_v & \overset{L3}{\Rightarrow} \\ \Rightarrow I_{v_2} \wedge S \wedge \neg\overline{p}, \Theta_1|\downarrow_{v_2} & \overset{PIH}{\Rightarrow} I'_{v_2} \end{aligned}$$

It holds that $\pi \models p$ (the locality constraint D6.1 and $\mathsf{Lab}(v_1, p) = d$); this shows the first equivalence. It holds that $\pi \models \neg\overline{p}|\downarrow_{v_2}$; either the filter preserves the literal $\overline{p}$, which means that due to the assignment $\neg\overline{p} \equiv p$ evaluates to $\top$ or the filter removes the literal and the negated empty clause is equivalent to $\top$ without any assumptions. The above reasoning shows the second equivalence.

Note that alternatively, this case can be shown via Res-*ab*-*ab* resolution as well.

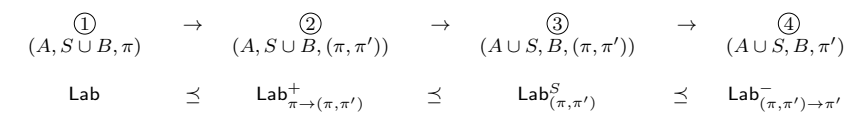

Fig. 14 Idea of PI property proof in [13].

Res-d-b′: This case violates the assumption of the theorem that $\mathsf{Lab} \preceq \mathsf{Lab}'$. Pivot p gets the strongest label b by Lab' in both antecedent vertices (i.e., v_1 and v_2). However, the first labeling function Lab has to assign a weaker label d to the pivot variable in one of the antecedent vertices. Variable p at that vertex is the witness that assumption $\mathsf{Lab} \preceq \mathsf{Lab}'$ is violated.

□

Existence of labeling functions. Theorem 3 requires two labeling functions related by their strength ($\mathsf{Lab} \preceq \mathsf{Lab}'$). However, such a pair of labeling functions may not exist. In the following, we discuss this in more detail. To simplify the situation, first assume that no clauses are moved between the A and B parts, i.e., $S \equiv \emptyset$.

Assume that A^C_π-local variable x is assigned True and that x satisfies a clause in A (formally expressed $\pi' \equiv \pi \wedge x$ and that $A^C_\pi \supset A_{\overline{\pi}'}$).

In such a case, labeling function Lab has to assign label a to all occurrences of literal x, since x is A^C_π-local (under assignment π). However, labeling function Lab' needs to assign label d to these literals because of π'. These literals show that $\mathsf{Lab} \not\preceq \mathsf{Lab}'$. A symmetric situation occurs if an unassigned variable becomes $B^C_{\pi'}$-local.

In [13], the PI property is shown in a different way, however, the proof is constructive. In other words, given a labeling function Lab, (the strongest locality-preserving) labeling function Lab' for $(A \cup S, B)$ and assignment π' are created (denoted as $\mathsf{Lab}^-_{(\pi,\pi')\to\pi'}$ in the proof).

In [13] the strongest possible locality preserving labeling function ($\mathsf{Lab}^-_{(\pi,\pi')\to\pi'}$) is constructed in 3 steps, which are highlighted in Fig. 14.

In the first step (① → ②), new variables from π' are assigned, thus the assignment being considered in ② is $\pi \wedge \pi'$. An *extended-assignment labeling function* ($\mathsf{Lab}^+_{\pi\to(\pi,\pi')}$)—the strongest locality-preserving labeling function if new variables get assigned—is created from Lab. In the second step (② → ③), the clauses of S are moved under the fixed variable assignment ($\pi \wedge \pi'$). The *strongest successor labeling function* ($\mathsf{Lab}^S_{(\pi,\pi')}$) is created from $\mathsf{Lab}^+_{\pi\to(\pi,\pi')}$. In the the last step (③ → ④), the assignment π is removed. The *restricted-assignment labeling function* ($\mathsf{Lab}^-_{(\pi,\pi')\to\pi'}$)—the strongest locality-preserving labeling function if variables get unassigned—is computed from $\mathsf{Lab}^S_{(\pi,\pi')}$.

