Vinter: A Vampire-Based Tool for Interpolation ⋆

Kryštof Hoder[1], Andreas Holzer[2], Laura Kovács[2], and Andrei Voronkov[1]

[1] University of Manchester
[2] TU Vienna

Abstract. This paper describes the Vinter tool for extracting interpolants from proofs and minimising such interpolants using various measures. Vinter takes an input problem written in either SMT-LIB or TPTP syntax, generates so called local proofs and then uses a technique of *playing in the grey areas of proofs* to find interpolants minimal with respect to various measures. Proofs are found using either Z3 or Vampire, solving pseudo-boolean optimisation is delegated to Yices, while localising proofs and generating minimal interpolants is done by Vampire. We describe the use of Vinter and give experimental results on problems from bounded model checking.

1 Introduction

Craig's interpolation [3] has become a useful technique for various tasks in software verification, such as bounded model checking [10], predicate abstraction [8], and loop invariant generation [11]. It provides a systematic way to generate predicates over program states, which are precise enough to prove particular program properties.

Let us introduce some notation and define interpolation through colors. All formulas in this paper are first-order. We will use the standard notion of an *inference*, written as $\frac{A_1 \ \dots \ A_n}{A}$, where $A_1, \dots, A_n$ denote the premises and A the conclusion of the inference. By a *derivation* or a *proof* we mean a tree built using inferences, see [9] for details. We assume that for every inference its conclusion is a logical consequence (in first-order predicate logic or in some theory) of its premises. If a formula A is derivable from a set of formulas S, we will write $S \vdash A$ and omit S if it is empty.

We will use three colors: blue, red and grey. Each symbol is colored in exactly one of these colors. By a symbol we mean a function or a predicate symbol; logical variables are not symbols. We say that a formula is *red* if it has at least one red symbol and contains only red and grey symbols. Similarly, a *blue* formula has at least one blue symbol and contains only blue and grey symbols. A formula is *grey* if all symbols in this formula are grey. Let R be a red formula, B a blue formula and $\vdash R \to B$. We call an *interpolant* of R and B any grey formula I such that (i) $\vdash R \to I$ and (ii) $\vdash I \to B$. That is, an interpolant I is in intermediate in power between R and B, and uses only grey symbols. Likewise, if $\{R, B\}$ is unsatisfiable, then a *reverse interpolant* of R and B is any grey formula I such that (i) $\vdash R \to I$ and (ii) $\{I, B\}$ is unsatisfiable. Interpolation-based verification methods make use of reverse interpolants, where R typically encodes a bounded program trace and B describes a property which is violated at the end of the trace.

⋆ We acknowledge funding from the University of Manchester and an EPSRC grant (Hoder and Voronkov), the FWF Hertha Firnberg Research grant T425-N23, the FWF National Research Network RiSE S11410-N23 and S11403-N23, the WWTF PROSEED grant ICT C-050 and the CeTAT project (Holzer and Kovács).

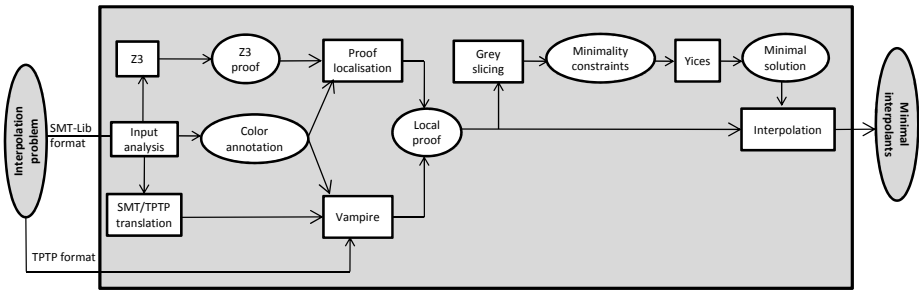

Fig. 1. Architecture of Vinter.

