Preprint from Proceedings of DATE 13, Grenoble, France, March 18-22, 2013

# Lemma Localization: A Practical Method for Downsizing SMT-Interpolants

Florian Pigorsch, and Christoph Scholl
University of Freiburg, Department of Computer Science, 79110 Freiburg im Breisgau, Germany
{pigorsch, scholl}@informatik.uni-freiburg.de

***Abstract*—Craig interpolation has become a powerful and universal tool in the formal verification domain, where it is used not only for Boolean systems, but also for timed systems, hybrid systems, and software programs. The latter systems demand interpolation for fragments of first-order logic. When it comes to model checking, the structural compactness of interpolants is necessary for efficient algorithms. In this paper, we present a method to reduce the size of interpolants derived from proofs of unsatisfiability produced by SMT (Satisfiability Modulo Theory) solvers. Our novel method uses structural arguments to modify the proof in a way, that the resulting interpolant is guaranteed to have smaller size. To show the effectiveness of our approach, we apply it to an extensive set of formulas from symbolic hybrid model checking.**

## I. Introduction

For mutually unsatisfiable formulas $A$ and $B$, a *Craig Interpolant* is a formula $I$, such that $I$ is implied by $A$, $I$ and $B$ are mutually unsatisfiable, and $I$ refers only to uninterpreted symbols common to $A$ and $B$. Or interpreted in the set-domain: $I$ is an over-approximation of $A$ which does not intersect with $B$. Interpolation has become – in addition to SAT- and SMT-solving – an universal workhorse in the domain of formal verification.

For purely *Boolean systems*, interpolation is used in many applications: McMillan introduced interpolation for the first time into the verification domain [1]. In [1] interpolants are used as overapproximations of reachable state sets; their use turns bounded model-checking into a complete method. The idea of [1] was picked up and refined by numerous researchers, e.g. in [2], [3], [4]. Logic synthesis for Boolean circuits [5] is another field of application.

Interpolants for several fragments of first-order logic have been successfully applied in *software verification* using predicate abstraction and refinement [6], [7], [8], [9], [10]. Moreover, for the verification of *hybrid systems* such interpolants have been used to optimize symbolic state set representations [11], [12]. Interpolants play another role in the verification of timed and hybrid systems, when bounded model checking for those systems [13], [14], [15] is combined with the ideas from [1], [16].

Especially when interpolants are used as symbolic state set predicates, not only their logical strength, but also their size has an essential impact on the efficiency of the overall verification algorithm. For instance, intensive compaction efforts to avoid exploding state set representations has turned out to be the key ingredient for approaches like [12].

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems”
(SFB/TR 14 AVACS, http://www.avacs.org/).



For Boolean systems, there are several methods in the literature that deal with the optimization of interpolants and proofs:

The authors of [17] check if intermediate clauses in the resolution proof are implied only by $A$ or $B$, and then treat these clauses as parts of $A$ or $B$. As a result, some parts of the proof can be ignored when computing an interpolant. In [18], the authors propose two methods to restructure a given resolution proof such that the proof becomes smaller; the proof minimization is motivated by the goals of smaller interpolants and smaller unsatisfiable cores. [19] introduces a method to detect and merge shared subproofs, [20] presents a set of local rewriting rules for proof graphs. The former two methods are justified by the need for small proofs in practical applications. In [21] and [22], proof transformations are discussed which influence the strength (in a logical sense) of interpolants; the structural compactness of interpolants is not considered. Moreover, [22] proposes a general interpolation system allowing for the production of interpolants of different logical strengths, again without considering compactness of interpolants.

In this paper we consider interpolants for first-order formulas. We present a novel method for reducing the structural size of interpolants for first-order formulas, which heavily relies on resolution proofs generated by SMT-solvers. We show how properties of the proofs can be exploited to avoid the computation of local interpolants for lemmata in the proof, resulting in interpolants of smaller size. Our method is based on enlarging theory lemmata which has the effect that local theory interpolants may be replaced by constants ‘true’ or ‘false’. At first sight this approach seems to be counterintuitive, since modern SMT($\mathcal{T}$)-solvers proceed exactly the other way round by *minimizing* theory lemmata in order to prune the search space in a DPLL-style search procedure. However, our method is used *after* the DPLL-style search as a postprocessing step to an existing proof. During postprocessing we always take care of preserving the complete structure and the validity of the original proof.

Interpolants produced by our prototype implementation on top of the MathSAT [23] SMT solver are significantly more compact than interpolants generated by a traditional interpolation algorithm, with only a slightly increased runtime.

This paper is structured as follows. In Sect. II we give a short introduction to SMT solving, resolutions proofs, and interpolation. Sect. III introduces our novel method for reducing the size of interpolants. This method is extensively evaluated in Sect. IV. Finally, we discuss further improvements and applications areas in Sect. V.