We have shown that under some additional assumptions (which are easy to satisfy in our motivation example), the labeling functions are connected by strength as shown in the third line of Fig. 14. Hereby, we have constructed (locality preserving) function $\mathsf{Lab}^-_{(\pi,\pi')\to\pi'}$ that is weaker than Lab as required. Moreover, we have shown that $\mathsf{Lab}^-_{(\pi,\pi')\to\pi'}$ is the strongest function satisfying the requirements.

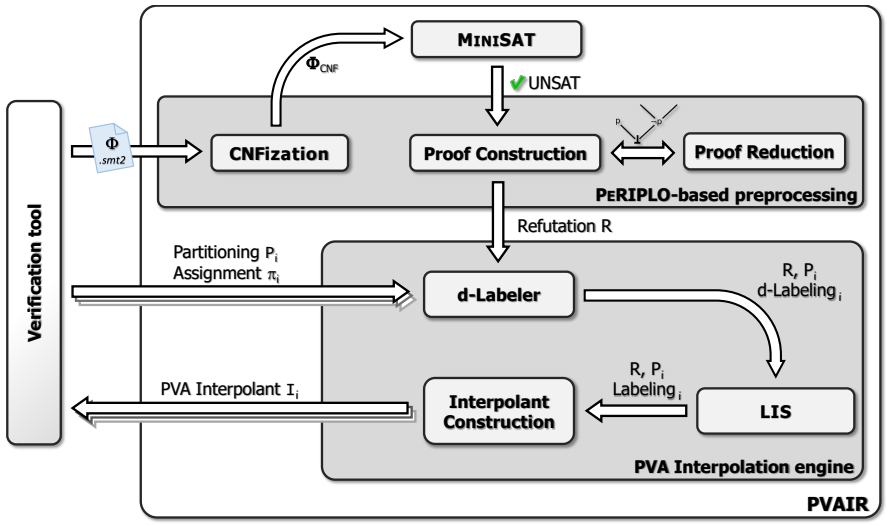

Fig. 15 PVAIR architecture.

DAG interpolants. Let us look back at our motivation example. For each ARG node (i), the PVA π_i blocking all paths not going via the node i can be easily constructed; see assignment π_3 in the motivation example. Such an assignment blocks each node j (i.e., assigns False to the corresponding structure-encoding variables n_j) that both cannot reach node i ($j \not\rightsquigarrow i$) and cannot be reached from node i ($i \not\rightsquigarrow j$); formally: $\pi_i \equiv n_i \wedge \bigwedge_j (\neg n_j \mid j \not\rightsquigarrow i \text{ and } i \not\rightsquigarrow j)$.

Let each ARG node (i) be annotated by the blocking PVA as shown above and Lab_i be (locality preserving) labeling functions such that their strength is decreasing along each ARG edge, i.e., for each ARG edge $i \to j$, it holds that $\mathsf{Lab}_i \preceq \mathsf{Lab}_j$.

Assume that node PVA interpolants are computed by LPAIS using the assignments and labeling functions above; formally, $I_i \equiv \mathsf{Lpaltp}(\mathsf{Lab}_i, R, A_i, B_i, \pi_i)$ where R is the Cond for given ARG and A_i is the part of R corresponding to the antecedent nodes of i. Then, by Theorem 3, it directly follows that node PVA interpolants form so called DAG interpolants.

The variable assignment is used to remove out-of-scope variables, while the path interpolation property guarantees a correct strength relation among node interpolants.

6 Evaluation

We implemented LPAIS in Partial Variable Assignment InterpolatoR (PVAIR). PVAIR is built on top of the open-source tool PeRIPLO [19], which provides refutations and is able to optimize the refutations for interpolation through transformations. PeRIPLO has been used in various verification projects, including function summarization in eVolCheck [9] and FunFrog [25], both as an interpolation engine and as a SAT solver.

The PVAIR architecture is shown in Fig. 15. It takes a propositional formula Φ, its (A, B)-partitioning, and a partial variable assignment π as input and produces PVA interpolant if the input formula is unsatisfiable. The input can be provided either in a file in the SMT-LIB 2.0 [4] format or via a C++ API.

The workflow of the PVAIR tool is as follows. First, the input formula is passed to the PeRIPLO-based preprocessing module. Since the formula can be in arbitrary form, it is transformed into CNF (the top box in Fig. 15) using an efficient, structure-sharing version of the Tseitin encoding [29]. Its satisfiability is then determined using the MiniSAT 2.2.0 solver [8].