If $\{R, B\}$ is unsatisfiable and we have a derivation Π of $\bot$ from $\{R, B\}$, where $\bot$ denotes a (grey) formula which is always false, we are interested in extracting a reverse interpolant from Π. There are algorithms [10, 9] for extracting a reverse interpolant from derivations satisfying a locality condition: a derivation is called *local* if no inference in this derivation contains both a red and a blue symbol. A derivation that is not local is called non-local.

In [7] we observe that changes in the grey parts of local proofs, that is, parts consisting of only grey formulas, can make significant changes in the interpolants extracted from proofs. Based on this observation, we defined the notion of *grey slicing* transformations of proofs and showed that all such transformations can be captured by a propositional formula in the following sense. Given a proof Π we can find a propositional formula p such that p is true on a proof Π' if and only if Π' is a local proof obtained from Π by grey slicing. This propositional formula, together with some size measures of formulas, can be used for building small interpolants by running a pseudo-boolean optimisation tool. Small interpolants are believed to be good for end-applications since they are easier to use in proofs and are more likely to generalise than larger ones.

While the method of [7] is general and can be used for generating small interpolants, the power of the method was not well understood. One of the reason was lack of realistic examples coming from state-of-the-art model checkers. Another major obstacle to the evaluation of [7] on realistic examples was lack of a standalone tool that would take as input an interpolation problem generated by a model checker and output minimal interpolants in a fully automatic way.

We address these problem by implementing the Vinter tool that implements the interpolant minimisation technique together with the technique of localising non-local proofs (Section 2). Vinter takes an input problem written in either SMT-LIB [1] or TPTP syntax [14], generates local proofs and then finds interpolants minimal with respect to various measures. Proofs are found using either Z3 [4] or Vampire [13], solving pseudo-boolean optimisation is delegated to Yices [5], while localising proofs and generating minimal interpolants is done by Vampire.

In our previous paper [6] we already described an implementation of interpolation in Vampire. Vinter is a new tool whose aim is to generate a *minimal* interpolant. The only feature it shares with [6] is a generation of local proofs using Vampire. Translation of non-local proofs into local ones, use of SMT problems and SMT-LIB syntax instead of annotated TPTP problems, automatic annotation of problems and, most importantly, the interpolant minimisation technique implementing [7] are novel.

Vinter is available at http://vprover.org/vinter.cgi. We evaluated Vinter on examples coming from the software model checker CPAchecker [2]. The results of this evaluation are presented in Tables 1-5 and detailed in Section 3.

The contribution of this paper comes with presenting what Vinter can do and how it can be used. We comment briefly on how Vinter obtains its results and refer to [7] for details. This paper serves as a guide to generating minimal interpolants with Vinter.

2 Tool Description

Figure 1 shows the architecture of Vinter. Vinter is run on an input problem, called the interpolation problem, and accepts two options denoting the input syntax (SMT-LIB or TPTP) and the theorem prover (Vampire or Z3) used for proving.

Given an input formula, Vinter makes the following steps: (i) creation of an interpolation problem (formulas annotated with coloring information); (ii) generation of a local proof; (iii) grey slicing analysis resulting in a pseudo-boolean constraint; (iv) solving the constraint; (v) generation of a minimal interpolant. In this section we explain all these steps in some detail.

Annotated Formulas and Interpolation Problems. To specify an interpolation problem, one needs to specify two formulas R and B, together with the coloring information. However, both the SMT-LIB and TPTP syntax describe only first-order problems. To specify interpolation problems, we extended the TPTP syntax as described in [6]. For example, one can use the following annotations to specify colors:

```
vampire(symbol,function,symbol_name,symbol_arity,symbol_color).
```

However, these *color annotations* can only be understood by Vampire, and are used by Vampire to produce local proof.

For the SMT-LIB syntax we chose a different convention to specify colors. This convention was chosen by analysing bounded model checking problems in SMT-LIB. Such problems describe several unfoldings of computations going through several program states $0, 1, \ldots$. One common way of describing such problems in the SMT community is to turn state variables into functions of one argument (the state). For example, the term $f(1)$ denotes the value of state variable f at state 1.

