PVAIR: Partial Variable Assignment InterpolatoR*

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

[1] University of Lugano, Switzerland, {name.surname}@usi.ch
[2] Charles University in Prague, Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems, Czech Republic
{name.surname}@d3s.mff.cuni.cz

Abstract Despite its recent popularity, program verification has to face practical limitations hindering its everyday use. One of these issues is scalability, both in terms of time and memory consumption. In this paper, we present Partial Variable Assignment InterpolatoR (PVAIR) – an interpolation tool exploiting partial variable assignments to significantly improve performance when computing several specialized Craig interpolants from a single proof. Subsequent interpolant processing during the verification process can thus be more efficient, improving scalability of the verification as such. We show with a wide range of experiments how our methods improve the interpolant computation in terms of their size. In particular, (i) we used benchmarks from the SAT competition and (ii) performed experiments in the domain of software upgrade checking.

1 Introduction

Symbolic model-checking algorithms rely on expressing a verification problem as a logical formula and determining whether the formula satisfies a given property. Many sub-tasks of model-checking, such as computing safe inductive invariants for programs and summarizing functionality with respect to properties critical to program correctness, rely on over-approximating parts of the formula. To keep the formal verification manageable and the run time low it is critical that the over-approximations are suitable for the model-checking task at hand. *Craig interpolation* [7] is a process for computing over-approximations of first-order formulas that has proven useful in both program verification and automatic approximation refinement [15]. The idea in 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 but still entails the properties of interest with respect to the rest of the formula. The *Labeled Interpolation System* (LIS) [8] is a widely used framework for computing Craig interpolants in propositional logic from a resolution refutation. The flexibility of LIS allows it to be used in a variety of verification tasks that place additional requirements for the interpolants [18].

In some tasks, (e.g., when proving safety of certain types of program updates or speeding up model-checking with parallel computing) it is useful to compute over-approximations of the formula under assumptions which are specific to the particular

* This work was partially supported by the Grant Agency of the Czech Republic project 14-11384S and by the SNF projects number 200020_163001 and 200021_153402.

application problem. However, the LIS framework in its original form does not allow for computing interpolants under assumptions. There are several reasons why such *focused interpolants* would be beneficial in particular in the LIS framework. Firstly, the focused interpolants are in general smaller and therefore more manageable for the model checker. Secondly, the properties of interpolants provided by the LIS framework, such as the path interpolation property [13], can be preserved in the focused interpolants. Thirdly, several focused interpolants can be computed from a single resolution refutation, while constructing a resolution refutation is computationally expensive. In [12], we introduced an interpolation system exploiting partial variable assignments to improve efficiency of interpolant computation. We proved that following a set of requirements put on labeling during interpolation results in interpolants with the path interpolation property, which is required by some verification tools, e.g. [1], to work.

This paper presents the *Partial Variable Assignment InterpolatoR* (PVAIR), the first implementation that is able to construct such focused interpolants. The implementation is based on the *Labeled Partial Assignment Interpolation System* (LPAIS) [12], an extension of LIS which supports focusing the interpolant in the manner required by the verification applications. The PVAIR solution is generic and can be used in various model checking-based scenarios. In this paper, in addition to providing the description of the tool architecture, we also report an initial experimental study on how the interpolants constructed with PVAIR behave in different example tasks. The results show a significant improvement in both interpolant size and the overall model checking time, suggesting that the approach is viable for constructing focused interpolants.

The general intuition behind the applications of PVAIR is that sometimes a symbolic model checker can provide a partial truth assignment for the formula being verified, coming from the knowledge of the program structure and meaning of the variables. As a result, some constraints of the formula can get satisfied; the LPAIS framework allows for removing such clauses during the interpolant computation. This improves the interpolation in two ways: the interpolation becomes faster, and the resulting interpolant can be significantly smaller. Because of the latter the interpolants can be handled in a more efficient way during the subsequent computation. PVAIR is built on top of the open-source tool PeRIPLO [18], which provides resolution proofs and is able to optimize the proofs for interpolation through transformations. PeRIPLO has been used in various verification projects, including function summarization in eVolCheck [10] and FunFrog [22], both as an interpolation engine and as a SAT solver.

We experimentally studied the performance of PVAIR on a set of its potential applications. We compared it to PeRIPLO during computation of a summary for a particular function using eVolCheck. In this experiment, PVAIR was used to rule out the program paths that do not call the function. We also applied PVAIR in more generic settings, when constructing interpolation problems from a subset of the SAT Competition benchmarks. This experiment resembles closely the scenario of computing focused interpolants for a divide-and-conquer approach for parallel model checking. In both types of benchmarks, we report a substantial reduction in interpolant sizes. As shown in the eVolCheck use case, smaller interpolants also result in considerably faster upgrade-checking steps.