In the case of an unsatisfiable input, an initial refutation is extracted from the solver in the compact MiniSAT internal proof format. The format is then transformed into a resolution DAG to allow more efficient handling of the refutation (Proof Construction). In particular, using the resolution DAG form, the refutation can be compressed using well-known proof reduction techniques such as structural hashing or pivot recycling [20,18] available in PeRIPLO (Proof Reduction). The proof reduction techniques can be enabled/disabled via a configuration file or API.

Once the refutation R is computed, it is passed together with the partitionings and variable assignments to the interpolation engine (the bottom box in Fig. 15). From this point on, any number of partial variable assignments π_i and partitionings P_i (into $A_i \wedge B_i$) can be given as input to the tool and used to construct the corresponding interpolants I_i. Note that in any case only one SAT-solver call will be made during the entire execution. Then the refutation is labeled; the labels are assigned to literals in the refutation based on the partitioning and the assignment and selected LIS-based interpolation algorithm (which can be chosen in the configuration file or via API). When the labeling is complete, it is used together with the partitioning and refutation R to compute interpolants (Interpolant Construction).

The construction starts by computing partial vertex-interpolants (according to the upper part of Tab. 1) for the leaf nodes of the refutation. The computation then proceeds from the leaves to the root node. During the interpolant construction, partial interpolants are optimized using Boolean constant propagation and structural sharing (hashing). The final interpolant is computed in the root node.

In symbolic model checking Craig interpolants are used as a tool to compute sets of program states with required properties; their usage varies significantly among verification techniques. Thus we decide to choose unsatisfiable benchmarks from SAT Competition [23]. They provide us with large and heterogeneous set of benchmarks. We also employed the PVAIR tool in software verification process; we applied it on computational problems generated by the eVolCheck tool during verification procedure. To demonstrate the tool performance, we measured the size of produced interpolants and its effect on the total verification time.

6.1 SAT competition

We used 47 unsatisfiable benchmarks from the SAT Competition 2011 from all categories—12 from the *Application* (APP), 11 from the *Crafted* (CRF), and 24 from the *Random* (RND) sets. Since the benchmarks are not partitioned, we generated six partitionings for each benchmark; we simulated the typical way the path interpolants are computed, i.e., we randomly choose a number n, and the first n clauses of the benchmark will belong to the A part, whereas the remaining clauses

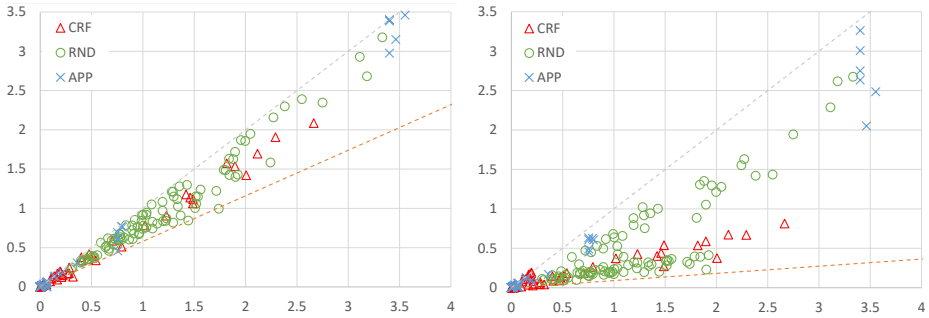

Fig. 16 Comparison of interpolant sizes computed without variable assignment [x] and with one variable assigned [y] (left) and five variables assigned (right).

PVAIs	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	92.8 %	83.0 %	78.1 %	83.7 %
5 vars	76.2 %	45.2 %	31.5 %	47.6 %
20 vars	48.3 %	10.1 %	4.8 %	15.0 %

Itp. from sub-prob.	APP	RND	CRF	All
No Assignment	344 298.7	1 308 750.1	489 469.1	776 573.9
1 var	69.5 %	55.0 %	65.5 %	58.8 %
5 vars	24.4 %	5.7 %	9.7 %	9.1 %
20 vars	0.12 %	0.01%	0.39%	0.09%

Table 2 Average interpolant sizes by category and number of assigned variables.

belong to the B part. We ensure that no partition is empty. No assignment is given by authors of the benchmarks, thus for each partitioning, we generate five random variable assignments consisting of a single, five, resp. twenty assigned variables. Assignments of various sizes indicate how the reduction scales w.r.t. the number of assigned variables.

