Flexible Interpolation for Efficient Model Checking

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

Faculty of Informatics, Università della Svizzera italiana
Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland

Abstract. Symbolic model checking is one of the most successful techniques for formal verification of software and hardware systems. Many model checking algorithms rely on over-approximating the reachable state space of the system. This task is critical since it not only greatly affects the efficiency of the verification but also whether the model-checking procedure terminates. This paper reports an implementation of an over-approximation tool based on first computing a propositional proof, then compressing the proof, and finally constructing the over-approximation using Craig interpolation. We give examples of how the system can be used in different domains and study the interaction between proof compression techniques and different interpolation algorithms based on a given proof. Our initial experimental results suggest that there is a non-trivial interaction between the Craig interpolation and the proof compression in the sense that certain interpolation algorithms profit much more from proof compression than others.

1 Introduction

Automated methods for formally verifying the absence of faults in a computer system are becoming increasingly important due to the significant role computers have in the society. Model checking [4] is one of the most successful approaches for formal verification. The underlying idea in model checking is to exhaustively explore a well-defined part of the state space of a system and either find errors or prove their absence in the studied state space. The problem is generally seen to be very hard and often undecidable, especially when the state space to be explored is the full state space of the system. To overcome the computational difficulty of verification many of the efficient approaches are based on describing the system using a logic-based formalism in which the lack of faults can be checked using efficient reasoning engines [3,6,7].

Many of the tools supporting traversal of the search space using logic-based, symbolic representation require methods for over-approximating parts of the state-space of the system being studied. A widely used approach is based on constructing Craig interpolants [5]. The idea is to partition an unsatisfiable logic formula into two parts $A \wedge B$ of which the A part needs to be over-approximated. Craig interpolation provides a way of constructing an interpolant I which safely over-approximates A in the sense that $A \rightarrow I$ and $I \wedge B$ is still unsatisfiable.

This paper studies a framework for constructing propositional Craig interpolants through compressed resolution refutations and the labeled interpolation system [8]. The approach itself has been discussed in our previous work [13,12,1]; the novelty of this paper is in presenting the techniques under a uniform notation and reporting initial experimental results on combining the previously studied techniques.

The presented techniques have been implemented in the PeRIPLO interpolation engine `http://verify.inf.usi.ch/periplo`. The paper is organized as follows: Section 2 discusses approaches for symbolic model checking where interpolation has natural applications and introduces interpolation and our notation for propositional logic. Section 3 discusses the approach PeRIPLO uses for compressing the refutations it creates, and Sec. 4 discusses the PeRIPLO implementation of the labeled interpolation system. We report the experimental study in Sec. 5 and conclude in Sec. 6.

2 Preliminaries

Symbolic model checking consists of determining exhaustively whether the implementation of a system conforms to its specification. The system is defined as a finite set of variables $X = \{x_1, \ldots, x_n\}$ whose values change over discrete time $t = 0, 1, \ldots$ according to a transition relation T, and at time $t = 0$ satisfy the *initial condition* $I(X)$. The initial condition and the transition relation are defined as formulas over first order logic. An assignment $\sigma(X)$ mapping each variable in X to a concrete value is a *state* of the system. Given two copies of the system variables X and X' and two states $\sigma(X)$ and $\sigma'(X')$ the system can transition from $\sigma(X)$ to $\sigma'(X')$ from time t to $t+1$ if the assignments satisfy the transition relation $T(X, X')$. In this paper we consider specifications on the *safety* of a system: A system is safe if, whenever the system starts from a state satisfying the initial conditions and transitions according to the transition relation, the visited states $\sigma_0, \sigma_1, \ldots$ never satisfy the *error condition* $E(X)$ defined in the specification also as a formula in first order logic.

