Tree Interpolation in Vampire*

Régis Blanc[1], Ashutosh Gupta[2], Laura Kovács[3], and Bernhard Kragl[4]

[1]EPFL [2] IST Austria [3]Chalmers [4]TU Vienna

Abstract. We describe new extensions of the Vampire theorem prover for computing tree interpolants. These extensions generalize Craig interpolation in Vampire, and can also be used to derive sequence interpolants. We evaluated our implementation on a large number of examples over the theory of linear integer arithmetic and integer-indexed arrays, with and without quantifiers. When compared to other methods, our experiments show that some examples could only be solved by our implementation.

1 Introduction

In interpolation-based verification approaches, a Craig interpolant [3] is a logical formula justifying why a program trace is spurious and therefore can be used, for example, to refine the set of predicates for predicate abstraction [10], invariant generation [13], and correctness proofs of programs [8]. As refining a path in the control flow graph of a program requires iterative computations of interpolants for each path location, Craig interpolants have been generalized to sequence interpolants for their use in bounded model checking non-procedural programs [10]. Using sequence interpolants to reason about programs with recursive procedures is however a non-trivial task. The work of [12] introduces the notion of tree interpolants, which can be used for the verification of concurrent [5] and recursive programs [8]. In this context, dependencies between program paths are encoded using a tree data structure, where a tree node represents a formula valid at an intermediate program location. Tree interpolants provide a nested structure for representing formulas, and therefore allow to reason about programs with function/procedure calls.

Similarly to Craig interpolation, the key ingredient in theorem proving based tree interpolation is the computation of special proofs, for example local or split proofs [10,9], with feasible interpolation. Interpolants from such proofs can be constructed in polynomial time in the size of the proof. Current approaches for building Craig/sequence/tree interpolants depend on the existence of such proofs. For example, [14] uses SMT reasoning to derive Craig interpolants in the quantifier-free theory of linear arithmetic and uninterpreted functions. This approach is further generalized in [7] for computing tree interpolants from propositional proofs and in [2,12,5] to derive tree interpolants in the theory of linear arithmetic and uninterpreted functions. Contrary to the above techniques, in [9] Craig interpolants are extracted from first-order local proofs in any sound

* This research was partly supported by the Austrian National Research Network RiSE (FWF grants S11402-N23 and S11410-N23) and the WWTF PROSEED grant (ICT C-050).

calculus, without being limited to decidable theories. However, the method of [9] cannot yet be used for deriving tree and sequence interpolants.

In this paper we address the generality of [9] and describe a tool support for extracting tree interpolants in arbitrary first-order theories (Section 4). Our method is implemented in the Vampire theorem prover [11] and extends Vampire with new features for theory reasoning and interpolation. Our implementation adds a general interpolation procedure to Vampire, which can be used for computing Craig interpolants, sequence interpolants and tree interpolants. For doing so, we reduce the problem of tree interpolations to iterative applications of Craig interpolants on tree nodes (Section 3). Our approach is different from [2,7] where tree interpolants are extracted from only one proof, by exploiting propositional reasoning or linear arithmetic properties. Our tool can be used in arbitrary theories and calculus, but comes at the cost of computing different proofs for each tree node. Our implementation can however be optimized when considering specific theories, reducing the burden of iterative proof computations. We tested our tool on challenging examples over arrays, involving reasoning with both quantifiers and theories (Section 5). To the best of our knowledge, our tool is the only approach able to derive tree interpolants with both quantifiers and theory symbols. We also evaluated our implementation on examples coming from the model checking of device drivers, where quantifier-free reasoning over linear integer arithmetic and integer-indexed arrays was required. On these examples our method does not perform as well as theory-specific approaches, e.g. [14]. The strength of our tool comes thus when tree interpolants in full first-order theories are needed. Extending our implementation with proof transformations for various theories is an interesting task for future work.