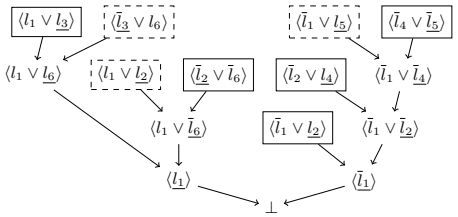

Figure 1: Refutation resolution proof; the clauses from A-part and B-part are in dashed and full boxes, respectively.

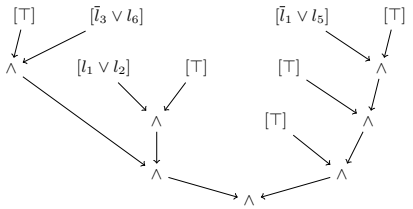

Figure 2: McMillan's interpolant.

2 Preliminaries and Background Theory

A *literal* is a Boolean variable l or its negation $\bar{l}$. A *clause* is a disjunction over a set of literals. We use angle brackets $\langle \Theta \rangle$ to denote the clause built from the literals in set Θ. A propositional formula in Conjunctive Normal Form (CNF) is a conjunction (or equivalently set) of clauses. A *resolution proof* for a set of clauses Φ is a rooted DAG with each node having either no antecedents (*leaf node*) or exactly two antecedents (*inner node*). Each node in the resolution proof is associated with node clause; from now on we use proof node and corresponding node clause equivalently. A leaf node corresponds to an input clause from Φ. Each inner node with two antecedents $\langle \Theta_1, p \rangle$ and $\langle \Theta_2, \bar{p} \rangle$ has node clause $\langle \Theta_1, \Theta_2 \rangle$, thus representing a resolution where p is the *pivot* variable.

Given an unsatisfiable CNF formula Φ and its (A,B)-partitioning into $A \wedge B$ parts, a Craig interpolant [7] is a formula I such that I is implied by A ($\models A \Rightarrow I$), unsatisfiable with B ($\models B \wedge I \Rightarrow \bot$), and defined over common symbols (variables) of A and B. An interpolant can be seen as an over-approximation of A still being strong enough to be unsatisfiable with B.

Example 1: Fig. 1 shows a resolution refutation proof for CNF formula $\Phi = \langle l_1 \vee l_2 \rangle \wedge \langle \bar{l}_3 \vee l_6 \rangle \wedge \langle \bar{l}_1 \vee l_5 \rangle \wedge \langle l_1 \vee l_3 \rangle \wedge \langle \bar{l}_2 \vee \bar{l}_6 \rangle \wedge \langle \bar{l}_4 \vee \bar{l}_5 \rangle \wedge \langle \bar{l}_2 \vee l_4 \rangle \wedge \langle \bar{l}_1 \vee l_2 \rangle$. Assume a (A,B)-partitioning with A consisting of the conjunction of the first three clauses and B of the remaining five clauses. There might not be just a single interpolant for an unsatisfiable formula; many different ones of various strengths can exist. Formula $I_1 \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge (\bar{l}_6 \vee l_2)]) \wedge (\bar{l}_1 \vee l_5)$ is one of the possible interpolants which can be computed from the proof in Fig. 1 using LIS. Fig. 2 shows how McMillan's interpolant $I_2 \equiv (l_1 \vee l_2) \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$ can be derived (after constant propagation) from the proof in Fig. 1, e.g., by LIS or LPAIS with an empty assignment. Note that for convenience we write the partial interpolant associated to a particular node of the proof into brackets.

As an over-approximation, Craig interpolants express properties for all models of the formula. However, this might be unnecessarily strong for some applications. For example, while constructing a function summary through interpolation, it is possible to

consider only the models corresponding to the paths going via the summarized function. Based on the encoding of the function body, a variable assignment blocking all the other paths can be derived. This applies also for the case of Abstract Reachability Graphs (ARGs). The label of a particular ARG node is an over-approximation of reachable states at that node. Since the paths in ARG which do not go via the node cannot influence the reachable states at that node, for each node it is possible to compute variable assignment blocking these paths; in other words, the assignment permits only the models corresponding to paths via the node. The node labels are computed by interpolation, however it is actually enough to compute a formula that is an interpolant for the models consistent with the assignment.