To show a system unsafe it sufficies to find a sequence of states $\sigma_0, \ldots \sigma_n$ satisfying

$$I(X_0) \wedge T(X_0, X_1) \wedge \ldots \wedge T(X_{n-1}, X_n) \wedge E(X_n). \tag{1}$$

To show a system safe one needs to find a formula $R(X)$ such that

$$\models I(X) \rightarrow R(X) \tag{2}$$

$$\models R(X) \wedge T(X, X') \rightarrow R(X'), \text{ and} \tag{3}$$

$$R(X) \wedge E(X) \text{ is unsatisfiable.} \tag{4}$$

The formula $R(X)$ above is the *safe inductive invariant* which is inductive by the second tautology and safe by the third formula. It is often more practical to interchange the roles of initial and error conditions since this will make the problem solving more incremental. In some algorithms this requires the definition of the inverse of the transition relation $T^{-1}(X, X')$.

In the following we will present two model-checking applications using this generic framework that will motivate our work on computing over-approximations: the *k-induction* for unbounded model checking, and *function summarization*. Finally we give the notation for propositional satisfiability and interpolation we will use in the paper.

k-Induction. A widely used algorithm for symbolic model checking is based on constructing the safe inductive invariant $R(X)$ by means of unrolling the transition relation k times, showing that the states reached after k steps do not satisfy the error condition, and trying to obtain a safe over-approximation of the initial condition based on the proof to heuristically compute an inductive invariant. This process is known as *k-induction*. To obtain the invariant in the form given in (2) the problem is stated as an over-approximation of the initial condition. In case of over-approximation of the final condition the resulting invariant will be safe in the sense that the inverse transition function T^{-1} cannot lead to a state satisfying the initial condition starting from a state satisfying the error condition. The critical part of this algorithm is the construction of the safe transitive invariant through over-approximation of the initial condition. A widely used approach for computing the over-approximation is through interpolation.

Function Summarization. In typical programming languages the programmer imposes a logical structure for a system by organizing the program into functions. From the perspective of model checking the functions offer an interesting approach for guiding the construction of the proof of correctness through *function summaries*.

Functions and their summaries are encoded into the transition function T modularly. Let program P have a function f, and let the encoding of the function f in logic be $|f|(X)$. If a proof of safety with respect to a verification condition c for a program is obtained, the function f can be over-approximated with respect to the verification condition in a safe way by replacing the encoding $|f|(X)$ with the over-approximating encoding $|\hat{f}_c|(X)$ that can still be used to prove correctness of the condition $c(X)$.

We mention two potential uses for this approach. The first is in verifying a sequence of verification conditions. Often the error condition $E(X)$ can be split in a natural way to several verification conditions $c_0, \ldots, c_n$ such that $E(X) = \vee_{i=0}^{n} c_i(X)$. Depending on the over-approximation and the relations between the conditions in the sequence it is often possible to organize the sequence so that the strong conditions are checked before the weak conditions. For instance [9] presents a heuristic for ordering verification conditions in a way that likely results in such a sequence. In this case an initial over-approximation can be used to verify the remaining conditions. The second application of function summaries is in verifying software upgrades. Given a function f and an upgraded function f', depending on the type of the upgrade it might be possible to avoid checking the compatibility of the new version of the software against the error condition. Instead of the expensive re-verification the check can be done locally by determining whether the safe over-approximation of the encoding $|\hat{f}_c|(X)$ contains the behavior of the upgraded function $|f'|(X)$, that is, by checking whether $|\hat{f}_c|(X) \rightarrow |F'|(X)$.

Interpolation and Propositional Satisfiability. All the above presented scenarios need an approach for constructing over-approximations of parts of the formula in Eq. (1). A widely used framework for this purpose is the Craig interpolation [5]. In this work we study in particular the *proof compression* and the *labeled interpolation system* [8] for propositional satisfiability.

Propositional satisfiability provides a convenient an expressive language for presenting instances of different model checking problems. Given finite set of Boolean variables B, the set of literals over B is $\{p, \neg p \mid p \in B\}$. A *clause* is a set of literals and a *formula in conjunctive normal form* (CNF) is a set of clauses. We use interchangeably the notation $\{l_1, \ldots, l_n\}$ and $l_1 \vee \ldots \vee l_n$, where l_i are literals, to denote clauses. Given a clause n, the set $\mathit{vars}(n) = \{p \mid p \in n \text{ or } \neg p \in n\}$ gives the variables of n.