To extract an interpolation problem from an SMT problem we "reverse engineer" SMT problems in the following way. If the SMT problem contains a unary function f which is only applied to integer constants, we consider it a state variable and replace any term of the form $f(i)$ by a constant f_i. After that we take a "middle state" m (the average integer value of all states) and consider all terms f_k for $k < m$ red and terms f_k for $k > m$ blue. The term f_m is considered grey. This corresponds to the standard use of interpolation in bounded model checking. Further, we consider the conjunction of all formulas containing red symbols as R and the conjunction of all formulas containing blue symbols as B.

Finally, if Vinter is run with Vampire, the interpolation problem written in the SMT-LIB syntax is translated into a TPTP problem with color annotations.

Generation of a Local Proof. This step depends on which theorem prover we use. We need a proof-producing theorem prover and ideally a prover producing local proofs. So far, there is not much choice on the market. Z3 seems to be the most efficient SMT solver that produces proofs which are usually non-local. Many first-order theorem provers produce proofs but only Vampire can produce local proofs [6].

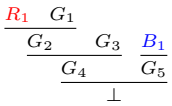

Fig. 2. Local proof Π_1.

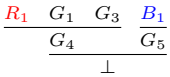

Fig. 3. Local proof Π_2 obtained by slicing off G_2 in Figure 2.

If Vinter is run using Vampire, we pass the interpolation problem to Vampire and use the option that makes it search only for local refutations. That is, if Vampire produces a proof, the proof is local.

If Vinter is run using Z3, in general we can only obtain non-local proofs. Then we use the technique of [7] to transform them into local proofs. This technique existentially quantifies away uninterpreted colored constants to make a non-local proof into a local one. In this paper however we do a bit more than [7]: instead of quantifying away only red symbols, we also existentially quantify away blue symbols. Tables 1-3 show the effect of quantifying away different colors.

Transforming non-local Z3 proofs into local ones by quantifying away colored symbols is also used in [12]. The method described in [12] also implements additional steps, such as eliminating quantifier instantiations and using a secondary interpolating prover for proof subtrees that cannot be localised. While the approach of [12] is more general than ours when it comes to localise Z3 proofs, it is quite specific to the set of proof rules used by Z3. Let us therefore note that our proof localisation can be applied to arbitrary SMT proofs. Moreover, Vinter is not restricted to SMT proofs only. To the best of our knowledge, Vinter is the first tool that generates interpolants both from SMT proofs and first-order resolution proofs, and minimises interpolants wrt various measures.

Grey Slicing. After obtaining a local proof Π, either by Vampire or by localising a Z3 proof, Vinter implements the main idea of the interpolant minimisation method: it encodes all *grey slicing transformations* of this local proof by a propositional formula P. Grey slicing is described in [7]. It is based on the idea that some grey formulas can be removed from a local proof without destroying locality and their removal can change the interpolant extracted from the proof.

Example 1. Let us illustrate how grey slicing changes the interpolants extracted from a local proof. Consider the local proof Π_1 given in Figure 2. Using the method of [9], the reverse interpolant extracted from Π_1 is $\neg G_2$.

By slicing off the grey formula G_2, that is by performing grey slicing with G_2 in Π_1, we obtain the local proof Π_2 given in Figure 3. The reverse interpolant extracted from Π_2 is $\neg G_4$. Note that both $\neg G_2$ and $\neg G_4$ can be used as a reverse interpolant extracted from Π.