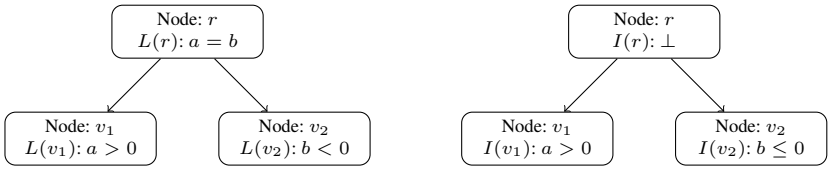
2 Tree Interpolation

All formulas in this paper are first-order, with standard boolean connectives and quantifiers. The *language* of a formula R, denoted by $\mathcal{L}_R$, is the set of all formulas built from the symbols occurring in R. By a symbol we mean function and predicate symbols; variables are not symbols. Given two formulas R and B such that $R \wedge B$ is unsatisfiable, a formula $I_{R,B}$ is called a *Craig interpolant of R and B* (or simply just an *interpolant*) iff $R \rightarrow I_{R,B}$, $I_{R,B} \wedge B$ is unsatisfiable, and $I_{R,B}$ contains only symbols that occur both in the languages of R and B. A proof of unsatisfiability of $R \wedge B$ is called *local* [9] if every proof step uses symbols either only from R, or only from B.

We describe the problem of tree interpolation, by adapting the notation of [1,12].

Definition 1. *A* tree interpolation problem $T = (V, r, P, L)$ *is a directed labeled tree, where V is a finite set of nodes, $r \in V$ is the root, $P : (V \setminus \{r\}) \mapsto V$ is a function that maps children nodes to their parents, and $L : V \mapsto \mathbb{F}$ is a labeling function that maps nodes to formulas from a set $\mathbb{F}$ of first-order formulas, such that $\bigwedge_{v \in V} L(v)$ is unsatisfiable.*

Let $T = (V, r, P, L)$ be a tree interpolation problem and P^* be the reflexive transitive closure of P. For $V_0 \subseteq V$ we write $L(V_0)$ to denote the formula $\bigwedge_{v \in V_0} L(v)$. For each $v \in V$ we define $V_{in}(v) = \{c \mid v \in P^*(c)\}$ and $V_{out}(v) = V \setminus V_{in}(v)$. The *problem of tree interpolation* is then to compute a tree interpolant, defined as follows.



(a) Tree interpolation problem. (b) Tree interpolant.

Fig. 1. An example of tree interpolation; a and b are integer-valued constants.

Definition 2. *Let $T = (V, r, P, L)$ be a tree interpolation problem. A* tree interpolant *for T is a function $I : V \mapsto \mathbb{F}$ satisfying the following conditions:*

(C1) $I(r) = \bot$*;*
(C2) for each $v \in V$*, we have:* $\big(\bigwedge_{P(c_i)=v} I(c_i) \wedge L(v)\big) \rightarrow I(v)$*;*
(C3) for each $v \in V$*, we have:* $\mathcal{L}_{I(v)} \subseteq \mathcal{L}_{L(V_{in}(v))} \cap \mathcal{L}_{L(V_{out}(v))}$*.*

In the following, we refer to $I(v)$ as a *node interpolant*, or simply just an *interpolant*, of node v. Figure 1(a) gives an example of a tree interpolation problem, and Figure 1(b) shows a correspoding tree interpolant.

3 Tree Interpolation Algorithm

When computing a tree interpolant for a tree interpolation problem T, we need to establish conditions (C1)-(C3) from Definition 2. Since $L(V_{in}(v)) \wedge L(V_{out}(v))$ is unsatisfiable for each $v \in V$, we can compute an interpolant between $L(V_{in}(v))$ and $L(V_{out}(v))$. However, computing all node interpolants $I(v)$ this way may violate condition (C2), as illustrated in Example 1.

Example 1. Consider the tree interpolation problem from Figure 1(a). We compute $I(v_1)$ as an interpolant between $L(v_1)$ and $L(v_2) \wedge L(r)$, and $I(v_2)$ as an interpolant between $L(v_2)$ and $L(v_1) \wedge L(r)$. In this example, we may take $I(v_1) = (a \geq 0)$ and $I(v_2) = (b \leq 0)$. By definition, $I(r) = \bot$. But then $I(v_1) \wedge I(v_2) \wedge L(r)$ is satisfiable, and hence $I(v_1) \wedge I(v_2) \wedge L(r) \rightarrow I(r)$ does not hold.