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

3 Methods of Proof Compression

A common approach for constructing interpolants is to compute a resolution refutation and label the refutation iteratively in a way that finally results in an interpolant. Since the resolution proof is often big and the interpolant size is one of the critical factors determining the usability of the interpolant it is preferable to obtain as small interpolants as possible. This section gives an in-depth view of the techniques implemented in the PeRIPLO tool for compressing the refutation once it has been constructed. In particular we cover the *local transformation framework*, the *pivot recycling algorithm* and an approach for delaying resolution steps that involve a unit clause as an antecedent.

The local transformation framework. In our experiments an important factor making resolution proofs of SAT solvers big is that the solver often resolves on a pivot several times. This type of redundancy can always be removed from a refutation and the resulting refutation will remain sound. The local transformation framework [13] addresses this issue. The framework consists of two rules for reducing a proof, complemented with two reshuffling rules that are employed to give more opportunities for the application of the two reducing rules. The four reduction and reshuffling rules are presented in Fig. 1. The restrictions on the application on the rule are listed on the leftmost column.

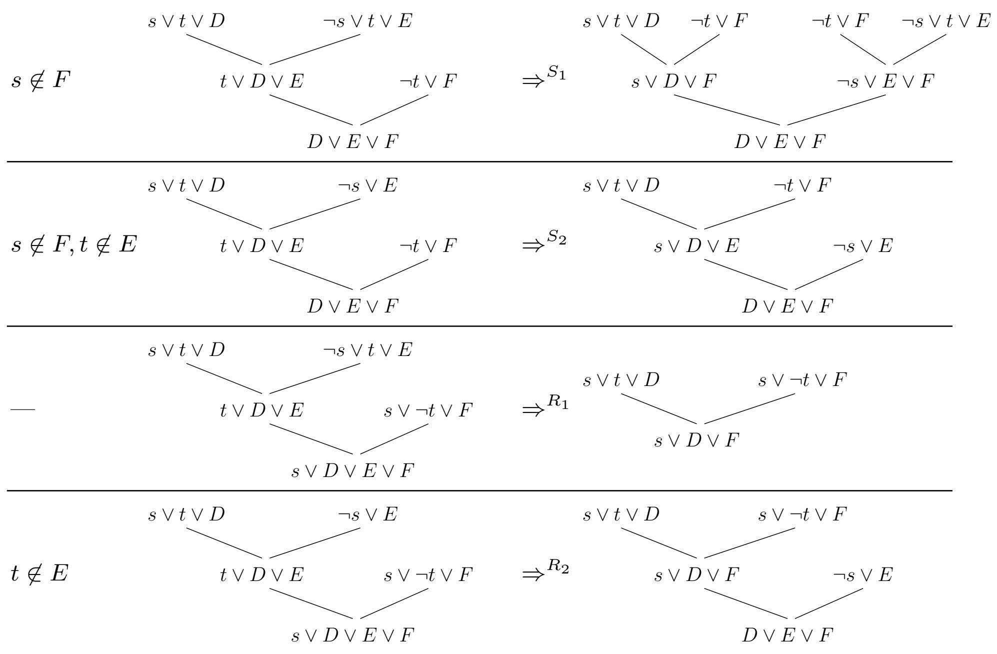

Fig. 1. Transformation rules

The PeRIPLO system uses the local transformation system to detect and remove redundancies from a refutation. The algorithm for applying the system is given in Fig. 2. The critical part of the algorithm is on lines 8–9 where the algorithm identifies a *context*, an environment which matches one of the rules in Fig. 1. The context consists of the two pivots p and q and the surrounding clauses, and since the system is symmetric the resolvents have both *left* and *right* contexts. Once a context is found the algorithm applies heuristically one of the transformation rules on the context on line 10.

The use of the transformation rules might render the refutation invalid if a clause is a resolvent in more than one resolution steps. To avoid the problem the rules R_1 and R_2 are not used in such cases. Finally the lines 12, 14, and 16 take care of the cases where resolution step has become useless due to the compression.