Focused interpolants. A *Partial Variable Assignment* (PVA) π assigns value $True$ resp. $False$ to some variables from formula Φ; alternatively, PVA can be seen as a conjunction of literals. Given a partial variable assignment π, a set of clauses A can be partitioned into A_π – a subset of clauses from A satisfied by the assignment, and the remaining clauses $A_{\overline{\pi}}$ which are not satisfied by π. For a given unsatisfiable formula Φ, its partitioning into $A \wedge B$ and a partial variable assignment π, a *Partial Variable Assignment Interpolant* [12], shortly *focused interpolant*, is a formula I such that $\pi \models A \Rightarrow I$ and $\pi \models B \wedge I \Rightarrow \bot$ and I is defined over unassigned shared variables between $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$, i.e., the symbols common to the π-unsatisfied parts of A and B. In other words, it is an interpolant, but only for models which agree on the values of variables assigned by π. Due to the weakened requirements, the focused interpolants can be of a smaller size compared to the Craig interpolants. The focused interpolants can be alternatively seen as Craig interpolants for the unsatisfied parts of the input – *sub-problem*, i.e., for $A_{\overline{\pi}} \wedge B_{\overline{\pi}}$ where literals falsified by the assignment are removed.

Example 1 (cont.): Let us assume assignment $\pi \equiv \bar{l}_2$ (i.e., assigning $False$ to variable l_2) and the set of clauses from our previous example. Given the assignment, B can be split into $B_\pi \equiv \langle \bar{l}_2 \vee \bar{l}_6 \rangle \wedge \langle \bar{l}_2 \vee l_4 \rangle$ and $B_{\overline{\pi}} \equiv \langle \bar{l}_4 \vee \bar{l}_5 \rangle \wedge \langle \bar{l}_1 \vee l_2 \rangle$. A_π is empty thus $A_\pi \equiv \top$ and $A_{\overline{\pi}} \equiv A$.

Craig and focused interpolants differ in the variables which could 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_5 and l_6. Since focused interpolants consider for the shared variables only unsatisfied parts of A resp. B (i.e., $A_{\overline{\pi}}$ and $B_{\overline{\pi}}$), fewer variables are shared; in our example only l_1 and l_5 could appear in a focused interpolant, which are those which can appear in a Craig interpolant for the sub-problem.

Given an assignment 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 above example result in $I_1[\pi] \equiv (l_1 \vee [(l_6 \vee \bar{l}_3) \wedge \bar{l}_6]) \wedge (\bar{l}_1 \vee l_5)$ and $I_2[\pi] \equiv l_1 \wedge (\bar{l}_3 \vee l_6) \wedge (\bar{l}_1 \vee l_5)$. None of them is a valid focused interpolant since both contain variable l_6. Note that $I_2[\pi]$ can be equivalently rewritten as $l_1 \wedge l_5 \wedge (\bar{l}_3 \vee l_6)$x; in general, such a transformation requires a complex analysis and not all interpolants can be transformed into focused interpolants as I_1 shows. This means that the aforementioned techniques can be used to reduce the size of the formula, however not to compute focused interpolants. Below we introduce a method to compute focused interpolants for propositional logic which produces interpolants smaller than the approach above.

Leaf v:		$\langle\Theta\rangle, [I]$	
$I =$	$-\langle\Theta\rangle\vert_{b,\pi}$	if $\langle\Theta\rangle \in A_{\overline{\pi}}$	Hyp-$A_{\overline{\pi}}$
	$\neg\langle\Theta\rangle\vert_{a,\pi}$	if $\langle\Theta\rangle \in B_{\overline{\pi}}$	Hyp-$B_{\overline{\pi}}$
	$\top$	if $\langle\Theta\rangle \in A_\pi \cup B_\pi$	Hyp-A_π, Hyp-B_π
Inner vertex v:		$\dfrac{v_1 : \langle p, \Theta_1\rangle, [I_1] \qquad v_2 : \langle \overline{p}, \Theta_2\rangle, [I_2]}{\langle\Theta_1, \Theta_2\rangle, [I]}$	
$I =$	$I_1 \vee I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = a$	Res-a
	$I_1 \wedge I_2$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = b$	Res-b
	$(I_1 \vee p) \wedge (I_2 \vee \overline{p})$	if $\mathsf{Lab}(v_1, p) \sqcup \mathsf{Lab}(v_2, \overline{p}) = ab$	Res-ab
	I_2	if $\mathsf{Lab}(v_1, p) = d^+$	Res-d^+
	I_1	if $\mathsf{Lab}(v_2, \overline{p}) = d^+$	Res-d^+

Table 1: Labeled Partial Assignment Interpolation System

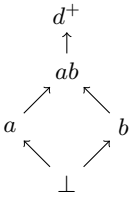

Figure 3: Lattice of labels ($\sqcup$).