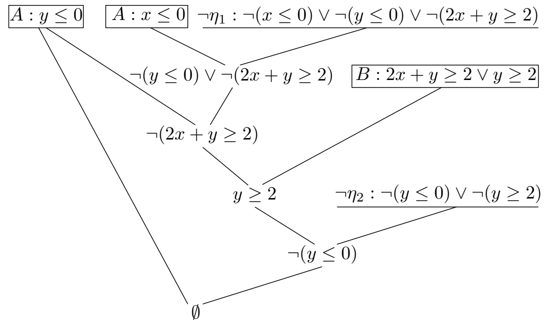


Fig. 1. A proof

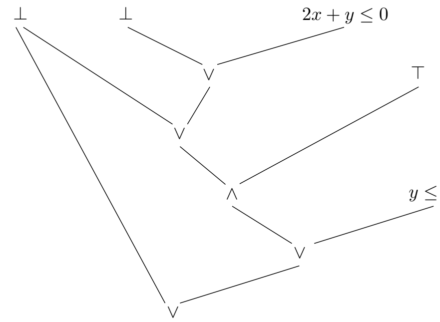


Fig. 2. An interpolant

## II. Preliminaries

A *signature* $\Sigma$ is a collection of function symbols and predicate symbols. A *theory* $\mathcal{T}$ gives interpretations to a subset of the symbols occuring in $\Sigma$. These symbols are called $\mathcal{T}$-symbols, symbols without interpretations are are called *uninterpreted*. A *term* is a first-order term built from the function symbols of $\Sigma$. For terms $t_1, \ldots, t_n$ and an $n$-ary predicate $p$, $p(t_1, \ldots, t_n)$ is an *atom*. A 0-ary atom is called proposition or *Boolean variable*. A (quantifier-free) *formula* is a Boolean combination of atoms.

A *literal* is either an atom or the negation of an atom. A literal built from an $n$-ary atom with $n > 0$ is called $\mathcal{T}$-literal. A *clause* is a disjunction of literals; for a clause $l_1 \vee \ldots \vee l_n$ we also use the set-notation $\{l_1, \ldots, l_n\}$. An empty clause, which is equivalent to $\bot$, is denoted with $\emptyset$. A clause, which contains a literal $l$ and its negation $\neg l$, is called *tautologic* clause, since it is equivalent to $\top$. In this paper we only consider *non-tautologic* clauses.

Let $C$ be a clause and $\phi$ be a formula. With $C \setminus \phi$, we denote the clause that is created from $C$ by removing all atoms occurring in $\phi$; $C \downarrow \phi$ denotes the clause that is created from $C$ by removing all atoms that are not occurring in $\phi$.

A formula is $\mathcal{T}$-satisfiable if it is satisfiable in $\mathcal{T}$, i.e., if there is a model for the formula where the $\mathcal{T}$-symbols are interpreted according to the theory $\mathcal{T}$. If a formula $\phi$ logically implies a formula $\psi$ in all models of $\mathcal{T}$, we write $\phi \models_{\mathcal{T}} \psi$.

*Satisfiability Modulo Theory* $\mathcal{T}$ (SMT($\mathcal{T}$)) is the problem of deciding the $\mathcal{T}$-satisfiability of a formula $\phi$.

Typical SMT($\mathcal{T}$)-solvers combine DPLL-style SAT-solving [24] with a separate decision procedure for reasoning on $\mathcal{T}$ [25]. Such a solver treats all atomic predicates in a formula $\phi$ as free Boolean variables. Once the DPLL-part of the solver finds a satisfying assignment, e. g. $l_1 \wedge \ldots \wedge l_n$, to this 'Boolean abstraction', it passes the atomic predicates corresponding to the assignment to a decision procedure for $\mathcal{T}$, which then checks whether the assignment is feasible when interpreted in the theory $\mathcal{T}$.[1] If the assignment is feasible, the solver terminates since a satisfying assignment to the formula $\phi$ has been found. If the assignment is infeasible in $\mathcal{T}$, the decision procedure derives a cause for the infeasibility of the assignment, say $\eta = m_1 \wedge \ldots \wedge m_k$, where $\{m_1, \ldots, m_k\} \subseteq \{l_1, \ldots, l_n\}$. We call the cause $\eta$ a $\mathcal{T}$*-conflict*, since $\eta \models_{\mathcal{T}} \bot$. The SMT($\mathcal{T}$)-solver then adds the negation of the cause, $\neg\eta = \{\neg m_1, \ldots, \neg m_k\}$, which we call $\mathcal{T}$*-lemma*, to its set of clauses and starts backtracking. The added $\mathcal{T}$-lemma prevents the DPLL-procedure from selecting the same invalid assignment again.

[1] This review describes the *lazy* SMT approach. Eager variants already check *partial* assignments to Boolean abstraction variables for consistency with the theory.

One can extend an SMT($\mathcal{T}$)-solver of this style in a straightforward way to produce proofs for the unsatisfiability of formulas [16], [26].

**Definition 1 ($\mathcal{T}$-Proof)** *Let $S = \{c_1, \ldots, c_n\}$ be a set of non-tautologic clauses and $C$ a clause.*

*A DAG $P$ is a* resolution proof *for the deduction of $\bigwedge c_i \models_{\mathcal{T}} C$, if*

1) *each leaf $n \in P$ is associated with a clause $n_{cl}$; $n_{cl}$ is either a clause of $S$ or a $\mathcal{T}$-lemma ($n_{cl} = \neg\eta$ for some $\mathcal{T}$-conflict $\eta$);*
2) *each non-leaf $n \in P$ has exactly two parents $n^L$ and $n^R$, and is associated with the clause $n_{cl}$ which is derived from $n^L_{cl}$ and $n^R_{cl}$ by resolution, i. e. the parents' clauses share a common variable (the* pivot*) $n_p$ such that $n_p \in n^L_{cl}$ and $\neg n_p \in n^R_{cl}$, and $n_{cl} = n^L_{cl} \setminus \{n_p\} \cup n^R_{cl} \setminus \{\neg n_p\}$; $n_{cl}$ (the* resolvent*) must not be a tautology;*
3) *there is exactly one root node $r \in P$; $r$ is associated with clause $C$; $r_{cl} = C$.*

Intuitively, a resolution proof provides a means to derive a clause $C$ from the set of clauses $S$ and some additional facts of the theory $\mathcal{T}$. If $C$ is the empty clause, $P$ is a proving the $\mathcal{T}$-unsatisfiability of $S$.