The pivot recycling algorithm. While the removal of the doubly appearing pivots can be done with the proof transformation system of Fig. 1, it is often useful to combine the approach with a more aggressive approach based on reachability

```
**Input** : $R$ — A refutation
  $T$ — A time limit
**Output** : $R'$ — a compressed refutation
while $T$ is not surpassed:
  $TS$ := topologically sorted list of clauses in $R$
  for $n \in TS$:
    if $n$ is not a leaf:
      $p := piv(n)$
      if $\neg p \in n^-$ and $p \in n^+$:
        $n := (n^- \cup n^+) \setminus \{p, \neg p\}$
        $lc$ := left context of $n$
        $rc$ := right context of $n$
        $ApplyRule(lc, rc)$
      else if $p \in n^+$:
        Substitute $n$ with $n^-$
      else if $\neg p \in n^-$:
        Substitute $n$ with $n^+$
      else
        Heuristically choose either $n^+$ or $n^-$ and substitute $n$ with it
```

Fig. 2. The Local Transformation Framework Algorithm.

on the refutation. One way of implementing the safe removal of extra resolution steps in a refutation DAG is to prevent the removal operation on resolvents that are used in more than one resolution step. However this approach is too restrictive since often the literals are resolved on other paths as well. For this purpose PeRIPLO uses the recycle pivots with intersection algorithm, presented in [10] and based on the original recycle pivots algorithm of [2]. We present an implementation adapted from [13] in Fig. 3, designed for a slightly more general case where the root of the refutation might contain a non-empty clause. The algorithm takes as input a refutation R and computes for each clause n in R the set of literals that can be safely removed from the literal into the set RL. The respective literals in $n \cap RL[n]$ are then removed from n and the algorithm guarantees that the refutation can be transformed to a valid refutation afterwards. The critical reasoning is done on lines 14, 20, 25, and 29 where the information on which literals can be removed on other paths where a resolvent n is resolved is used to refine the removable literals for its parents n^- and n^+.

Delaying unit resolution. A good heuristic for reducing the size of the refutation is to move the resolution steps where one of the resolvents is a unit clause to the root. This is useful since it gives a natural way of guaranteeing that the unit clauses are resolved only once in the refutation. The PeRIPLO solver implements this idea as the *PushdownUnits* algorithm [13] by identifying sub-refutations that end in a unit clause, detaching them from the refutation and, if necessary, attaching them above the resolution step resulting in the root.

```
**Input** : $R$ — A refutation
**Output** : $RL$ — the mapping from resolvents to the literals that can be removed in them
$TS$ := topologically sorted list of clauses in $R$
$RL := \emptyset$ // The set of removable literals
for $n \in TS$:
  if $n$ is not a leaf:
    if $n$ is the root:
      $RL[n] := \{\neg p \mid p \in n\}$
    else:
      $p := piv(n)$
      if $p \in RL[n]$:
        $n^+ := null$
        if $n^-$ not seen yet:
          $RL[n^-] := RL[n]$
          Mark $n^-$ as seen
        else $RL[n^-] := RL[n^-] \cap RL[n]$
      else if $\neg p \in RL[n]$:
        $n^- := null$
        if $n^+$ not seen yet:
          $RL[n^+] := RL[n]$
          Mark $n^+$ as seen
        else $RL[n^+] := RL[n^+] \cap RL[n]$
      else if $p \notin RL[n]$ and $\neg p \notin RL[n]$:
        if $n^-$ not seen yet:
          $RL[n^-] := RL \cup \{p\}$
          Mark $n^-$ as seen
        else $RL[n^-] := RL[n^-] \cap (RL[n] \cup \{p\})$
        if $n^+$ not seen yet:
          $RL[n^+] := RL[n] \cup \{\neg p\}$
          Mark $n^+$ as seen
        else $RL[n^+] := RL[n^+] \cap (RL[n] \cup \{\neg p\})$
return $RL$.
```