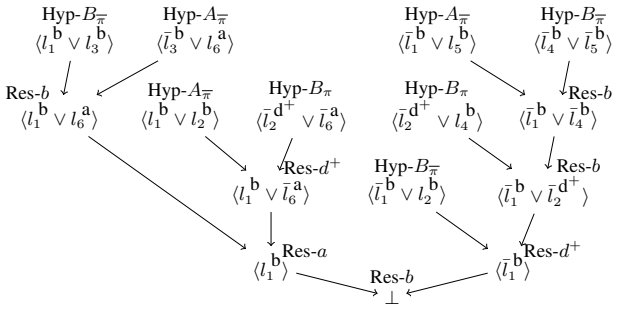

Figure 4: Labeled proof and rules to be applied at proof nodes.

Labeled Partial Assignment Interpolation System (LPAIS) — an extension of the Labeled Interpolation System [8] — yields focused interpolants from the resolution refutation of $A \wedge B$.

In LPAIS, each literal in the clauses of the resolution proof is assigned a label a, b, ab, or d^+. Labels a, b, and ab have the same meaning as in LIS, while the label d^+ is used for the literals from the assignment π. The lattice of labels is defined by the Hasse diagram in Fig. 3. The labels are specified via a *labeling function* Lab; e.g., $\mathsf{Lab}(v_2, \overline{p})$ is the label of literal $\overline{p}$ at node v_2 of the proof. The label of a literal in an inner node v is computed using join operator $\sqcup$ (defined by Fig. 3) from the labels of the literal in the antecedent nodes ($\mathsf{Lab}(v, l) = \mathsf{Lab}(v_1, l) \sqcup \mathsf{Lab}(v_2, l)$, where v_1 and v_2 are the antecedent nodes of v). Formal definition of labeling function as well as the requirements that labels must satisfy are described in [12].

Example 1 (cont.): Fig. 4 shows how LPAIS assigns labels to literals; the label of a literal is shown as superscript. When choosing the strongest possible labeling, LPAIS yields, for empty assignments, McMillan's interpolants; in particular, only variables occurring in $A_{\overline{\pi}}$ but not in $B_{\overline{\pi}}$ are labeled a (i.e., l_6), all the others (except for the literals from the assignment) re-labeled b.

The labeled partial assignment interpolation system assigns a *partial interpolant* $[I]$

to each proof node according to the rules described in Tab. 1. The partial interpolants of the leaf nodes are directly constructed from the node clauses (it means those forming $A \wedge B$) using the rules in the upper part of Tab.1. The applied Hyp-$*$ rule is determined by the set inclusion check in the middle column; in particular by occurrence of the node clause in $A_{\overline{\pi}}$, A_{π}, B_{π} and $B_{\overline{\pi}}$. A partial interpolant for the Hyp-$A_{\overline{\pi}}$ rule, defined as $\langle \Theta \rangle|_{b,\pi}$, represents a clause which is created from the node clause $\langle \Theta \rangle$ by omitting the literals over the π-assigned variables and those whose label differs from b. In particular node clause $\langle \bar{l}_3^{\,\mathrm{b}} \vee l_6^{\,\mathrm{a}} \rangle$ yields partial interpolant $\langle \bar{l}_3^{\,\mathrm{b}} \vee l_6^{\,\mathrm{a}} \rangle|_{b,\pi} \equiv [l_3]$. The leaf nodes with clauses satisfied by π have the partial interpolant $\top$.

For inner nodes, the rule from Tab. 1 is chosen based on the labels of the pivot in the antecedents (denoted by v_1 and v_2). Note the Res-d^+ rules, which correspond to the case where the pivot is satisfied by the assignment in one of the antecedents. In these cases, the partial interpolant is the same as the partial interpolant in the antecedent not being satisfied by the assignment; due to such nodes the size of the LPAIS interpolant is smaller compared to the LIS interpolant.

Example 1 (cont.): Fig. 5 shows how focused interpolant $I_\pi \equiv l_1 \vee \bar{l}_3$ for our example can be derived. Note the dotted arrows at nodes corresponding to Res-d^+ resolutions; they highlight the antecedents whose partial interpolants are ignored and their sub-trees do not contribute to final focused interpolant. Also note that the focused interpolant I_π is smaller compared to both $I_1[\pi]$ and $I_2[\pi]$ from the examples above.