Fig. 1 shows a resolution proof for the unsatisfiability of $S = (x \le 0) \wedge (y \le 0) \wedge (2x + y \ge 2 \vee y \ge 2)$. To prove the unsatisfiability, the solver added two $\mathcal{T}$-lemmata $\neg\eta_1 = \neg(x \le 0) \vee \neg(y \le 0) \vee \neg(2x + y \ge 2)$ and $\neg\eta_2 = \neg(y \le 0) \vee \neg(y \ge 2)$.

**Definition 2 (Craig Interpolant)** *[27] Let $A$ and $B$ be two formulas, such that $A \wedge B \models_{\mathcal{T}} \bot$. A* Craig interpolant *$I$ is a formula such that*

1) *$A \models_{\mathcal{T}} I$,*
2) *$B \wedge I \models_{\mathcal{T}} \bot$,*
3) *the variables and uninterpreted symbols in $I$ occur both in $A$ and $B$.*

Given a $\mathcal{T}$-unsatisfiable set of clauses $S = \{c_1, \ldots, c_n\}$, $(A, B)$ a disjoint partition of $S$, and $P$ a proof for the $\mathcal{T}$-unsatisfiability of $S$, an interpolant for $(A, B)$ can be constructed by the following procedure [16]:

1) For every leaf $n \in P$ associated with a clause $n_{cl} \in S$, set $n_I = n_{cl} \downarrow B$ if $n_{cl} \in A$, and set $n_I = \top$ if $n_{cl} \in B$.
2) For every leaf $n \in P$ associated with a $\mathcal{T}$-lemma $\neg\eta$ ($n_{cl} = \neg\eta$), set $n_I = \mathcal{T}$-INTERPOLANT$(\eta \setminus B, \eta \downarrow B)$.

3) For every non-leaf node $n \in P$, set $n_I = n_I^L \vee n_I^R$ if $n_p \notin B$, and set $n_I = n_I^L \wedge n_I^R$ if $n_p \in B$.
4) Let $r \in P$ be the root node of $P$ associated with the empty clause $r_{cl} = \emptyset$. $r_I$ is the interpolant of $A$ and $B$.

Note that the interpolation procedure differs from pure Boolean interpolation [1] only in the handling of $\mathcal{T}$-lemmata. $\mathcal{T}$-INTERPOLANT$(A, B)$ produces an interpolant for an unsatisfiable pair of conjunctions of $\mathcal{T}$-literals. In [26], the authors list interpolation algorithms for several theories.

Fig. 2 shows a Craig interpolant resulting from the proof in Fig. 1, when partitioning $S$ into $(A, B)$ with $A = (x \leq 0) \wedge (y \leq 0)$ and $B = (2x + y \geq 2 \vee y \geq 2)$. Propagating constants, the result becomes $(2x + y \leq 0) \vee (y \leq 0)$. The $\mathcal{T}$-interpolant for the $\mathcal{T}$-conflict $\eta_1$ is a positive linear-combination of $\eta_1$'s A-literals[2], which is conflicting with a positive linear-combination of the remaining literals, e. g. $2 \cdot (x \leq 0) + 1 \cdot (y \leq 0) \equiv (2x + y \leq 0)$ is conflicting with $1 \cdot (2x + y \geq 2)$. The same holds for the $\mathcal{T}$-interpolant of $\eta_2$.

## III. Lemma Localization

We base our optimization technique on the following lemma:

**Lemma 1** *Let $\neg\eta$ be a $\mathcal{T}$-lemma and $(A, B)$ a pair of formulas;*

1) *if $\eta \setminus B$ is a $\mathcal{T}$-conflict, then $\perp$ is a valid interpolant for $(\eta \setminus B, \eta \downarrow B)$,*
2) *if $\eta \downarrow B$ is a $\mathcal{T}$-conflict, then $\top$ is a valid interpolant for $(\eta \setminus B, \eta \downarrow B)$.*

*In case (1) we call $\neg\eta$ an A-local lemma, in case (2) a B-local lemma, otherwise $\neg\eta$ is a non-local lemma.*

*Proof:*

- Assume $\eta \setminus B$ is a $\mathcal{T}$-conflict, i. e. $(\eta \setminus B) \models_{\mathcal{T}} \perp$, and $(\eta \downarrow B) \wedge \perp \models_{\mathcal{T}} \perp$. Furthermore, no uninterpreted symbols occur in $\perp$. So $\perp$ is a valid interpolant for $(\eta \setminus B, \eta \downarrow B)$.
- Assume $\eta \downarrow B$ is a $\mathcal{T}$-conflict, i. e. $(\eta \downarrow B) \models_{\mathcal{T}} \perp$. $(\eta \setminus B) \models_{\mathcal{T}} \top$ and $((\eta \downarrow B) \wedge \top) \models_{\mathcal{T}} (\perp \wedge \top) \models_{\mathcal{T}} \perp$. Furthermore, no uninterpreted symbols occur in $\top$. So $\top$ is a valid interpolant for $(\eta \setminus B, \eta \downarrow B)$.