Example 1 shows that node interpolants are logically weaker than node labels. Already computed node interpolants have to be taken into account for computing further node interpolants. Our tree interpolation algorithm is based on this observation and summarized in Algorithm 1.

In line 4 all node interpolants are initialized to ∞, representing undefined. A node interpolant in our algorithm is thus either undefined or a first-order formula. Then we iterate over all nodes of T according to the loop condition in line 6. That is, we always choose an arbitray node v with undefined interpolant, such that the interpolants of its children have already been computed. In lines 7-8 the tree nodes are partitioned into $V_{in}(v)$ and $V_{out}(v)$, which are used to obtain the formulas R_v and B_v, by taking the conjunction of node labels from root to leaves up to the first defined node interpolant (see Algorithm 2). Then $I(v)$ is set to a Craig interpolant of R_v and B_v (line 9). Using

Algorithm 1 Tree Interpolation.

Input: Tree interpolation problem $T = (V, r, P, L)$
Output: Tree interpolant I of T
for each $v \in V$ **do**
    $I(v) = \infty$
end for
for each $v \in V$ such that $I(v) = \infty$ and $I(c) \neq \infty$ for each $c \in V$ with $v = P(c)$ **do**
    $R_v = S(V_{in}(v), v)$ (call to Alg. 2)
    $B_v = S(V_{out}(v), r)$ (call to Alg. 2)
    $I(v) = CraigInterpolant(R_v, B_v)$
end for

Algorithm 2 Interpolant/Label Collection.

Input: Set of tree nodes $V_0 \subseteq V$ and a node $v \in V$
Output: Node interpolant of v or conjunction of children interpolants and label of v

$$S(V_0, v) = \begin{cases} I(v) & \text{if } I(v) \neq \infty \\ \bigwedge_{P(c)=v \wedge c \in V_0} S(V_0, c) \wedge L(v) & \text{otherwise} \end{cases}$$

induction over the set of nodes, it is now easy to prove that $I(v)$ satisfies the constraints of tree interpolation for every node v, and hence Algorithm 1 computes a tree interpolant I of T. Note that Algorithm 1 does not specify the concrete order in which the nodes are visited. Different feasible orderings lead to different tree interpolants.

4 Implementation in Vampire

We implemented the tree interpolation method of Algorithm 1 in the Vampire theorem prover. To make Vampire able to compute tree interpolants, we had to extend Vampire with new functionalities, including reading tree interpolation problems, deriving tree interpolants, computing interpolants of tree nodes, and theory-specific proof transformation steps for proof localisation. We also extended Vampire with built-in data types for integer-indexed arrays, and added array axioms to the built-in theory reasoning engine of Vampire. All together computing tree interpolants in Vampire required about 5000 lines of C++ code. The architecture of our implementation is given in Figure 2.

Tool usage. Our implementation is available at http://vprover.org/tree_itp. For using it, one should simply invoke Vampire on the command line as follows:

```
vampire --show_interpolant tree --[vampire/z3] problem
```

The choice of using either `vampire` or `z3` refers to proof generation (see later), whereas the input format of `problem` is as detailed below.

Input. Inputs to our implementation are tree interpolation problems in the SMT-LIB 1.2 format, using the input standard of [1]. Propositional variables are used to denote tree nodes, and logical implication is used to specify parent-child relations between nodes.

Tree interpolation. We use Algorithm 1 to compute and output a tree interpolant I of T. We explore the tree in a breadth-first manner, starting from the leaves of the tree. At each level, we visit the nodes from left-to-right and compute their interpolants.