An assignment applied onto (interpolant) formula (i.e., if $I[\pi]$ is computed) can reduce the size of the formula only if the assigned variable appears in the (interpolant) formula (i.e., the variable has to be shared). However, LPAIS reduce the size of the interpolants even if the assigned variable does not appear in the interpolant, since the reduction is done as a part of interpolant computation and not as a post-processing step.

PVAIR implements the LPAIS framework. The tool can generate the McMillan's [16], Pudlák's [17], and McMillan's′ [8] interpolants and their equivalents in presence of assignments. Additionally, PVAIR supports constructing different interpolants by providing different labelings for the literals in the leaves. The relative logical strength of interpolants constructed with LPAIS from the same resolution refutation is determined

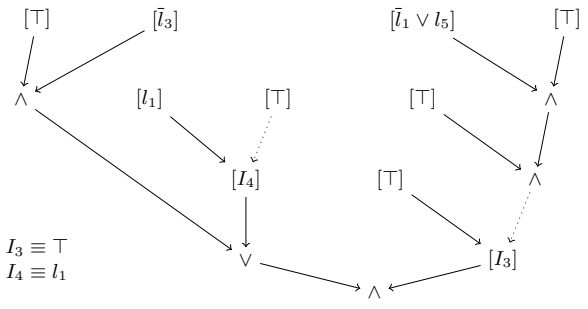

Figure 5: Focused interpolant I_π, using labeling of Fig. 4.

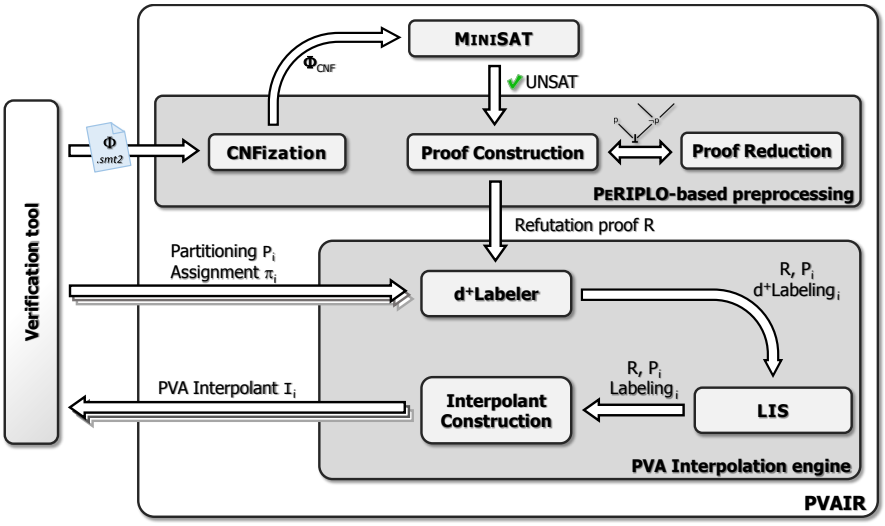

Figure 6: PVAIR architecture.

by the labeling function used. For instance, the McMillan's focused interpolants are sufficiently strong to have the path-interpolation property.

3 The Tool Architecture

The PVAIR architecture is shown in Fig. 6. It takes a propositional formula, its (A, B)-partitioning, and a partial variable assignment as input and produces focused interpolants if the input formula is unsatisfiable. The input can be provided either in a file in the SMT-LIB 2.0 format or via a C++ API.

When a verification tool decides to compute interpolants (e.g., to obtain either function summaries in the case of upgrade-checking and over-approximations of reachable states for covering checks) it constructs an input formula Φ which encodes the program being verified. Further, based on the way the input formula is constructed, the verification tool decides how to partition it (e.g., to obtain a summary of a given function) and which partial variable assignment to use (e.g., depending on the changes detected in the new version of the program). These inputs are then passed to the PVAIR tool.

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 an arbitrary form, it is transformed into CNF (the top box in Fig. 6) using an efficient, structure-sharing version of the Tseitin encoding [25]. Its satisfiability is then determined using the MINISAT 2.2.0 solver [9].