■

Lemma 1 gives us a means to reduce the size of an interpolant for a given proof of unsatisfiability: assume a proof with a set of non-local $\mathcal{T}$-lemmata. When building an interpolant for this proof, each non-local lemma produces a possibly new $\mathcal{T}$-literal as a local interpolant (by $n_I := \mathcal{T} - \text{INTERPOLANT}(\eta \setminus B, \eta \downarrow B)$). If we had a way to add redundant $\mathcal{T}$-literals to these non-local lemmata, such that some lemmata became local and thus would have $\perp$ or $\top$ as interpolants instead of new $\mathcal{T}$-literals, the resulting interpolant would have a reduced size due to constant propagation.

At first sight, our method seems to contradict the standard approach used in SMT($\mathcal{T}$)-solvers which typically makes use of *minimal* infeasible assignments (and thus minimal $\mathcal{T}$-conflicts) produced by a theory solver in order to prune the search space of the DPLL-based solver as much as possible, leading to smaller proofs. However, we add redundant $\mathcal{T}$-literals to non-local lemmata only as a postprocessing step to an existing proof. In doing so we take care of preserving the validity of the original proof. By Fig. 3 we explain the basic idea: The upper part of Fig. 3 shows a detail of a bigger resolution proof $P$: the node $n$ has three children $n_1$, $n_2$, and $n_3$, whose clauses are derived from their parents by resolution. The clauses of all three children contain the literal $c$. $n$'s clause, however, does not contain the literal $c$. 'Pushing up' $c$ from $n_1$, $n_2$, and $n_3$ to $n$ (i. e. adding $c$ to $n$'s clause), creates a modified graph $P'$ (shown in the lower part of Fig. 3), in which the resolutions at $n$'s children $n_1$, $n_2$, and $n_3$ are still valid. If, on the other hand, one child's clause did not contain the literal $c$ (e. g. $n_3$'s clause), and we pushed up $c$ to $n$'s clause, the resolution at this child would not be valid anymore (the resolvent would then contain $c$ but the original clauses did not). Furthermore, one must not push a literal into a node's clause if the literal matches the node's pivot, since the resolvent of two clauses can not contain the pivot variable.

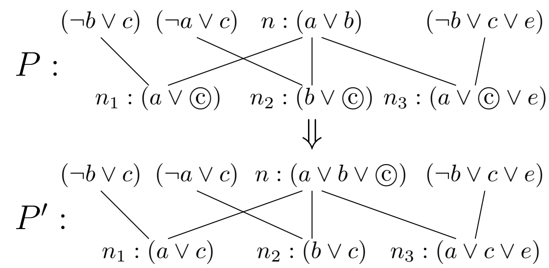


Fig. 3. Pushing-up literals

In general, one can 'push up' literals to a node's clause that are (1) in the intersection of all of its children's clauses, and (2) do not match the node's pivot.

This procedure can be applied recursively to $n$'s parents until the leaves of the proof are reached.

Alg. 1 gives an implementation of this idea.

**Algorithm 1:** PushUp

**Input** : Proof $P$
**Output**: Modified proof
`// step 1: push up` $\mathcal{T}$`-literals`
**for** $n \in P$ *in reverse topological order* **do**
  **if** $n$ **is** *a non-leaf* **or** $n_{cl}$ **is** *a $\mathcal{T}$-lemma* **then**
    $L := \bigcap_{n' \in children(n)} n'_{cl}$;
    $L := L \cap \{l | l \text{ is a } \mathcal{T}\text{-literal}\}$;
    **if** $n$ **is** *a non-leaf* **then** $L := L \setminus \{n_p, \neg n_p\}$;
    $n_{cl} := n_{cl} \cup L$
`// step 2: rebuild resolutions`
**for** $n \in P$ *in topological order* **do**
  **if** $n$ **is** *a non-leaf* **then** $n_{cl} := res(n^L_{cl}, n^R_{cl})$
**return** $P$;

The algorithm traverses the proof in reverse topological order starting at the root node and finally ending at the leaf nodes. At a non-leaf node $n$ the procedure computes the intersection $L$ of $\mathcal{T}$-literals in the clauses of $n$'s children (which have been processed before), removes $n$'s pivot variable from $L$, and inserts the resulting set to $n$'s clause. For leaves associated with $\mathcal{T}$-lemmata, the same procedure is applied, except for the removal of pivot variables. Leaves not associated with $\mathcal{T}$-lemmata are not modified.

[2]Note that in the case of linear inequations an appropriate linear-combination can be computed by linear programming [28].

At this stage the proof may no longer be a legal resolution proof, since some literals that have been pushed-up in the proof may have 'got stuck' and thus did not reach any leaves.

To turn the proof into a legal proof again, we reconstruct the resolutions by traversing the graph in topological order starting at the leaves and recomputing the non-leaves' clauses by resolution using the original pivots.

We now show in a two-step proof, that the *PushUp*-operation, given a valid resolution proof for the $\mathcal{T}$-unsatisfiability of a clause set $S$, produces a valid resolution proof.