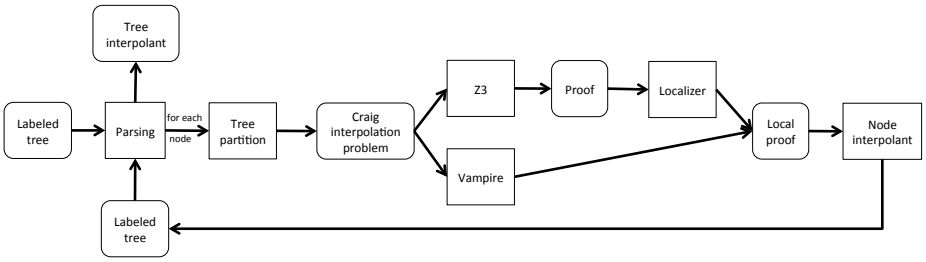


Fig. 2. Tree interpolation in Vampire.

Interpolants of tree nodes. When computing interpolants for tree nodes, we adapt the Craig interpolation procedure of [9] to our setting of R_v and B_v. We collect the set of symbols occurring only in R_v, respectively in B_v. The set of symbols used in a Craig interpolant of R_v and B_v is then defined as the set of symbols common to both R_v and B_v. With such specification of symbols, our task is to derive a local proof of $R_v \wedge B_v \rightarrow \bot$ and compute a Craig interpolant from this proof. We used [9] to construct Craig interpolants from local proofs and computed interpolants that are small both in their number of symbols and quantifiers. For generating local proofs we used the following two directions.

Proof localization. We generate local proofs either by using Vampire or by localizing an SMT proof generated by the Z3 SMT solver [4]. When using Vampire, the symbol specifications of R_v and B_v are used as Vampire annotations to ensure that the generated Vampire proofs are local. When running Z3, we first obtain a Z3 proof which might not be local. Then, using the symbol specifications of R_v and B_v we try to localize the SMT proof by (i) quantifying away some constant symbols of R_v or B_v as explained in [9] and by (ii) applying proof transformation steps over linear arithmetic and uninterpreted functions (as explained later). For parsing and localising SMT proofs, our implementation extends Vampire with built-in sorts for integer-indexed arrays and adds array reasoning based on the extensionality axioms of the array theory. To use Craig interpolants in further steps of tree interpolation, we also extended Vampire with a parser for converting Vampire formulas into the SMT-LIB format, by considering theory-specific reasoning over linear integer arithmetic and arrays.

Theory-specific reasoning for proof localization. In some cases, Vampire rewrites the non-local parts of an SMT proof without using the approach of [9] for introducing quantifiers. In [6], we have presented theory-specific proof rewriting rules that localize proofs involving linear arithmetic with uninterpreted function symbols. Vampire uses now some of these rules to recognize patterns of non-local SMT proofs and rewrite them into a local proof in the quantifier-free theory of linear arithmetic and uninterpreted functions. To illustrate how theory-aware SMT proof localisation is performed in our implementation, consider the following example. Let R be $a = b \wedge b = c$ and B the formula $c = d \wedge a \neq d$, where a, b, c, d are integer-valued constants. Clearly, R and B is unsatisfiable. A possible SMT proof of unsatisfiability (e.g. by Z3) might involve the following steps: derive $b = d$ from $b = c$ and $c = d$ and derive $a = d$ from $a = b$ and

Table 1. Tree interpolation in Vampire on quantified array problems.

Example	Description	Tree Interpolant
`Init`	set array elements to 0, update one element to 1	all elements 0 or 1
`Sorted`	sort array in ascending order	ordering between two concrete array elements
`Sorted2`	sort array in ascending order	ordering for range of array elements
`Shift`	set array elements to the values of their neighbours	array elements are all equal

$b = d$, which then contradicts $a \neq d$. Note that $b = d$ yields a non-local proof step as it uses symbols that are not common to R and B. Our proof transformation in Vampire will then reorder this equality derivation. The rewritten proof will then derive $a = c$ from $a = b$ and $b = c$ and infer $a = d$ from $a = c$ and $c = d$; clearly, this proof is local and, unlike [9], uses no quantifiers over b and d.