The propositional formula P encoding all grey slicing transformations of a local proof Π is built by Vinter such that every satisfying assignment to P represents a local proof obtained from Π by grey slicing. Moreover, for every grey formula G in Π the formula P contains a variable p_G that is true if and only if G occurs in the interpolant. When constructing the formula P we use the property that Π is local. That is, a grey formula G in Π is either a leaf of Π or the conclusion of an inference satisfying exactly one of the following conditions: (i) the inference has only grey premises; (ii) the inference has at least one red premise and its all other premises are red or grey; (iii) the inference has at least one blue premise and its all other premises are blue or grey. Depending on the inference introducing G in Π, we generate formulas over p_G expressing

	♯ benchmarks	♯ local proofs
Vampire	4217	1903
Z3	4217	3593
red		3501
blue		3517

Table 1. Vinter results using Vampire, respectively Z3.

under which conditions G is used in the interpolant constructed from Π. We then take P as the conjunction of all formulas over p_G.

The derived formula P allows us to optimise the extracted interpolant using various measures, such as the total number of different symbols in the interpolant, the total number of different atoms, or the total number of quantifiers in such atoms. The minimisation problem can be described as a pseudo-boolean constraint using P and a linear expression built from variables p_G. For example, if we are interested in generating an interpolant that uses a minimal number of quantifiers, we construct the linear expression $\sum_G \text{quant}(G) \cdot p_G$ and derive the pseudo-boolean constraint $min_G\big(\sum_G \text{quant}(G) \cdot p_G \ \wedge \ P\big)$, where $\text{quant}(G)$ denotes the number of quantifiers in G. A solution to this constraint yields an interpolant that is minimal in the number of quantifiers.

The grey slicing and the pseudo-boolean constraint construction steps of Vinter are implemented in Vampire.

Solving the Constraint. We pass the resulting pseudo-boolean constraint to Yices [5] and generate a satisfying assignment that it is minimal wrt a given measure. In order to compute the minimal solution of the pseudo-boolean constraint, we use a divide-and-conquer approach to constrain the minimal value of the solution. That is, we make iterative calls to Yices until the upper and lower bounds on the solution become tight. For solving pseudo-boolean constraints we chose Yices since Yices runs under all recent versions of Windows, Linux and MacOS. One could expect that using pseudo-boolean constraint solvers instead of Yices and generating suboptimal solutions will give a considerable improvement in the pseudo-boolean constraint solving part of Vinter. We leave this task for future work.

Generation of a Minimal Interpolant. If Yices finds a minimal solution, we reconstruct a local derivation corresponding to this solution and use the algorithm of [9] to extract an interpolant from it. This part of Vinter is also implemented in Vampire.

3 Experimental Results

Generating interpolation problems, localising proofs, grey slicing and minimising interpolants are written in C++, using all together 4209 lines of C++ code. In addition, Vinter contains about 200 lines of shellscript code for merging its various parts and realising the architecture of Figure 1. All experiments described in this section were obtained on a 64-bit 2.33 GHz quad core Dell machine with 12 GB RAM.

Benchmarks. In [7] we reported on initial results on generating minimal interpolants, by using examples from the TPTP and the SMT-LIB libraries. Experiments on more realistic verification benchmarks, that is on examples coming from concrete verification tools were left for future study.

In this paper, we address this task and evaluate Vinter on 4217 examples generated by the software model checker CPAchecker [2]. Some of these examples are also used in the *software verification competition* – see http://sv-comp.sosy-lab.org/benchmarks.php.

benchmark	measure	no	> 1	> 2	> 3	> 5
all	weight	3219	282	54	24	4
all	atoms	3417	84	25	5	
all	quant	3501				
ssh	weight	99	62	18		
ssh	atoms	160	1			
ssh	quant	161				
systemc	weight	806	199	22	17	2
systemc	atoms	936	69	18	3	
systemc	quant	1005				
nested	weight	2314	21	14	7	2
nested	atoms	2321	14	7	2	
nested	quant	2335				

Table 2. Minimal interpolants extracted from localised Z3 proofs, after quantifying away red symbols.

benchmark	measure	no	> 1	> 2	> 3	> 5
all	weight	2096	1367	30	20	4
all	atoms	3434	54	24	5	
all	quant	2392	10	1115		
ssh	weight	74	72	18		
ssh	atoms	162	2			
ssh	quant	159		5		
systemc	weight	804	192	5	15	2
systemc	atoms	951	45	19	3	
systemc	quant	995	10	13		
nested	weight	1218	1103	7	5	2
nested	atoms	2321	7	5	2	
nested	quant	1238		1097		

Table 3. Minimal interpolants extracted from localised Z3 proofs, after quantifying away blue symbols.