**Lemma 2** *Let $S = \{c_1, \ldots, c_n\}$ be a set of clauses, $P$ be a resolution proof of the $\mathcal{T}$-unsatisfiability of $S$, and let $P'$ be the result of Step 1 of the* PushUp*-operation on $P$.*

*For node $n' \in P'$ and the corresponding node $n \in P$, it holds that $n'_{cl} \geq n_{cl}$ and $n'_{cl}$ is not a tautology.*

*Proof:* The first part ($n'_{cl} \geq n_{cl}$) is obvious, since the operation only adds new literals to a clause but does not remove any literals. The second part can be proven by induction over the node's distance to the root node.

In the base case the distance is 0, and thus $n' \in P'$ and $n \in P$ are the root nodes of $P'$ and $P$. $P$ is a proof of unsatisfiability, so $n_{cl}$ is the empty clause by definition. No literals are added to $n_{cl}$ ($n$ has no children), resulting in $n'_{cl} = \emptyset$, which is clearly not a tautology.

In the inductive step, we are looking at a node $n' \in P'$ with at least one child. We assume that $n'_{cl}$ is a tautology, i. e. there is a literal $l$, with $l \in n'_{cl}$ and $\neg l \in n'_{cl}$. Since all clauses occurring in $P$ are either clauses of $S$, $\mathcal{T}$-lemmata, or produced by resolution, they are all non-tautological. For the corresponding node $n \in P$ it holds that $l \notin n_{cl}$ or $\neg l \notin n_{cl}$.

*Case 1*: $l \notin n_{cl}$ and $\neg l \notin n_{cl}$. $l$ and $\neg l$ must have been added to $n'_{cl}$ by the *PushUp*-operation, i. e. for all $c' \in children(n')$ it must hold that $l \in c'_{cl}$, $\neg l \in c'_{cl}$, which is a contradiction to the induction hypothesis.

*Case 2*: $l \notin n_{cl}$ and $\neg l \in n_{cl}$. $l$ must have been added to $n'_{cl}$ by the *PushUp*-operation, i. e. for all children $c' \in children(n')$ it must hold that $l \in c'_{cl}$ (*). $l \in c'_{cl}$ implies that $l \in c_{cl}$ or $l$ is added to $c_{cl}$ by the *PushUp*-operation. In both cases it follows $c_p \neq l$. For all children $c \in children(n)$ $\neg l \in n_{cl}$ and $c_p \neq l$ implies that $\neg l \in c_{cl}$ which in turn implies $\neg l \in c'_{cl}$ (**). (*) and (**) contradict the induction hypothesis.

*Case 3*: $l \in n_{cl}$ and $\neg l \notin n_{cl}$. Similar to Case 2. ∎

This lemma states that the same resolutions are possible in $P$ and the result $P'$ of Step 1 of the *PushUp*-operation on $P$. This allows for a successful rebuilding of the resolutions by Step 2 of the *PushUp*-operation.

**Theorem 1** *Let $S = \{c_1, \ldots, c_n\}$ be a set of clauses, $P$ be a resolution proof of the $\mathcal{T}$-unsatisfiability of $S$, and let $P'$ be the result of the* PushUp*-operation on $P$.*

*$P'$ is a resolution proof of the $\mathcal{T}$-unsatisfiability of $S$.*

*Proof:*

1) Let $n'$ be a leaf node of $P'$ and $n$ be the corresponding leaf node of $P$. Case (a) $n_{cl} \in S$: $n_{cl}$ is not modified by *PushUp* and thus $n'_{cl} \in S$. Case (b) $n_{cl} = \neg\eta$ for some $\mathcal{T}$-lemma $\neg\eta = \{l_1, \ldots, l_k\}$: The clause $n'_{cl}$ of the corresponding node $n' \in P'$ is created from $n_{cl}$ by adding some set of $\mathcal{T}$-literals $\{l'_1, \ldots, l'_m\}$. Since $l_1 \vee \ldots \vee l_k \models_{\mathcal{T}} \top$ (definition of a $\mathcal{T}$-lemma), it holds that $l_1 \vee \ldots \vee l_k \vee l'_1 \vee \ldots \vee l'_m \models_{\mathcal{T}} \top$. So $n'_{cl}$ is a $\mathcal{T}$-lemma, too.
2) Let $n'$ be a non-leaf node of $P'$. By construction it holds that $n'_{cl} = res(n'^{l}_{cl}, n'^{r}_{cl})$.
3) Let $r'$ be the root node of $P'$. Assume that $r'_{cl}$ is not the empty clause, w. l. o. g. $l \in r'_{cl}$ for some $\mathcal{T}$-literal $l$. $l$ must have been added to some $\mathcal{T}$-lemma of a leaf node $n$ in the first part of $PushUp$. It is easy to see from Alg. 1 that this can only happen if every path $\pi$ from the root node $r \in P$ to the leaf $n$ contains a non-leaf node $m$ with $l \in m_{cl}$; since $r_{cl}$ is the empty clause in the original proof, $l$ must vanish by resolution with pivot $l$ on the sub-path $\pi'$ from $r$ to $m$. All pivots remain unchanged in the modified proof, so $l$ is eliminated by resolution on any path to $r'$. $l \notin r'_{cl}$, which is a contradiction to the assumption.

∎

Using the *PushUp*-operation and Lemma 1, we define a modified interpolation algorithm (Alg. 2), which we call *interpolation with lemma localization*.

