Interpolation and Symbol Elimination in Vampire*

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

[1] University of Manchester
[2] TU Vienna

Abstract. It has recently been shown that proofs in which some symbols are colored (e.g. local or split proofs and symbol-eliminating proofs) can be used for a number of applications, such as invariant generation and computing interpolants. This tool paper describes how such proofs and interpolant generation are implemented in the first-order theorem prover Vampire.

1 Introduction

Interpolation offers a systematic way to generate auxiliary assertions needed for software verification techniques based on theorem proving [7, 10], predicate abstraction [5, 7], constraint solving [15], and model-checking [12, 1].

In [9] it was shown that symbol-eliminating inferences extracted from proofs can be used for automatic invariant generation. Further, [10] gives a new proof of a result from [7] on extracting interpolants from colored proofs:[3] this proof contains an algorithm for building (from colored proofs) interpolants that are boolean combinations of symbol-eliminating steps. Thus, [10] brings interpolation and symbol elimination together.

Based on the results of [9, 10] we implemented colored proof generation in the first-order theorem prover Vampire [14]. Colored proofs form the base for our interpolation and symbol elimination algorithms.

The purpose of this paper is to describe how interpolation and symbol elimination are implemented and can be used in Vampire. We do not overview Vampire itself but only describe its new functionalities. The presented features have been explicitly designed for making Vampire appropriate for formal software verification: symbol elimination for automated assertion (invariant) synthesis and computation of Craig interpolants for abstraction refinement. Unlike its predecessors, the "new" Vampire thus provides functionalities which extend the applicability of state-of-the-art first theorem provers in verification. To the best of our knowledge, it is the first theorem prover that supports both invariant generation and interpolant computation.

The obtained symbol eliminating inferences and interpolants contain quantifiers, and can be further used as invariant assertions to verify properties of programs manipulating arrays and linked lists [13, 9]. We believe that software verification may benefit from the interpolant generation engine of Vampire.

Implementation. The new version of Vampire is available from http://www.vprover.org and runs under most recent versions of Linux (both 32 and 64 bits), MacOS and Windows. Vampire is implemented in C++ and has about 73,000 lines of code.

* This work has been partly done while the second authors was at ETH Zürich.

[3] Such proofs are also called *local* and *split proofs*, in this paper we will call them colored.

Experiments. We successfully applied Vampire on benchmarks taken from recent work on interpolants and invariants [6, 19, 3, 4, 15, 8] – see Section 4 and the mentioned URL. Our methods can discover required invariants and interpolants in all examples, suggesting its potential for automated software verification.

Related work. There are several interpolant generation algorithms for various theories. For example, [12, 5, 7, 1] derive interpolants from resolution proofs in the combined ground theory of linear arithmetic and uninterpreted functions. The approach described in [15] generates interpolants in the combined theory of arithmetic and uninterpreted functions using constraint solving techniques over an a priori defined interpolants template. The method presented in [13] computes quantified interpolants from first-order resolution proofs over scalars, arrays and uninterpreted functions.

Our algorithm implemented in Vampire automatically extracts interpolants from colored first-order proofs in the superposition calculus. Theories, such as arithmetic or theories of arrays, can be handled by adding theory axioms to the first-order problem to be proved. Thus, interpolation in Vampire is not limited to decidable theories for which interpolation algorithms are known. One can use arbitrary first-order axioms. However, a consequence of this generality is that we do not guarantee finding interpolants even for decidable theories. Moreover, if a theory is not finitely axiomatisable, we can only use its incomplete first-order axiomatisation.

As far as we know, symbol elimination has not been implemented in any other system. A somehow related approach to symbol elimination is presented in [13, 16] where theorem proving is used for inferring loop invariants. Contrary to our approach, the cited works are adapted to prove given assertions as opposed to generating arbitrary invariants. Using the saturation-based theorem prover SPASS [18], [13] generates interpolants as quantified invariants that are strong enough to prove given assertions. In [16] templates over predicate abstraction are used, reducing the problem of invariant discovery to that of finding solutions, by the Z3 SMT solver [2], for unknowns in an invariant template formula. Unlike [13, 16], we automatically generate invariants as symbol eliminating inferences in full-first order logic, without using predefined predicate templates or assertions.