To be precise, we took the following three benchmark suites: *ControlFlowInteger* examples that express properties about the control-flow structure and the integer variables of programs; *SystemC* examples about concurrent programs; and *Nested* loop examples. From each of these 4217 examples we generated interpolation problems for Vinter, as follows. We used CPAchecker to determine the reachability of error locations in the benchmark files. Each time an error location was encountered, the unsatisfiable formula encoding the infeasibility of the error-prone program path was output in the SMT-LIB format. These SMT-LIB examples were further used as inputs to Vinter. All SMT-LIB examples we used CPAchecker involved linear arithmetic and uninterpreted functions.

Generating local proofs. The interpolation problems coming from CPAchecker are using few quantifiers, but have a deeply nested linear arithmetic structure. Such problems can be efficiently proved by SMT solvers, such as Z3. On the contrary, first-order theorem provers, such as Vampire, are good in dealing with quantifiers but have difficulties with theory reasoning. Theory reasoning in Vampire is supported by using sound but incomplete theory axiomatisations. On the other hand, when we run Vinter with Vampire, whenever Vampire produces a proof, the proof is local. This is not the case with Z3, essentially proofs produced by Z3 are non-local and thus need to be localised for interpolant generation. Depending which theorem prover Vinter is using, generating minimal interpolants is challenging due to theory reasoning and/or proof localisation.

Table 1 gives an overview of Vinter's results on the CPAChecker benchmarks. The first column of Table 1 specifies the prover used for finding proofs, the second column the number of interpolation problems, and the third column the number of examples for which a local proof was found. When Vinter was run with Z3, Table 1 also list the number of localised proofs by quantifiying away the red, respectively the blue constants.

When we ran Vinter with Z3, all 4217 CPAchecker examples were proved by Z3 in essentially no time. However, when considering color annotations over the proof symbols, the proof localisation step of Vinter generated local proofs for 3593 examples out of the 4217 examples. Some of these proofs could only be localised by quantifying away the blue (respectively, the red) constants. More precisely, by analysing our results, we observed that 3501 Z3 proofs were localised by quantifying away the red constants, and 3517 Z3 proofs were localised when we quantified away the blue constants. Our experiments give thus practical evidence that our extension to [7] for quantifying away either red or blue symbols effects interpolant minimisation. Combining red *and* blue symbol quantification in the *same* non-local proof is an interesting task to investigate.

benchmark	measure	no	> 1	> 2	> 3	> 5	> 10	> 20	> 50
all	weight	1266	637	217	152	81	22	8	2
	atoms	1702	201	111	18	2			
	quant	1833	70	47	8				
ssh-simpl	weight	85	8	4	4	3	1		
	atoms	84	1	1					
	quant	85							
systemc	weight	592	53	29	16	11	8		
	atoms	616	16	13					
	quant	645							
nested	weight	597	576	184	132	67	13	8	2
	atoms	1002	171	97	18	2			
	quant	1103	70	47	8				

Table 4. Minimal interpolants extracted from Vampire proofs.

When evaluating Vinter using Vampire on the 4217 CPAchecker examples, we ran Vampire with a 60 seconds time limit. The CPAchecker examples expressed in the SMT-LIB syntax were translated by Vinter into an annotated Vampire input. Using the coloring annotations over the input symbols, Vampire produced local proofs for 1903 benchmarks out of the 4217 examples. For the remaining 2314 examples, Vampire failed to generate a proof. One reason why Vampire failed to produce a proof was that these examples have a deeply nested linear arithmetic structure. In addition, we also observed that the coloring annotations may lead to performance degradation in Vampire's reasoning processes. We believe that improving the theory reasoning and the local proof generation engines of Vampire would yield better performances of Vinter.