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 proof (Proof Construction). In particular, using the resolution DAG form, the proof can be compressed using well-known proof reduction techniques such as structural hashing or pivot recycling [19,20]

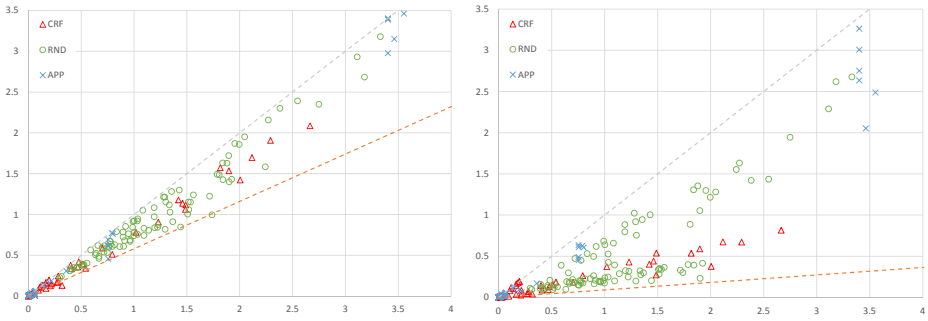

Figure 7: Comparison of interpolant sizes computed without variable assignment [x] and with one variable assigned [y] (left) and five variables assigned (right).

available in PERIPLO (Proof Reduction). The proof reduction techniques can be enabled/disabled via a configuration file or API.

Once the resolution proof R is computed, it is passed together with the partitionings and variable assignments to the interpolation engine (the bottom box in Fig. 6). 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. The first step inside the PVA interpolation engine is labeling all the literals in $A \wedge B$. The d^+Labeler will distribute d^+ labels among the literals according to the assigned variables, whereas the LIS will label the remaining literals according to the partitioning and the 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 resolution proof R to compute interpolants (Interpolant Construction).

The construction starts by computing partial 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. In each inner node, depending on the label of the pivot, a partial interpolant of the node is computed by combining the partial interpolants from the antecedent nodes (the bottom part of Tab. 1). 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.

For the details on PVAIR usage, we refer the reader to the Tutorial section of the tool web page available at http://verify.inf.usi.ch/pvair.

4 Experiments

We ran PVAIR on two types of experiments: (1) SAT Competition benchmarks and (2) 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.

Focused Itp.	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.

4.1 SAT Competition

From the way focused interpolants are computed by PVAIR it is obvious that they are smaller compared to the Craig interpolants. In this part we illustrate the actual difference. Compared to experiments on functions summaries in the latter part, SAT Competition provides us with a larger set of more heterogeneous kinds of benchmarks. This helps one to see how the reduction of the size varies among inputs from different domains.

For experiments, we chose 47 unsatisfiable benchmarks from the SAT Competition[3] 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 chose n, first n clauses of the benchmark belonged to the A-part, the remaining clauses to the B-part. No assignment is given by authors of the benchmarks, thus for each partitioning we generated 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.

Since focused interpolants can be seen as Craig interpolants for a sub-problem, for each pair of partitioning and assignment, we created the sub-problem instance and used PVAIR to computed the Craig interpolant. Sub-problems are simpler compared to the benchmark from which they were generated, so 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 proof; in contrast to focused interpolants which, for a particular benchmark, are all computed from the same proof. The path interpolation property [13], which is often exploited during program model checking, might be missing in this case.

As to the interpretation of the results: *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 generation the input) provides a reasonable assignment together with a partitioning. The interpolants for a sub-problem can be seen as an alternative to focused interpolants because of their similar meaning, however these interpolants lack the properties of the focused ones.

For comparison, we use McMillan's interpolants – a widely used approach. The proof reduction techniques were disabled; we used the default PERIPLO settings. All benchmarks were run on a Linux blade server with Xeon X5687 CPU using the timeout of 60 minutes and the memory limit of 20GB using the Parallel environment [24].

Fig. 7 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

[3] http://www.satcompetition.org/

size if no assignment is provided (Craig interpolant) while the y-axis represents the size of the focused interpolants with a single (resp. five) 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). The orange dashed line shows the average size of Craig interpolants for sub-problems. This illustrates what price is paid by focused interpolants for the path interpolation property and a single SAT solver call. Both graphs show interesting reduction in size for focused interpolants as well as substantially larger reduction in case of five assigned variables. In both graphs the same partition of the same benchmark share 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 above, reporting precise numbers. The table on the left-hand side compares the sizes of focused interpolants to Craig interpolants (in the *No assignment* row). 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. The table on the right-hand side compares the sizes of Craig interpolants for the benchmark with the Craig interpolants for sub-problems (corresponding to the assignments used in the left table). The table shows that these interpolants are on average smaller compared to the focused ones. The more variables are assigned, the bigger the difference is. While the sizes are comparable for a few assigned variables, the price paid for the path interpolation property of focused interpolants is high for larger assignments (e.g., twenty variables) .