Optimizations. To reduce the number of local proof computations, we implemented the following heuristic. When extracting the symbols of R_v and B_v for a node v, we also derive whether R_v uses only symbols common to B_v. If this is the case, we take R_v as the interpolant $I_{R_v B_v}$. A similar heuristic is implemented also when B_v contains only symbols common to R_v. In our experiments we observed that these heuristics save expensive theorem proving calls.

Sequence and Craig Interpolants. Our implementation can also be used to compute sequence interpolants. In this case, the sequence structure is represented as a sequence of SMT-LIB assumptions, and no additional propositional variables are used to denote assumptions (i.e. tree nodes). To use Vampire for computing sequence interpolants, one should specify `sequence` instead of `tree` in the command run execution of Vampire. Our implementation can also be used to simply compute Craig interpolants of two formulas, by using the approach of [9] and specifying `on` instead of `tree` in the command line. Tree interpolation in Vampire hence brings a general interpolation procedure, which can be used for tree, sequence and Craig interpolation.

5 Experiments

We evaluated tree interpolation in Vampire using two benchmark suites. One is a collection of 4 examples where quantified reasoning over the array content is needed. These examples are taken from [13,15] and involve common array operations, such as initialization, copying and sortedness (Table 1). The other one is a collection of 175 problems (Table 2), extracted from the bounded model checking of device drivers [12]. These examples are expressed in the quantifier-free theory of linear integer arithmetic and integer-indexed arrays. All experiments reported here were obtained using a Lenovo X200 machine with 4GB of RAM and Intel Core 2 Duo processor with 2.53GHz, and are available at the url of our tool.

Quantified array problems. We computed tree interpolants in the quantified theory of arrays and integer arithmetic for 4 array problems. The examples involved procedure calls implementing array initialization, copy and sorting. We manually converted these problems into corresponding tree interpolation problems, and then run our implementation. Each tree interpolation problem had a tree with 3 nodes. Example 2 shows later one of these benchmark and Table 1 summarizes our experiments. For each example, the table states the name of the example, gives a brief description of the program,

Table 2. Tree interpolation in Vampire on quantifier-free array problems.

Prover	Nb. Benchmarks	Success	Time
Vampire	175	101	$60s$
Z3	175	113	$60s$

Table 3. Tree interpolation in Vampire and iZ3.

Tool	Quantified problems		Quantifier-free problems	
	Total	Solved	Total	Solved
Vampire	4	4	175	141
iZ3	4	1	175	175

and summarizes the tree interpolant. For the examples `Init`, `Sorted2`, and `Shift`, one tree node required the computation of a quantified node interpolant. All examples were solved in less than 1 second. The tree interpolants generated by our method were successfully used to proved quantified safety assertions over arrays.

Quantifier-free array problems. The experiments described in Table 2 involved parsing $738'890$ lines of SMT-LIB, with an average of about 90 tree nodes per benchmark. This means, that deriving a tree interpolant required on average computing 90 node interpolants per benchmarks. We distinguish between the use of Z3 or Vampire for computing local proofs of node interpolants – see column 1 of Table 2. Column 2 shows the number of benchmarks used in our experiments. Colum 3 gives the number of problems on which our implementation succeeded to compute tree interpolants, and column 4 list the average time (in seconds) per problem required by our implementation.

When using Vampire for local proof generation, we derived tree interpolants for 101 examples. Since the benchmarks were quantifier-free, the tree interpolants were quantifier-free as well. The 74 examples on which Vampire was not able to compute local proofs required more complex reasoning about arrays, involving both reading and writing to arrays.

When using Z3 for local proof generation, Z3 proved all 175 examples, however the returned proofs were not local. We succeeded to localize proofs, and hence compute tree interpolants, for 113 examples, out of which 14 tree interpolants contained quantifiers. We failed on 62 examples either because (i) proofs could not be localized, or (ii) quantified node interpolants were computed. When using quantified node interpolants in further steps of the tree interpolation, Z3 failed to find proofs in many cases.

Finally we note that some tree interpolation problems could only be solved by either using Vampire or Z3 for local proof generation. In total, we derived tree interpolants for 141 examples. The results of Table 2 suggest that improving theory-specific proof transformations as well as reasoning with both theories and quantifiers would yield better results for tree interpolation in first-order theories.