**Algorithm 2:** Interpolation with Lemma Localization

**Input** : Set of clauses $S$ with $S \models_{\mathcal{T}} \bot$ and disjoint partition $(A, B)$ of S
**Output**: Interpolant for $(A, B)$
1 $P$ := `ComputeProof(`$A$, $B$`)`;
2 $P'$ := `PushUp(`$P$`)`;
3 **for** *leaf* $n \in P'$ **with** $n_{cl} \in S$ **do**
4   **if** $n_{cl} \in A$ **then** $n_I$ := $n_{cl} \downarrow B$;
5   **else** $n_I$ := $\top$;
6 **for** *leaf* $n \in P'$ **with** $n_{cl} = \neg\eta$ **and** $\neg\eta$ **is** $\mathcal{T}$*-lem.* **do**
7   **if** $\eta \setminus B$ **is** $\mathcal{T}$*-unsat.* **then** $n_I$ := $\bot$;
8   **else if** $\eta \downarrow B$ **is** $\mathcal{T}$*-unsat.* **then** $n_I$ := $\top$;
9   **else** $n_I$ := $\mathcal{T}$-INTERPOLANT$(\eta \setminus B, \eta \downarrow B)$;
10 **for** *non-leaf* $n \in P'$ *in topolocical order* **do**
11   **if** $n_p \notin B$ **then** $n_I$ := $n_I^L \vee n_I^R$;
12   **else** $n_I$ := $n_I^L \wedge n_I^R$;
13 **return** $r_I$, **where** $r$ **is** *the root node of* $P'$;

The algorithm differs from the original interpolation algorithm in two places: In line 2 the *PushUp*-operation is called to insert $\mathcal{T}$-literals into the $\mathcal{T}$-lemmata of the proof's leaves. In lines 7–8 Lemma 1 is applied to discover local lemmata; if a local lemma is found, the interpolant is set to $\top$ or $\bot$, otherwise the algorithm falls back to standard $\mathcal{T}$-interpolation.

The interpolant computed by Alg. 2 differs from the original interpolant constructed by the procedure from [16] only by the fact that some interpolants for unsatisfiable pairs of conjunctions of $\mathcal{T}$-literals are replaced by constants $\top$ or $\bot$. Apart from that the interpolants are structurally equivalent. Our interpolant can be simplified by constant propagation such that our interpolant is guaranteed to have smaller or equal size as the original interpolant.

## IV. Experimental Results

To show the effectiveness of our approach, we have implemented a prototype for the interpolation of formulas over $\mathcal{LA}(\mathbb{Q})$ (linear inequations over the rationals with rational

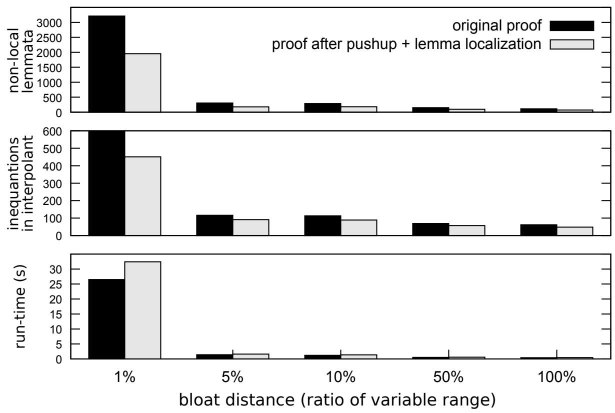


Fig. 5. Results of interpolation

coefficients) closely following Alg. 2. The initial proof of unsatisfiability for a given formula is computed using MathSAT's proof API[3] [23], the checks for A/B-locality of lemmata (lines 8 and 9 in the algorithm) are conducted with MathSAT[4], the interpolants for non-local lemmata (line 10 of the algorithm) are taken from the initial proof.

As benchmarks for our implementation we chose intermediate state sets produced by the symbolic model-checker FOMC [12] for hybrid systems; these state sets are represented with LinAIGs, which basically are arbitrary Boolean combinations of Boolean variables and linear inequations over $\mathbb{Q}$, and thus are formulas over $\mathcal{LA}(\mathbb{Q})$. Given such a state set $\phi$, we produce a bloated version $\phi' = \text{BLOAT}(\phi, \epsilon)$ by pushing all inequations 'outwards' by a positive distance $\epsilon$. Fig. 4 sketches a 2-dimensional state set $\phi$ with its bloating $\phi'$. An interpolant of $(\phi, \neg\phi')$ is an over-approximation of $\phi$ which does not deviate from $\phi$ by more than the distance $\epsilon$. Increasing the distance $\epsilon$ gives the interpolation algorithm more freedom, and thus may produce an interpolant with a smaller representation, but on the other hand increases the over-approximation. Given a state set and a fixed $\epsilon$, the goal is to find an interpolant whose representation is as small as possible.

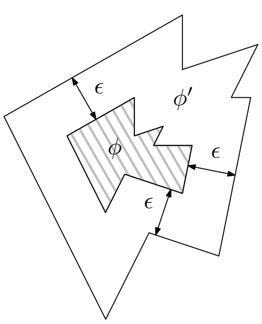


Fig. 4. Bloated state set

We have taken several intermediate state sets from model-checker runs on different models (a total of 188 state sets). In these models a variable range for each rational variable is given. We applied bloating to the state sets with distances $\epsilon$ corresponding to 1%, 5%, 10%, 50% and 100% of the variable ranges (resulting in a total amount of 940 benchmarks). The original state sets contain up to 7 rational variables, up to 591 inequations, up to 28 additional Boolean variables, and up to 18000 binary Boolean connectives.