Generating minimal interpolants. Vinter implements the following three measures to minimise interpolants: (i) the number of symbols (*weight* measure), (ii) the number of atoms (*atom* measure), and (iii) the number of quantifiers (*quant* measure) in the interpolant. For solving the pseudo-boolean optimization problems describing minimality constraints over interpolant formulas, we ran Yices with a timeout of 126 seconds.

Tables 2 and 3 summarise our results on minimising interpolants extracted from Z3 proofs, whereas Table 4 reports on our experiments for generating minimal interpolants from Vampire proofs. The first column of these tables lists the set of CPAchecker examples: *all* refers to all benchmarks for which minimal interpolants were generated, whereas *ssh*, *systemc* and *nested* denote examples out of all these benchmarks which come from the *ControlFlowInteger*, *SystemC*, and *Nested* benchmark suites. For each benchmark set, the second column shows the measure used for minimising interpolants. Similarly, for each benchmark set, columns starting with column three present the number of those examples for which the measure decreased and by the factor given in the headers of the columns. For example > 2 means that the measure increased by a factor greater than 2. Note that the best improvement was obtained by running Vampire, and especially on the *nested* benchmarks. This means that having initial proofs of good quality is crucial for the success of the method.

Table 5 illustrates the sizes of interpolants before and after weight minimisation. For example, one can see that on all examples, 37 interpolants had the weight ≥ 50 before minimisation and only 5 of them had a weight ≥ 50 after the minimisation. Note that the largest interpolants (with 100–500 symbols) were all minimised.

By analysing our results, we also encountered problems on which Vampire produced local proofs, and hence interpolants, but the Z3 proof could not be localised and thus interpolants could not computed.

benchmark		0	≥ 1	≥ 3	≥ 5	≥ 10	≥ 20	≥ 50	≥ 100
all	before	524	1379	1303	770	348	121	37	6
	after	524	1379	1248	396	226	46	5	
ssh-simpl	before	8	77	75	13	3	1	1	
	after	8	77	71	6	2	1	1	
systemc	before	360	285	227	152	41	9	3	
	after	360	285	219	124	17	2		
nested	before	156	1017	1001	605	304	111	33	6
	after	156	1017	958	266	207	43	4	

Table 5. Weight of interpolants before and after minimisation, using Vampire proofs.

Summarising, we believe that the experimental results of Tables 2-5 indicate that Vinter can be successfully used for generating *small* interpolants and that the effect of minimisation is better when the initial proofs are local. For example, Vinter could sometimes decrease the weight of interpolants by a factor of more than 100. We believe that Vinter can help software verification tools deal with significantly smaller interpolants.

4 Conclusions

We describe the Vinter tool for localising proofs, extracting interpolants from local proofs and minimising interpolants using various measures. We present the use of Vinter and evaluate Vinter on a collection of bounded model checking examples. Future work includes integrating Vinter with end-applications and generating interpolation-friendly proofs.

References

1. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
2. D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proc. of CAV*, pages 184–190, 2011.
3. W. Craig. Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22(3):269–285, 1957.
4. L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, pages 337–340, 2008.
5. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proc. of CAV*, pages 81–94, 2006.
6. K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. In *Proc. of IJCAR*, pages 188–195, 2010.
7. K. Hoder, L. Kovacs, and A. Voronkov. Playing in the Grey Area of Proofs. In *Proc. of POPL*, pages 259–272, 2012.
8. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Proc. of TACAS*, pages 459–473, 2006.
9. L. Kovacs and A. Voronkov. Interpolation and Symbol Elimination. In *Proc. of CADE*, pages 199–213, 2009.
10. K. L. McMillan. An Interpolating Theorem Prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
11. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, pages 413–427, 2008.
12. K. L. McMillan. Interpolants from Z3 Proofs. In *Proc. of FMCAD*, pages 19–27, 2011.
13. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
14. Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.