For comparison, we use McMillan's interpolants—a widely used approach. Experiments were run on a Linux server with Intel Xeon X5687 CPU using the timeout of 60 minutes and the memory limit of 20GB using the GNU Parallel environment [27]. The proof reduction techniques were disabled; we used the default PeRIPLO settings.

Fig. 16 compares the sizes of the computed interpolants. Each point in the graph corresponds to a single partitioning of a benchmark; the x-axis represents the interpolant size if no assignment is provided (Craig interpolant), while the y-axis represents the size of the PVA interpolant with a single (resp. five and twenty) assigned variable(s). For presentation clarity, the y-axis is the average size of all five random assignments generated for a given partitioning. The values on axes represent millions of nodes if an interpolant is represented as DAG (counting literals and Boolean operators). All graphs show expected reduction in the size for PVA interpolants as well as substantially larger reduction in case of five resp. twenty assigned variables. In all graphs, the same partition of the same benchmark shares the same x-value, thus it is possible, especially for the larger ones, to compare their reductions.

Tab. 2 summarizes the results shown in the graphs, reporting precise numbers. The first table compares the sizes of PVA interpolants to Craig interpolants. The *No assignment* row shows the average size of Craig interpolants for a given benchmark type. The remaining rows show the relative sizes of focused interpolants w.r.t. the *No assignment* row. The application benchmarks exhibit a smaller reduction compared to the other types, and even for twenty assigned variables, the interpolants are half in the size of the Craig interpolants.

Time and memory demands are crucial properties of each interpolation tool. The reduction in overall running time and required memory roughly correspond to the reduction of interpolant sizes; e.g., on average PVAIR is 11% faster and requires 9% less memory if a single variable is assigned. The time and memory savings occur during the interpolant computation phase due to smaller interpolants being handled.

A PVA interpolant can be seen as a Craig interpolant for the sub-problem corresponding to the related partial assignment. For each pair of partitioning and assignment, we created the sub-problem instance and used PVAIR to compute the Craig interpolant for it. Sub-problems are simpler compared to the benchmark from which they were generated; the satisfied clauses and falsified literals are removed. As a result also the interpolants for sub-problems are typically smaller compared to Craig interpolants of the benchmark. However, the interpolant for each sub-problem is computed from a different refutation; in contrast to focused interpolants, which, for a particular benchmark, are all computed from the same refutation. This means that the sequence of interpolants for sub-problems may not have the path interpolation property [30].

Conclusions of the experimentation. The *No assignment* reflects the state-of-the-art approaches, where Craig interpolants are used directly. Focused interpolants show how the size of the interpolants can be reduced if the model checker (i.e., a tool generating the input) provides an assignment together with a partitioning. While in some cases interpolants for a sub-problem can be seen as an alternative to focused interpolants because of their similar meaning, these interpolants lack the properties of the focused ones.

6.2 Applying PVAIR for Checking Software Upgrades

The usefulness of PVAIR is motivated by the tremendous role of interpolation in symbolic model checking. One of the possible applications of PVAIR is checking software upgrades [24]. The eVolCheck tool relies on interpolation to generate so called function summaries for a given program S and an assertion a, and then it uses them to accelerate verification of a modified version U of S against the same assertion a.

Below, we describe the technique in more detail. eVolCheck follows the Bounded Model Checking paradigm, in which loops are unrolled a fixed number of times, and the verification of S against a is reduced to the satisfiability of formula $S \wedge \neg a$. If this formula is unsatisfiable, a set of interpolants can be extracted from the resolution refutation. We refer to them as to *function summaries* [26]. By construction, a function summary is an over-approximation of behavior of

the corresponding function, and it preserves all the necessary information about reachable states in S which is relevant to the proof that a holds in S.

Given a modified version U of S, eVolCheck *validates* the existing function summaries for the new behavior of the corresponding functions in U. In that context, programs S and U must have a non-empty set of common function calls. eVolCheck traverses this set starting from the deepest level of the (unwound during preprocessing) function call tree and checks whether each original function summary still over-approximates the new behavior of the corresponding function. If there is a function call, the original summary of which does not over-approximate the new behavior, eVolCheck propagates the check to the caller function. If there is no function to propagate then U is buggy. If at some depth of the unwound call tree all the function summaries are proven to be valid, then U is safe, and eVolCheck reconstructs the summaries for the modified function calls.