2 Colored Proofs, Symbol Elimination and Interpolation

Colored proofs are used in a context when some (predicate and/or function) symbols are declared to have colors. In colored proofs every inference can use symbols of at most one color, as a consequence, every term or atomic formula used in such proofs can use symbols of at most one color, too. We will call a symbol, term, clause etc. *colored* if it uses a color, otherwise it is called *transparent*.

In **symbol elimination** [9] we are interested in inferences having at least one colored premise and a transparent conclusion; such inferences are called *symbol-eliminating*. Conclusions of symbol-eliminating inferences can be used to find loop invariants. Symbol elimination can be reformulated as *consequence-finding*: we are trying to find transparent consequences of a theory including both colored and transparent formulas. Note that, unlike traditional applications of first-order theorem proving, we are not interested in finding a refutation: symbol-eliminating inferences can be obtained by running a theorem prover on a satisfiable formula, for which no refutation exists.

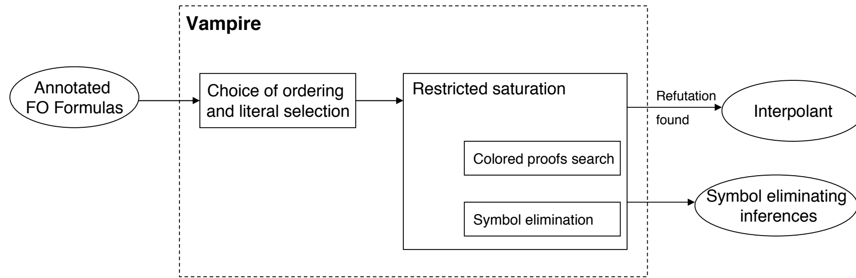

Fig. 1. Interpolation and Symbol Elimination in Vampire.

A formula I is called an **interpolant** of formulas L and R (with respect to a theory T) if the following conditions are satisfied:

(1) $T \vdash L \rightarrow I$;
(2) $T \vdash I \rightarrow R$;
(3) I uses only symbols occurring either in T or in both L and R.

Interpolation can be reformulated in terms of colors as follows: we assign one color to symbols occurring only in L and another color to symbols occurring only in R: then the last condition on interpolants can be reformulated as *I is transparent*. For extracting interpolants from colored proofs we use the algorithm described in [10].

The notion of interpolant has been changed in the model-checking community starting with [12]. Namely, the condition (2): $T \vdash I \rightarrow R$ has been replaced by (2a): $T \vdash I \wedge R \rightarrow \bot$. To avoid any confusion between the two notions of interpolant, in [10] any formula I satisfying conditions (1), (2a) and (3) is called a *reverse interpolant of L and R*. Clearly, reverse interpolants for L and R are exactly interpolants of L and $\neg R$.

In the sequel, we reserve the notation L and R for the two formulas whose interpolant is to be computed.

3 Tool Overview

Vampire [14] is a general purpose first-order theorem prover based on the resolution and superposition calculus. To implement symbol elimination and interpolation in Vampire, we had to extend it by new functionalities, change the inference mechanism to be able to generate colored derivations, and implement an algorithm for extracting interpolants. The workflow of interpolation and symbol elimination in Vampire is illustrated in Figure 1.

Annotated formulas. Vampire reads problems expressed in the TPTP syntax [17]: a Prolog-like syntax allowing one to specify input axioms and conjecture for theorem provers. We had to extend the input syntax to make it rich enough to define colors and interpolation requests. In fact, we had to extend it even more since in the application of interpolation and symbol elimination the set of symbols that can occur in the interpolants is not necessarily the intersection of the languages of L and R with addition of theory symbols. We extended the TPTP syntax with Vampire-specific declarations. Their use is illustrated in Figure 2 and detailed in Example 1 taken from [13].

```
vampire(symbol,function,a,0,left).
vampire(symbol,function,b,0,left).
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,c,0,right).
vampire(option,show_interpolant,on).
```

```
vampire(left_formula).
  fof(a1,axiom, q(f(a))).
  fof(a1,axiom, ~q(f(b))).
vampire(end_formula).
vampire(right_formula).
  fof(a2,conjecture, ? [V] : (f(V)!=c)).
vampire(end_formula).
```

Fig. 2. Specification of Interpolation.

EXAMPLE 1. [13] Consider the problem of computing an interpolant of $q(f(a)) \wedge \neg q(f(b))$ (i.e. L) and $\exists v(f(v) \neq c)$ (i.e. R).