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., PVAIR is 11% faster and requires 9% less memory on average if a single variable is assigned. The time and memory savings occur as well during the interpolant computation phase due to smaller interpolants being handled.

4.2 Applying PVAIR for Checking Software Upgrades

The usefulness of PVAIR is motivated by the tremendous role of interpolation in model checking. One of the possible applications of PVAIR is checking software upgrades by means of function summarization [23] implemented in the tool EVOLCHECK. Given a program S and an assertion a, EVOLCHECK verifies S with respect to a (i.e., proves that $S \wedge \neg a$ is unsatisfiable) and, for each function call in S, it constructs the interpolant and treats it as a *function summary*. In [21] we show that even if the constructed function summary is an over-approximation of the function behavior of S, it preserves the safety of the assertion a in S.

EVOLCHECK *validates* the computed function summaries to over-approximate the behavior of the corresponding functions of a program upgrade, 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 unsafe. 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 functionality*. Then by construction, the original summaries of S are still valid over-approximation of the new function behavior in U. But at the same time, 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 possibly shrink) the summaries to become more accurate with respect to U.

The *refreshed* summaries may be used to verify a further updated program W that additionally may *introduce new functionality* with respect to U. On the other hand, the summaries may be also used to speed up verification of a new assertion b, implanted in the code of U [21]. To enable both scenarios, the constructed summaries need to be externally stored and further migrated across the verification runs. Thus, the size of the summary also becomes important.

While EVOLCHECK does not provide a way to refresh summaries except of complete re-verification of U from scratch, PVAIR becomes particularly useful. Let $\Delta_{S,U}$ denote the behavioral difference of 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. Notably, all benchmarks used 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 `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.

The results of our experiments are shown in Tab. 3. 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:

- the use of PVAIR yields smaller summaries compared to the ones by PERIPLO,
- 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] to construct the "orig" interpolants (the ones constructed without $\Delta_{S,U}$).

C program		Interpolant (function summary) size				Verification time (sec)			
name	# var. assigned	# var. orig.	# var. PVAI	# cl. orig	# cl. PVAI	boot. orig.	boot. PVAI	upgr. orig.	upgr. PVAI
Test 0	3 vars	15227	62.61 %	45192	62.21 %	18.93	99.17 %	4.025	65.96 %
Test 1	1 var	23273	78.46 %	69330	78.31 %	10.36	99.24 %	4.034	77.79 %
Test 2	2 vars	31278	59.19 %	93345	58.98 %	8.71	100.32 %	3.878	57.61 %
Test 3	1 var	12236	63.80 %	36219	63.31 %	7.34	100.12 %	1.256	71.50 %
Test 4	2 vars	20447	74.57 %	60852	74.37 %	12.40	101.94 %	2.982	81.35 %
Test 5	3 vars	24716	32.50 %	73659	32.05 %	12.20	102.94 %	3.855	39.46 %
Test 6	3 vars	33076	37.89 %	98739	37.58 %	12.63	102.16 %	7.951	40.05 %
Test 7	1 var	12478	57.47 %	36945	56.91 %	8.88	100.29 %	2.350	57.96 %
Test 8	1 var	21201	50.42 %	63114	50.04 %	14.46	97.55 %	3.706	50.94 %
Test 9	2 vars	20314	39.71 %	60453	39.22 %	21.42	101.26 %	4.581	40.30 %

Table 3: EVOLCHECK verification statistics.

As can be seen from the table, 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.

5 Related work

This section compares the PVAIR approach with various techniques for reducing the size of an interpolant based on variable assignments, proof compression, and interpolant post-processing.

Variable assignments. Given a variable assignment, the most straightforward way to reduce the interpolant size is to apply the assignment directly onto the interpolant formula and propagate Boolean constants. This idea is used in the UFO [1] tool. Due to the tight integration into the interpolation process, LPAIS yields smaller interpolants compared to this simple approach. Since the assignment is considered by LPAIS already during the interpolant construction, this results in larger parts of the interpolant being cut away.