Applying PVAIR *to* eVolCheck. Consider the case when U is obtained from S by *removing some behaviors* (deleting some lines of code), i.e., U is a refinement of S. Then by construction, the original summaries of S are still valid over-approximations of the new behavior of the corresponding functions in U. However, they might be unnecessarily general and consume excessive memory. While the use of the original summaries does not break soundness of the further upgrade checking, it is practical to refresh, and thereby potentially shrink, the summaries to become more accurate with respect to U.

The baseline eVolCheck's algorithm is not designed for refreshing summaries besides completely re-verifying U. In contrast, after integrating it with PVAIR, this approach becomes natural. Let $\Delta_{S,U}$ denote the behavioral difference between S and U, i.e., the set of behaviors of S not present in U. If the set $\Delta_{S,U}$ is non-empty, it could be exploited by PVAIR to generate the partial interpolants that represent new summaries for each function in U. These updated summaries are still guaranteed to preserve safety of the assertion a in U.

Experiments. We experimented with PVAIR on a set of 10 pairs of different benchmarks written in C. They were crafted by us based on the prior experience with eVolCheck. In particular, we focused on cases which are hard for the default implementation of eVolCheck. In particular, all benchmarks were quite simple (tens of LOC), however, they use non-linear arithmetic operations. After the required propositional encoding (i.e., bit-blasting), the resulting large-size formulae have been a bottleneck for solving and interpolation using the original eVolCheck approach.

In our experiments, for each pair of programs, S and U, we obtained U from the corresponding S by assigning guards in some conditional expressions. In particular, we replaced the code `if P do A else do B` by `assume(P); A`. This is equivalent to assigning `P = true`, and $\Delta_{S,U}$ consists of the behaviors specified by `assume(¬P); B`. For simplicity, in our experiments, we assumed that $\Delta_{S,U}$ affected only a single function f.

To study the behavior of eVolCheck in this scenario, we constructed examples of the form given in Fig. 17. The idea is that the programmer determines the initial version of the function $\mathbf{f^S}$ (*left top*) to be too complex with respect to the assumptions and the assertion (*right bottom*), and re-implements the function as $\mathbf{f^U}$ (*left bottom*). Function $\mathbf{f^U}$ has fewer behaviors than function $\mathbf{f^S}$ and therefore

```
f^S(x1, x2, x3, x4):
  if (*):
    res := cplx1(x1, x2, x3, x4)
  else:
    res := cplx2(x1, x2, x3, x4)
  if (*):
    res := res + cplx1(x1, x2, x3, x4)
  else:
    res := res + cplx2(x1, x2, x3, x4)
  return res

f^U(x1, x2, x3, x4):
  res := cplx2(x1, x2, x3, x4)
  res := res + cplx2(x1, x2, x3, x4)
  return res
```

```
cplx1(x1, x2, x3, x4):
  return 5 × (x1 × x2 + x3 × x4)

cplx2(x1, x2, x3, x4):
  return 5 × (x1 + x2 × x3 + x4)

main:
  assume(3 < b_i < 20 for all i = 1, ..., 4)
  res := f(b1, b2, b3, b4)
  assert(res > b1 + b2 * b3)
```

Fig. 17 An eVolCheck example benchmark. An initial and an optimized version of the function **f** (*left*, $\mathbf{f^S}$ and $\mathbf{f^U}$, respectively), and the functions $\mathbf{cplx_1}, \mathbf{cplx_2}$ simulating complex operations and a main function (*right*).

the summary of $\mathbf{f^S}$ is valid for $\mathbf{f^U}$. In the examples, the number of conditional expressions per function ranges from one to three.

The results of our experiments are shown in Tab. 3 and Tab. 4[2]. For each S and U, we identified $\Delta_{S,U}$ and obtained the set of conditional expressions to be assigned in S (column *#var. assigned*). Then we performed two steps: (1) *constructed* the summary of f without/with $\Delta_{S,U}$; and (2) *validated* the corresponding summaries of f with respect to the new code in U. This experiment illustrates to what extent:

(a) the use of PVAIR yields smaller summaries than the ones of PeRIPLO,
(b) the use of smaller summaries improves the overall performance of eVolCheck.