Fig. 3. The *RecyclePivotsWithIntersection* algorithm

The PeRIPLO proof compression algorithm. The PeRIPLO system uses an approach for proof compression that combines both the pivot recycling algorithm presented in Fig. 3 and the proof reduction framework (Fig. 2). The hybrid algorithm is presented in Fig. 4. The algorithm calls as the first step the procedure for moving unit resolutions to the root of the refutation and then repeatedly calls the functions *RecyclePivotsWithIntersection* and *ReduceAndExpose* to gradually obtain a more compact proofs.

```
Input : $R$ — A refutation;
  $I$ — the number of loop iterations;
  $T$ — A time limit for the proof reduction framework
Output : $R'$ — A compressed refutation
1 $R' := PushdownUnits(R)$
2 for $i = 0$ to $I$
3     $R' := RecyclePivotsWithIntersection(R')$
4     $R' := ReduceAndExpose(R', T)$
5 return $R'$.
```

Fig. 4. The hybrid algorithm for proof compression

4 Labeling in Interpolation

The PeRIPLO interpolation algorithm is based on computing the propositional interpolant from a refutation and a labeling which allows tuning of the interpolant to specific needs and the refutation. The implementation is based on the *labeled interpolation system* originally presented in [8] (LIS) and further developed in [1].

The system works on an *interpolation instance* (R, A, B), where R is the refutation of $A \wedge B$ and A is the formula to be over-approximated. Given a clause n in R and a variable $p \in vars(n)$ occurring in the clause, the system assigns a unique label $L(p, n)$ from the set $\{a, b, ab\}$ to the occurrence (p, n). In the leaf clauses the labeling is restricted to $L(p, n) = a$ if $p \notin vars(B)$ and $L(p, n) = b$ if $p \notin vars(A)$, but can be freely chosen for leaf occurrences of variables in $vars(A) \cap vars(B)$. In the resolvent clauses n_r of R the labeling $L(p, n_r)$ is determined by the label in n^+ and n^-. If $p \in vars(n^+) \cap vars(n^-)$ and $L(p, n^+) \neq l(p, n^-)$, then $L(p, n) = ab$, and in all other cases the label of the occurrence $L(p, n)$ is either $L(p, n^+)$ or $L(p, n^-)$.

The final interpolant is constructed based on the labeling and the refutation R iteratively for each clause in R starting from the leaf clauses and ending in the root. In particular, for a leaf clause n_l the interpolant is

$$I(n_l) = \begin{cases} \bigvee\{p \mid p \in n_l \text{ and } L(vars(p), n_l) = b\} & \text{if } n_l \in A\text{, and} \\ \bigwedge\{\neg p \mid p \in n_l \text{ and } L(vars(p), n_l) = a\} & \text{if } n_l \in B \end{cases} \tag{5}$$

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

$$I(n_r) = \begin{cases} I(n^+) \vee I(n^-) & \text{if } L(p, n^+) = L(p, n^-) = a, \\ I(n^+) \wedge I(n^-) & \text{if } L(p, n^+) = L(p, n^-) = b, \text{ and} \\ (I(n^+) \vee p) \wedge (I(n^-) \vee \neg p) & \text{otherwise.} \end{cases} \quad (6)$$

Several different approaches for constructing efficient labelings have been proposed. These include approaches for logically strong and weak interpolants [11,8], and our recent work on proof-sensitive labelings [1].

5 Experiments

We report here experimental results on both the proof compression approaches discussed in Sec. 3 and the labeled interpolation system of Sec. 4 using the PeRIPLO tool. Figure 5 shows the architecture of the tool. The experiments use a very basic form of proof compression where the algorithm in Fig. 4 uses the iteration count $I = 1$ and does not run *ReduceAndExpose* on line 4. The interpolator is used with six different interpolation algorithms: the weakest and the strongest interpolants M_w, M_s available from the LIS; three versions PS_w, PS, PS_s of a labeling function that attempt to minimize the interpolant size by labeling occurrences so that the minimum number of literals appear in the partial interpolants of the leaves; and P, an algorithm that labels all occurrences with ab. The experiments are done using the FunFrog system as the application. The system employs function summarization as explained in Sec. 2.