The first three declarations shown in the left column of Figure 2 say that a, b are constants (function symbols of arity 0) and q is a unary predicate symbol colored in the "left" color (that is, in the language of L). Likewise, the fourth declaration in the left column says that c is a constant colored in the "right" color (that is, in the language of R). Finally, the left column contains an option that sets interpolant generation. This option can also be passed in the command line. The declarations `fof(..)` are TPTP declarations for introducing formulas. The vampire declarations `left_formula`, `right_formula` and `end_formula` are used to define L and R. If we have formulas not in the scope of the `left_formula` or `right_formula` declarations, they are considered as part of the theory T.

To use Vampire for symbol elimination, we can simply assign all symbols to be eliminated the left color and leave the right color unused. For concrete examples see http://www.vprover.org.

Colored proof generation. In order to support the generation of colored proofs, the following had to be implemented.

1. We had to block inferences that have premises of two different colors.
2. We had to change the simplification ordering and literal selection, so that colored terms are larger than transparent ones, and that (when possible) transparent literals are selected only when there are no colored ones. To make colored terms bigger than transparent, we had to implement the Knuth-Bendix ordering with ordinals as defined in [11] and make colored symbols to have the weight ω, while transparent symbols to have finite weights.

To output the *conclusions of symbol eliminating inferences*, we check premises of each inference that produced a transparent clause, and if one of the premises is colored, we output the resulting clause.

Interpolants are generated from refutations using the algorithm described in [10]. For instance, given the input shown in Example 1, Vampire outputs the interpolant $\neg\forall x\forall y(f(x) = f(y))$.

Symbol Elimination. To make Vampire output conclusions of symbol-eliminating inferences, one should set the option `show_symbol_elimination` to `on`. As Vampire is not supposed to terminate in the symbol-eliminating mode, it is wise to specify a time limit when it is run in this mode.

Formulas	Coloring	Reverse Interpolant
$L: z < 0 \land x \leq z \land y \leq x$ $R: y \leq 0 \land x + y \geq 0$	left: z right: -	$x < 0$
$L: g(a) = c + 5 \land f(g(a)) \geq c + 1$ $R: h(b) = d + 4 \land d = c + 1 \land f(h(b)) < c + 1$	left: g, a right: h, b	$c + 1 \leq f(c + 5)$
$L: p \leq c \land c \leq q \land f(c) = 1$ $R: q \leq d \land d \leq p \land f(d) = 0$	left: c right: d	$p \leq q \land (q > p \lor f(p) = 1)$
$L: f(x_1) + x_2 = x_3 \land f(y_1) + y_2 = y_3 \land y_1 \leq x_1$ $R: x_2 = g(b) \land y_2 = g(b) \land x_1 \leq y_1 \land x_3 < y_3$	left: f right: g, b	$x_1 > y_1 \lor x_2 \neq y_2 \lor x_3 = y_3$
$L: c_2 = car(c_1) \land c_3 = cdr(c_1) \land \neg(atom(c_1))$ $R: \neg(c_1) = cons(c_2, c_3)$	left: $car, cons$ right: -	$\neg atom(c_1) \land c_1 = cons(c_2, c_3)$
$L: Q(f(a)) \land \neq Q(f(b))$ $R: f(V) = c$	left: Q, a, b right: c	$\exists x, y : f(x) \neq f(y)$
$L: a = c \land f(c) = a$ $R: c = b \land \neq (b = f(c))$	left: a right: b	$c = f(c)$
$L: True \land a'[x'] = y \land x' = x \land y' = y + 1 \land z' = x'$ $R: \neg(a'[z'] = y' - 1)$	left: x, y right: -	$1 + a'[x'] = y' \land x' = z'$

Table 1. Interpolation with Vampire.

4 Experiments

We have successfully run Vampire on benchmark examples taken from recent literature on interpolation and invariant generation. In this section we present two different sets of experimental results that underline the effectiveness of our implementation. The reported results were obtained on a machine with 2 GHz processor and 2GB of RAM.

Interpolation. Table 1 summarises some of our results for computing interpolants on examples that have been used as motivating examples by previous techniques [6, 19, 15, 13]. The first column of Table 1 presents the input formulas L and R whose interpolants is going to be computed. The second column shows symbols declared colored, whereas the third column shows the interpolant generated by Vampire.