We collected the size of the resulting interpolants and total verification time needed to perform steps (1) and (2). We used the Pudlák interpolation algorithm [17] (which is more suitable for function summaries than McMillan's) to construct the "orig." interpolants (the ones constructed without $\Delta_{S,U}$). As to the runtime spent by particular phases of verification, the largest amount of it is spent by SAT solver, while translation of the program into the formula and generation of the interpolants are negligible.

Conclusions of the experimentation. As can be seen from the tables, the use of PVAIR helped eVolCheck to make the function summaries up to 60% smaller compared to the ones produced by PeRIPLO (columns *#var. orig* vs. *#var. PVAI*, and *#cl. orig* vs. *#cl. PVAI*), while taking almost no additional time (columns *boot. orig.* vs. *boot. PVAI*). Furthermore, eVolCheck spent up to 60% less effort in the validating step (columns *upgr. orig.* vs. *upgr. PVAI*), in which the model checker finally confirmed that the new code is safe. In other words, in the considered verification scenario and driven by PVAIR, eVolCheck improved both the size of the summaries and the overall verification time, without sacrificing soundness of the entire model checking procedure.

[2] Note that the implementation of PVAIR does not involve any parallel computation—this is planned as future work.

C program		Interpolant (function summary) size			
name	#var	#var orig.	#var PVAI	#cl orig	#cl PVAI
Test 0	3 vars	15227	62.61 %	45192	62.21 %
Test 1	1 var	23273	78.46 %	69330	78.31 %
Test 2	2 vars	31278	59.19 %	93345	58.98 %
Test 3	1 var	12236	63.80 %	36219	63.31 %
Test 4	2 vars	20447	74.57 %	60852	74.37 %
Test 5	3 vars	24716	32.50 %	73659	32.05 %
Test 6	3 vars	33076	37.89 %	98739	37.58 %
Test 7	1 var	12478	57.47 %	36945	56.91 %
Test 8	1 var	21201	50.42 %	63114	50.04 %
Test 9	2 vars	20314	39.71 %	60453	39.22 %

Table 3 eVolCheck verification statistics—interpolant size.

C program		Verification time (sec)			
name	#var	boot. orig.	boot. PVAI	upgr. orig.	upgr. PVAI
Test 0	3 vars	18.93	99.17 %	4.025	65.96 %
Test 1	1 var	10.36	99.24 %	4.034	77.79 %
Test 2	2 vars	8.71	100.32 %	3.878	57.61 %
Test 3	1 var	7.34	100.12 %	1.256	71.50 %
Test 4	2 vars	12.40	101.94 %	2.982	81.35 %
Test 5	3 vars	12.20	102.94 %	3.855	39.46 %
Test 6	3 vars	12.63	102.16 %	7.951	40.05 %
Test 7	1 var	8.88	100.29 %	2.350	57.96 %
Test 8	1 var	14.46	97.55 %	3.706	50.94 %
Test 9	2 vars	21.42	101.26 %	4.581	40.30 %

Table 4 eVolCheck verification statistics—computation time.

7 Conclusion

In this article, we described the Partial Variable Interpolation System, extending LIS with a partial variable assignment. The assignments are used to focus the computed interpolants on particular sub-problems. The main motivation for this step was two-fold. The first goal was to make the computed interpolant smaller to improve the efficiency of the entire model checking process. The second goal was to make the computation of the interpolant itself more efficient in terms of both time and memory. Our experiments show that there is a clear improvement in both points.

We have also presented proofs of important properties of focused interpolants—in particular, the proof of correctness of our LPAIS, and the proof that focused interpolants computed upon a single refutation satisfy, under some additional assumptions, the path-interpolation property. Many interpolation-based verification tools require the property.