All experiments were conducted on one core of an Intel Xeon machine with 3.0GHz and a memory limit of 4GB RAM.

We first take a look at the largest state set. Fig. 5 compares the original interpolation method (black bars) and our novel technique (light grey bars) described in Alg. 2 for several bloating distances $\epsilon$. The upper chart shows the number of non-local lemmata in the original proof and in the modified proof produced by Alg. 2. For the $1\%$ bloating, the number of non-local lemmata drops from $3212$ in the original proof to $1959$ in the modified proof - about $39\%$ of the non-local lemmata in the proof become local due to pushing literals from the resolution proof into the lemmata. As an effect the number of inequations in the computed interpolant drops from $599$ to $451$ (in both cases after constant propagation). This corresponds to a decrease of about $24\%$. This comes at a cost: the total computation time (including generation of the original proof and building the final interpolant) slightly rises from $26.5$ to $32.4$ seconds (increase by $22\%$); the additional time is spent for the *PushUp*-operation and the additional locality checks for all lemmata. Moreover, Fig. 5 shows for increasing bloating distances that all numbers (non-local lemmata, inequations in interpolants, run-times) decrease due to increased degrees of freedom for the interpolation.

We now use the complete benchmark set to quantitatively compare the original interpolation scheme with our modified technique.

Fig. 6 shows scatter charts depicting the numbers of inequations in the resulting interpolants for the original interpolation scheme (x-axis) and the modified scheme (y-axis) – one for those formulas where the number of inequations in the original interpolant is $< 100$ and one for $\geq 100$. A point in one of the charts corresponds to one state set with a specific $\epsilon$[5]; the x-value of such a point gives the number of inequations in the resulting interpolant for the original scheme, the y-value gives the number of inequations for the modified scheme. If a point lies below the black diagonal line, the modified method resulted in a lower number of inequations.[6]

In $65\%$ of the benchmarks we can observe a reduction in the number of inequations by $\leq 10\%$ due to our proposed method, for 23% of the benchmarks the improvement is $> 10\%$ and $\leq 20\%$, for 6% of the benchmarks the improvement is $> 20\%$ and $\leq 30\%$, for the remaining 6% of the benchmarks the number of inequations is reduced by more than 30%. Due to the nature of our method, the number of inequations can never increase.

Fig. 7 shows a similar scatter chart for the total run-time (including the computation of the initial proof, performing pushup and lemma localization, and building the final interpolant). [7]

As expected, the time consumption of our proposed method is slightly larger that the original method (the points lie above the black diagonal) due to the extra work done by the *PushUp*-operation and lemma localization. For 38% of the benchmarks the increase in runtime is $\leq 10\%$, for 55% the increase is $>$ $10\%$ and $\leq 20\%$, for the remaining 7% of the benchmarks the increase in run-time is $> 20\%$ and $\leq 30\%$; we never observed an increase in run-time by more than 30%[8].

A break down of algorithm 2 reveals that the additional time needed for interpolation when using lemma localization on average divides about in half into the the *PushUp*-operation and the detection of local lemmata via SMT-calls.

[3] MathSAT 5, version 5.1.10

[4] For the sake of simplicity we are using a full-blown SMT solver, although an LP solver would suffice for performing the necessary checks.

[5] Due to chart scaling issues we omit the five data points corresponding to the state set already discussed in Fig. 5

[6] The gray lines correspond to 10%, 20%, and 30% improvement.

[7] The gray lines correspond to 10%, 20%, and 30% increase in run-time.

[8] Since small run-times are relatively unstable, we restrict the previous evaluation to benchmarks with relevant run-times $\geq 0.5s$.

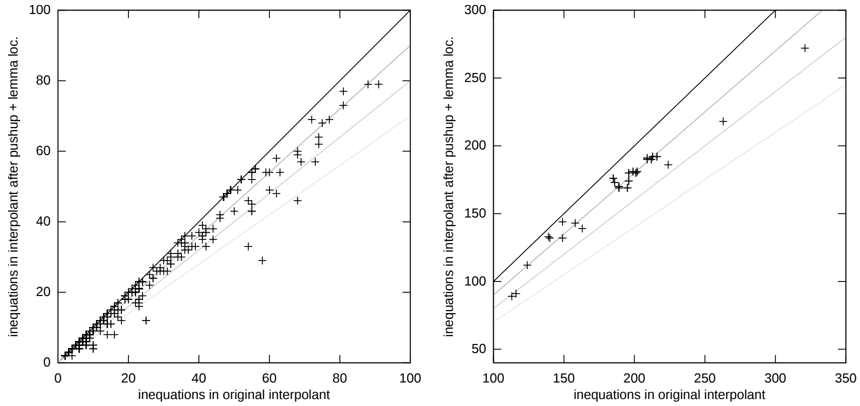


Fig. 6. All benchmarks, inequations in resulting interpolant

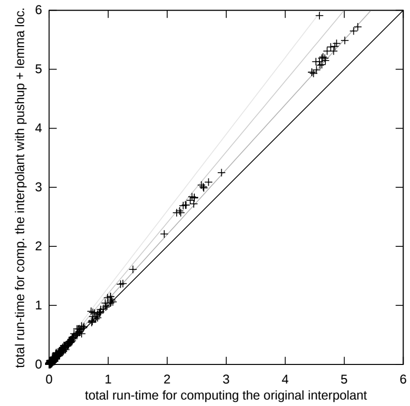


Fig. 7. All benchmarks, total run-time

Altogether, we can observe significant improvements by *interpolation with lemma localization* on the benchmark set. These improvements could be achieved without relevant run-time penalties.