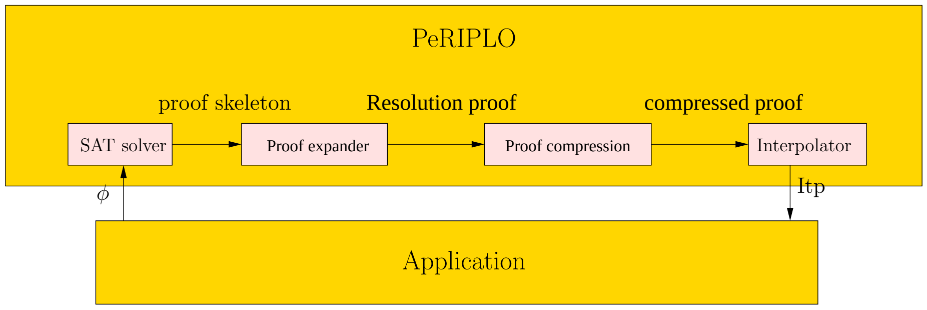

Fig. 5. The PeRIPLO architecture.

Table 1 reports the relation of size and time between the interpolants resulting from the algorithm without and with proof compression over roughly 25 interpolation instances. In general the proof compression helps in reducing both the size of the interpolant and the time required to construct the interpolant. The reduction in size is more significant than in run time. This is not unexpected

Table 1. Relative compression efficiency for different labeling functions.

	M_w	PS_w	P	PS	PS_s	M_s
time	3.25	3.10	2.77	2.19	2.10	2.20
size	15.15	15.86	12.04	7.20	6.57	6.13

since the run time contains several constant elements such as the time required to solve the instance. Interestingly the efficiency of the proof compression is not the same for all the interpolation algorithms. The algorithms M_w, PS_w, and P profit significantly more from the compressed refutation than the other algorithms. These algorithms often produce bigger interpolants, and there seems to be a non-trivial interaction between how the proof is reduced and how the different labeling functions are able to use the smaller proof.

We report the individual results as scatter plots in figures 6 and 7 for both time ($\times$) and size ($\square$). The results show a consistent reduction in all cases but also show several cases where the compression results in two orders of magnitude reduction in size.

6 Conclusions

This paper presents a range of applications in model checking where interpolation plays a critical role. We present two major techniques that affect the efficiency of interpolation: proof compression and labeling. Both techniques are described in detail showing how they are implemented in the propositional interpolation tool PeRIPLO. Finally we analyze the effect of proof compression when combined with different interpolant labellings on one of the applications. We reveal an interesting behavior that not all labeling functions profit in the same way from the proof compression. This suggests a non-trivial interaction between the interpolation and the proof compression that requires further studying.

Currently we are planning to extend the ideas presented in this paper to Satisfiability Modulo Theories in general, and applying them in other novel application domains where interpolation is useful.

Acknowledgements. This work has been supported by the SNF project numbers 200020_163001 and 200021_153402, and the ICT COST Action IC1402.

References

1. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A proof-sensitive approach for small propositional interpolants. In: Proc. VSTTE 2015 (2015), to appear