Experimental comparison. We compared our tool to the tree interpolation procedure of iZ3 [1]. Table 3 shows that iZ3 performs much better on the quanifier-free examples of Table 2. However, on the quantified array problems, iZ3 succeeded only on the `Sorted` example where the tree interpolant did not involved quantifiers. Unlike iZ3, we derived tree interpolants for all quantified problems. For the problems where iZ3 failed, we either observed an incorrect interpolant, a segmentation fault or a failed proof attempt by Z3. Example 2 shows an interpolation problem for which iZ3 computes an incorrect result. Table 3 underlines the advantage of tree interpolation in Vampire: it can be used for quantified reasoning over (arbitrary) first-order theories. To the best of our knowledge, no other approach can compute quantified tree interpolants.

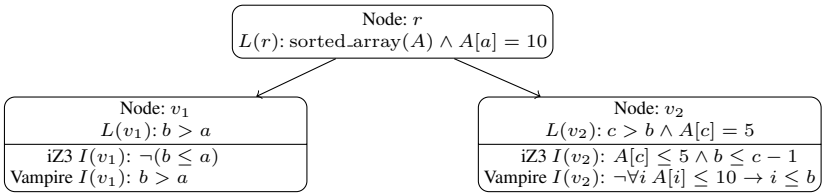


Fig. 3. Tree interpolation on the `Sorted2` example from Table 1.

Example 2. The tree structure of Figure 3 shows the tree interpolation problem of the `Sorted2` example of Table 1. In this example, a, b, c are integer-valued constants and A is an array of integers. Further, in the root label we have $\mathrm{sorted\_array}(A) \Leftrightarrow (\forall i)(\forall j)\ i < j \rightarrow A[i] < A[j]$. Figure 3 also shows the incorrect tree interpolant computed by iZ3 and the correct tree interpolant computed by Vampire.

6 Conclusion

We described how tree interpolation in Vampire is implemented and can be used. Our implementation extends Vampire with deriving tree interpolants, computing interpolants for tree nodes, theory-specific proof localisations, and built-in data structures for arrays. In addition, tree interpolation in Vampire can be used to compute sequence or Craig interpolants. Our experiments highlight the advantage of our implementation for quantified tree interpolation. Future work includes extending our implementation with better theory reasoning both for proof localisation and proving, and deriving tree interpolants from only one proof of unsatisfiability. We are also interested in evaluating the quality of our tree interpolants in the context of model checking, by using them for proving safety properties of problems.

Acknowledgements. We thank Ken McMillan for the quantifier-free benchmarks.

References

1. iZ3 Documentation. http://research.microsoft.com/en-us/um/redmond/projects/z3/old/iz3documentation.html.
2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In *Porc. of SAS*, pages 300–316, 2012.
3. W. Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. of Symbolic Logic*, 22(3):269–285, 1957.
4. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.
5. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In *Proc. of POPL*, pages 331–344, 2011.
6. A. Gupta and A. Thevenet-Montagne. Tree Interpolants via Localized Proofs, 2012. http://pub.ist.ac.at/~agupta/papers/localize-draft.pdf.

7. A. Gurfinkel, S. F. Rollini, and N. Sharygina. Interpolation Properties and SAT-based Model Checking. In *Proc. of ATVA*, 2013. To appear.
8. M. Heizmann, J. Hoenicke, and A. Podelski. Nested Interpolants. In *Proc. of POPL*, pages 471–482, 2010.
9. K. Hoder, L. Kovács, and A. Voronkov. Playing in the Grey Area of Proofs. In *Proc. of POPL*, pages 259–272, 2012.
10. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Proc. of TACAS*, pages 459–473, 2006.
11. L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, pages 1–35, 2013.
12. K. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical report, MSR, 2013.
13. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Proc. of TACAS*, pages 413–427, 2008.
14. K. L. McMillan. Interpolants from Z3 Proofs. In *Proc. of FMCAD*, pages 19–27, 2011.
15. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *Proc. of PLDI*, pages 223–234, 2009.