As future work, we plan to fully integrate PVAI in software verification process, through implementing the approach into SMT solvers such as the OpenSMT2 [11] solver. In particular, we are interested in applying PVAI in parallelization of software verification by case splitting upon selected program variables. This is a similar approach as in the example code in Fig. 17; the variable representing a branching condition is to be used as a case splitting variable. Then, two interpolants (repre-

senting summaries) are computed in parallel, one for the case when the condition holds, the other for the case when the condition does not hold. The advantage of this approach is in splitting the verification formula into two smaller ones (each one omitting the non-reachable branch) and computing them in parallel. Further, if there is an update of the code in a single branch only, just one summary has to be re-checked and potentially re-computed. The only possible disadvantage we currently envision is that if the condition variable is not local to the function where it is used, the case splitting has to propagate to lower levels of the call tree, affecting thus interpolant computation there as well. This way, the number of the summaries for a single functions would grow with all the possible case splitting values, i.e., exponentially. As a future work, we will research on the methods to decide what variables are suitable for case splitting as well as how to limit the total number of case splittings on a single path in the call tree. Nonetheless, our preliminary results indicate that this can further improve the model checking performance.

References

1. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2012.
2. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation – 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2012.
3. Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In Madhusudan and Seshia [14], pages 672–678.
4. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.
5. Gianpiero Cabodi, C. Loiacono, and D. Vendraminetto. Optimization techniques for Craig Interpolant compaction in Unbounded Model Checking. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1417–1422. EDA Consortium San Jose, CA, USA / ACM DL, 2013.
6. William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.
7. Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant Strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.
8. Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
9. Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. eVolCheck: Incremental Upgrade Checker for C. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2013.

10. Arie Gurfinkel, Simone Fulvio Rollini, and Natasha Sharygina. Interpolation Properties and SAT-Based Model Checking. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis – 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 255–271. Springer, 2013.
11. Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. OpenSMT2: An SMT Solver for Multi-core and Cloud Computing. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 547–553, Cham, 2016. Springer International Publishing.
12. Pavel Jančík, Leonardo Alt, Grigory Fedyukovich, Antti E. J. Hyvärinen, Jan Kofroň, and Natasha Sharygina. PVAIR: Partial Variable Assignment InterpolatoR. In *To appear in Fundamental Approaches to Software Engineering (FASE) 2016, LNCS 9633*, 2016.
13. Pavel Jančík, Jan Kofroň, Simone Fulvio Rollini, and Natasha Sharygina. On Interpolants and Variable Assignments. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 123–130. IEEE, 2014.
14. P. Madhusudan and Sanjit A. Seshia, editors. *Computer Aided Verification – 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer, 2012.
15. Kenneth L. McMillan. Interpolation and SAT-Based Model Checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
16. Kenneth L. McMillan. Lazy Abstraction with Interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.
17. Pavel Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Journal of Symbolic Logic*, 62(3):981–998, 1997.
18. Simone Rollini, Roberto Bruttomesso, and Natasha Sharygina. An Efficient and Flexible Approach to Resolution Proof Reduction. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing – 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, volume 6504 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2010.
19. Simone Fulvio Rollini, Leonardo Alt, Grigory Fedyukovich, Antti Eero Johannes Hyvärinen, and Natasha Sharygina. PeRIPLO: A Framework for Producing Effective Interpolants in SAT-Based Software Verification. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 683–693. Springer, 2013.
20. Simone Fulvio Rollini, Roberto Bruttomesso, Natasha Sharygina, and Aliaksei Tsitovich. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, 45(1):1–41, 2014.
21. Simone Fulvio Rollini, Ondrej Sery, and Natasha Sharygina. Leveraging Interpolant Strength in Model Checking. In Madhusudan and Seshia [14], pages 193–209.
22. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments: 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, pages 1–21, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
23. SAT Competition. `http://www.satcompetition.org/`.
24. O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 114–121, Oct 2012.
25. Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis – 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 203–207. Springer, 2012.

26. Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 160–175, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
27. O. Tange. GNU Parallel – The Command-Line Power Tool. *The USENIX Magazine*, 36(1):42–47, Feb 2011.
28. Stefano Tonetta. *Abstract Model Checking without Computing the Abstraction*, pages 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
29. Grigori S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II, Volume 8 of Seminars in Mathematics, V. A. Steklov Mathematical Institute*, Leningrad, 1969. Consultants Bureau.
30. Yakir Vizel and Orna Grumberg. Interpolation-Sequence based Model Checking. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 1–8. IEEE, 2009.
31. Yakir Vizel, Arie Gurfinkel, and Sharad Malik. Fast Interpolating BMC. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, number 9206 in Lecture Notes in Computer Science, pages 641–657. Springer International Publishing, July 2015. DOI: 10.1007/978-3-319-21690-4_43.