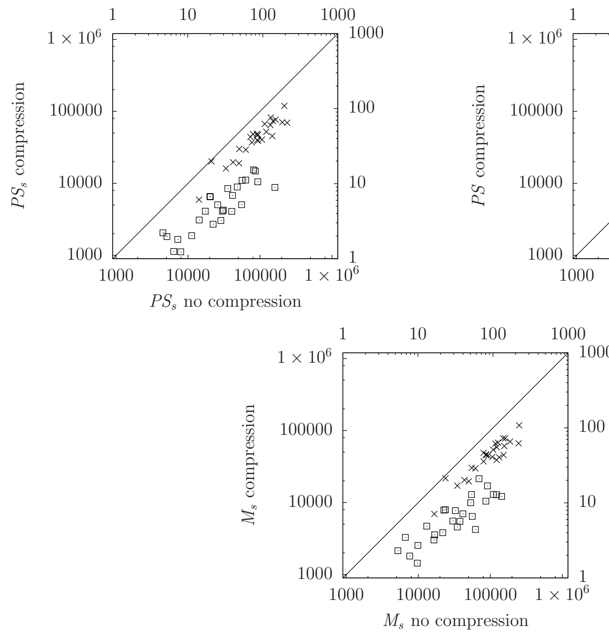

Fig. 6. Run time and interpolant sizes for the PS_s, PS, and M_s interpolation algorithms with and without proof compression. The left and bottom axes are in bytes and the top and right axes are in seconds. The $\times$ and $\square$ symbols are, respectively, the time and the size measurements.

2. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Proc. HVC 2008. LNCS, vol. 5394, pp. 114–128. Springer (2009)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer (1999)
4. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proc. Logic of Programs 1981. LNCS, vol. 131, pp. 52 – 71. Springer (1982)
5. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic 22(3), 269–285 (1957)
6. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7(3), 201–215 (1960)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
8. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Proc. VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer (2010)
9. Fedyukovich, G., D'Iddio, A.C., Hyvärinen, A.E.J., Sharygina, N.: Symbolic detection of assertion dependencies for bounded model checking. In: Egyed, A., Schaefer, I. (eds.) Proc. FASE 2015. LNCS, vol. 9033, pp. 186–201. Springer (2015)
10. Fontaine, P., Merz, S., Paleo, B.W.: Compression of propositional resolution proofs via partial regularization. In: Proc. CADE-23. pp. 237–251. LNCS, Springer (2011)

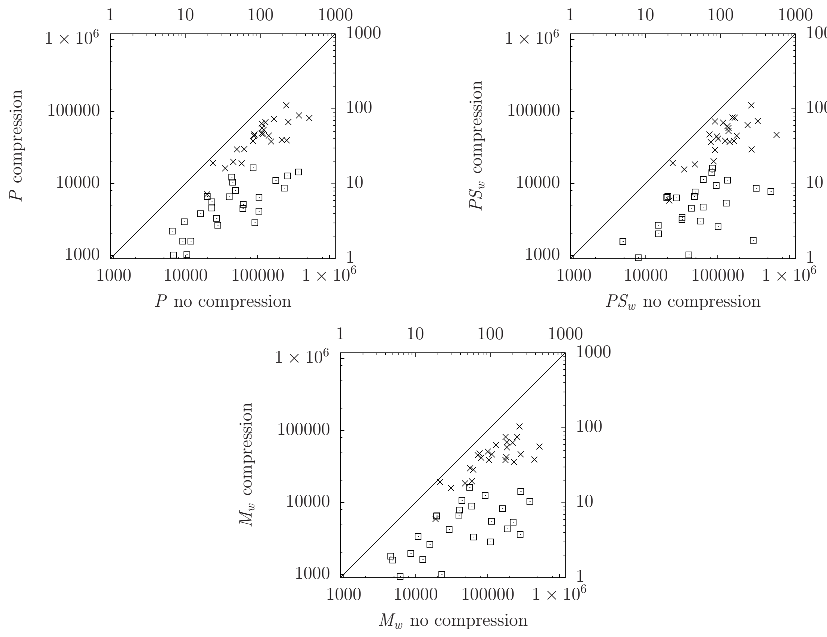

Fig. 7. Run time and interpolant sizes for the P, PS_w, and M_w interpolation algorithms with and without proof compression. The left and bottom axes are in bytes and the top and right axes are in seconds. The $\times$ and $\square$ symbols are, respectively, the time and the size measurements.

11. McMillan, K.L.: An interpolating theorem prover. In: Proc. TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer (2004)
12. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolant-based software verification. In: Proc. LPAR 2013. pp. 683–693. LNCS, Springer (2013)
13. Rollini, S.F., Bruttomesso, R., Sharygina, N., Tsitovich, A.: Resolution proof transformation for compression and interpolation. Formal Methods in System Design 45(1), 1–41 (2014)