Proof compression. Interpolants are often derived from a resolution proof and therefore their size is roughly proportional to the size of the proof. Several methods for compressing a resolution proof exist [2,11,4,2,19,6]. Different variants of these techniques are applied in PdTRAV [5] verification framework, the PERIPLO tool, and the Skeptik [3] proof transformer, just to name a few examples. In this work, the reduction of the interpolant size is based on the fact that only a proof of the unsatisfied part of the input formula is needed. Since the omitted (i.e., satisfied) parts can be important w.r.t. other assignments, the proof compression techniques cannot remove these parts from the proof. As a result, these techniques are orthogonal and PVAIR can benefit from proof compression if applied.

Interpolant post-processing. Once an interpolant is computed, various techniques can be used to reduce its size. Such techniques include constant propagation, structural

sharing, and various equivalence and subsumption checks. PdTRAV, for example, internally uses BDD-based sweeping to detect the equivalences and balancing/rewriting over And-Inverter Graphs [14] representation to further reduce the size of an interpolant. Any such post-processing technique producing smaller equivalent formulae can be applied to the interpolants produced by the PVAIR tool.

6 Conclusions

In this paper we presented the PVAIR interpolation tool, which exploits partial variable assignments obtained from an application-specific source to compute focused interpolants. The tool uses the extension of the labeled interpolation system, LPAIS, to construct the interpolants from a resolution refutation. We presented a potential application for the focused interpolants, in particular in software upgrade checking where we require the path interpolation property. We performed an initial study using a wide range of experiments varying the size of the partial variable assignment. The results show a good improvement compared to the baseline and suggest that the approach taken for computing focused interpolants has significant potential in reducing the interpolant size and model checking time. In the future we plan to integrate the PVAIR tool into a concrete implementation of a parallel model checker as well as to study other applications of model checking where partial assignments arise naturally.

References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. From Under-Approximations to Over-Approximations and Back. In *TACAS*, pages 157–172, 2012.
2. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-Time Reductions of Resolution Proofs. In *HVC*, pages 114–128, 2008.
3. J. Boudou, A. Fellner, and B. W. Paleo. Skeptik: A Proof Compression System. In *IJCAR*, pages 374–380, 2014.
4. J. Boudou and B. W. Paleo. Compression of propositional resolution proofs by lowering subproofs. In *TABLEAUX*, pages 59–73, 2013.
5. G. Cabodi, C. Loiacono, and D. Vendraminetto. Optimization Techniques for Craig Interpolant Compaction in Unbounded Model Checking. In *DATE*, pages 1417–1422, 2013.
6. S. Cotton. Two Techniques for Minimizing Resolution Proofs. In *SAT*, pages 306–312, 2010.
7. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Symbolic Logic*, pages 269–285, 1957.
8. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant Strength. In *VMCAI*, pages 129–145, 2010.
9. N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, pages 61–75, 2005.
10. G. Fedyukovich, O. Sery, and N. Sharygina. eVolCheck: Incremental upgrade checker for C. In *TACAS*, pages 292–307, 2013.
11. P. Fontaine, S. Merz, and B. W. Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In *CADE-23*, pages 237–251, 2011.
12. P. Jancik, J. Kofroň, S. F. Rollini, and N. Sharygina. On Interpolants and Variable Assignments. In *FMCAD*, pages 123–130, 2014.

13. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In H. Hermanns and J. Palsberg, editors, *TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.
14. A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *DAC*, pages 232–237, 2001.
15. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.
16. K. L. McMillan. An Interpolating Theorem Prover. In *TACAS*, pages 16–30, 2004.
17. P. Pudlák. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Symbolic Logic*, pages 981–998, 1997.
18. S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. PeRIPLO: A Framework for Producing Effective Interpolant-based Software Verification. In *LPAR*, pages 683–693, 2013.
19. S. F. Rollini, R. Bruttomesso, N. Sharygina, and A. Tsitovich. Resolution Proof Transformation for Compression and Interpolation. *Formal Methods in System Design*, pages 1–41, 2014.
20. S. F. Rollini, O. Sery, and N. Sharygina. Leveraging Interpolant Strength in Model Checking. In *CAV*, pages 193–209, 2012.
21. O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based Function Summaries in Bounded Model Checking. In *HVC*, pages 160–175, 2011.
22. O. Sery, G. Fedyukovich, and N. Sharygina. FunFrog: Bounded model checking with interpolation-based function summarization. In *ATVA*, pages 203–207, 2012.
23. O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *FMCAD*, pages 114–121, 2012.
24. O. Tange. GNU Parallel – The Command-Line Power Tool. *The USENIX Magazine*, pages 42–47, 2011.
25. G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, 1969.