## V. Conclusions

In the previous section we have presented *interpolation with lemma localization*, a modified interpolation algorithm for formulas over $\mathcal{T}$, which modifies a given resolution proof by pushing up $\mathcal{T}$-literals to the $\mathcal{T}$-lemmata, without actually changing the structure and the validity of the proof. Extending the $\mathcal{T}$-lemmata by $\mathcal{T}$-literals possibly creates *local* lemmata with trivial interpolants, and thus reduces the size of the final interpolant. We have also shown the effectiveness of this approach on a large set of real-world benchmarks from hybrid model-checking.

Our approach seems to be completely orthogonal to purely propositional approaches for proof restructuring and interpolant compaction, such as [17], [18], [19], [20], as well as to other methods which optimize already existing state set predicates, such as [11]. It may be combined with these techniques yielding even smaller interpolants.

We have shown the effectiveness of our method on single state sets from a hybrid model checker. For the future we plan to provide an extended hybrid model checker using over-approximated state sets based on the computed interpolants (in combination with a counterexample-guided refinement procedure).

## References


[1] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV*, 2003, vol. 2742, pp. 1–13.

[2] J. Marques-Silva, "Improvements to the Implementation of Interpolant-Based Model Checking," in *Proc. of CHARME*, 2005, pp. 367–370.

[3] G. Cabodi, M. Murciano, S. Nocco, and S. Quer, "Stepping forward with interpolants in unbounded model checking," in *Proc. of ICCAD*, 2006, pp. 772–778.

[4] B. Li and F. Somenzi, "Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking," in *Proc. of TACAS*, 2006, pp. 227–241.

[5] C.-C. Lee, J.-H. R. Jiang, C.-Y. R. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," in *Proc. of ICCAD*, 2007, pp. 227–233.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from Proofs," in *Proc. of POPL*, 2004, pp. 232–244.

[7] K. L. McMillan, "Lazy abstraction with interpolants," in *Proc. of CAV*, 2006, pp. 123–136.

[8] I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim, "Slicing Abstractions," in *Proc. of FSEN*, 2007.

[9] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos - A Software Model Checker for SystemC," in *Proc. of CAV*, 2011, pp. 310–316.

[10] D. Kroening and G. Weissenbacher, "Interpolation-Based Software Verification with Wolverine," in *Proc. of CAV*, 2011, pp. 573–578.

[11] C. Scholl, S. Disch, F. Pigorsch, and S. Kupferschmid, "Computing Optimized Representations for Non-convex Polyhedra by Detection and Removal of Redundant Linear Constraints," in *Proc. of TACAS*, 2009, pp. 383–397.

[12] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz, "Exact and Fully Symbolic Verification of Linear Hybrid Automata with Large Discrete State Spaces," *Science of Computer Programming*, vol. 77, no. 10-11, pp. 1122–1150, 2012.

[13] G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "Bounded Model Checking for Timed Systems," in *FORTE*, 2002, pp. 243–259.

[14] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani, "Verifying Industrial Hybrid Systems with MathSAT," *Electr. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 17–32, 2005.

[15] M. Fränzle and C. Herde, "HySAT: An efficient proof engine for bounded model checking of hybrid systems," *Formal Methods in System Design*, vol. 30, no. 3, pp. 179–198, 2007.

[16] K. L. McMillan, "An Interpolating Theorem Prover," *Theoretical Computer Science*, vol. 345, no. 1, pp. 101 – 121, 2005.

[17] J. D. Backes and M. D. Riedel, "Reduction of Interpolants for Logic Synthesis," in *Proc. of ICCAD*, 2010, pp. 602–609.

[18] O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman, "Linear-Time Reductions of Resolution Proofs," in *Proc. of HVC*, 2009, pp. 114–128.

[19] C. Sinz, "Compressing Propositional Proofs by Common Subproof Extraction," in *Proc. of EUROCAST*, 2007, pp. 547–555.

[20] S. F. Rollini, R. Bruttomesso, and N. Sharygina, "An Efficient and Flexible Approach to Resolution Proof Reduction," in *Proc. of HVC*, 2011, pp. 182–196.

[21] R. Jhala and K. L. McMillan, "Interpolant-Based Transition Relation Approximation," *Logical Methods in Computer Science*, vol. 3, no. 4, 2007.

[22] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant Strength," in *Proc. of VMCAI*, 2010, pp. 129–145.

[23] A. Griggio, "A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic," *JSAT*, vol. 8, pp. 1–27, January 2012.

[24] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[25] L. de Moura, H. Ruess, and M. Sorea, "Lazy Theorem Proving for Bounded Model Checking over Infinite Domains," in *Proc. of CADE*, 2002, pp. 1–4.

[26] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories," *ACM Trans. Comput. Logic*, vol. 12, no. 1, pp. 7:1–7:54, Nov. 2010.

[27] W. Craig, "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory," *Journal of Symbolic Logic*, vol. 22, no. 3, pp. pp. 269–285, 1957.

[28] A. Rybalchenko and V. Sofronie-Stokkermans, "Constraint Solving for Interpolation," in *Proc. of VMCAI*, 2007, pp. 346–362.