All interpolants given in Table 1 were computed by Vampire in essentially no time (e.g. in less than 0.1 second). In the first four examples of Table 1 a simple axiomatisation of arithmetic with the greater-than relation and successor function was used. The fifth example of Table 1 uses the theory of lists, whereas the last example of Table 1 uses the combined theory of arrays and arithmetic.

The last example of Table 1 originates from an example taken from [6], and is a request to prove the infeasibility of the following one-path program annotated by a pre- and a post-condition:

$$\{\top\} \quad a[x] := y; y := y + 1; z := x \quad \{a[z] \neq y - 1\}.$$

Let x, y, z, a denote the initial and x', y', z', a' the final values of program variables. Based on the bounded-model checking approach [12], proving infeasibility of the above program path boils down to computing an interpolant for the formulas $\top \land T(\{x, y, z, a\}, \{x', y', z', a'\})$ and $a'[z'] \neq y' - 1$, where $T(\{x, y, z, a\}, \{x', y', z', a'\}) \equiv a'[x'] = y \land x' = x \land y' = y + 1 \land z' = x'$ is the transition relation defined by the program. The interpolant computed by Vampire proves that the program has no feasible path from the initial state to the final state.

Symbol Elimination. Experiments with symbol elimination on array programs taken from [4, 8] are summarised in Table 2. We ran Vampire in the symbol elimination mode with a time limit of 10 seconds. We recall that each conclusion of a symbol-eliminating inference is a loop invariant [9].

Loop	♯ of SEI	♯ of Min SEI	SEI as Invariant

`Initialisation` [8]
$a = 0;$
while $(a < m)$ **do**
$aa[a] = 0; a = a + 1$
end do
	399	15	$\forall x : 0 \leq x < a \rightarrow aa[x] = 0$

`Copy` [8]
$a = 0;$
while $(a < m)$ **do**
$bb[a] = aa[a]; a = a + 1$
end do
	379	14	$\forall x : 0 \leq x < a \rightarrow bb[x] = aa[x]$

`Vararg` [8]
$a = 0;$
while $(aa[a] > 0)$ **do**
$a = a + 1$
end do
	1	1	$\forall x : 0 \leq x < a \rightarrow aa[x] > 0$

`Partition` [4]
$a = 0; b = 0; c = 0;$
while $(a < m)$ **do**
if $(aa[a] >= 0)$
then $bb[b] = aa[a]; b = b + 1$
else $cc[c] = aa[a]; c = c + 1$
end if;
$a = a + 1$
end do
	150	61	$\forall x : 0 \leq x < b \rightarrow$
			$\exists y : 0 \leq y < a \rightarrow bb[x] = aa[y]$

`Partition_Init` [8]
$a = 0; c = 0;$
while $(a < m)$ **do**
if $(aa[a] == bb[a])$
then $cc[c] = a; c = c + 1$
end if;
$a = a + 1$
end do
	18	13	$\forall x : 0 \leq x < c \wedge 0 \leq x < a \rightarrow$
			$aa(cc(x)) = bb(cc(x))$

Table 2. Symbol Elimination with Vampire on Array Programs.

For all examples of Table 2 we show in the rightmost column a desired invariant that could be computed using other techniques. We were interested in the following: (i) can Vampire generate the invariant itself and if not, (ii) can Vampire generate invariants that would imply the desired invariant?

After running Vampire in the symbol-eliminating mode we sometimes obtain a large set of invariants. The number of invariants (that is, the number of symbol-eliminating inferences) is shown in the second column of the Table 2. To make them usable we did the following minimization: remove invariants that are implied by the theory axioms or by other invariants. For the task we used Vampire itself. Obviously, the problem whether an invariant is implied by other invariants, is undecidable, so we ran Vampire with a time limit of 0.3 seconds once for each invariant, trying to prove it from the remaining invariants and the theory axioms. The number of invariants that could not be proved redundant is shown in the third column of Table 2.

We tried many more examples, always with success, that is, the invariants generated by Vampire using symbol elimination always implied the desired invariant. There are many interesting issues related to symbol elimination which cannot be discussed here due to lack of space.

5 Conclusion

We described how interpolant generation and symbol elimination are implemented and can be used in the first-order theorem prover Vampire. Future work includes integrating Vampire into software verification tools for automatically generating interpolants and supporting the entire process of verification. Inferring a minimal set of invariants and improving these invariants can also be done using theorem proving and remains an interesting topic for further research.

References

1. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 397–412, 2008.
2. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
3. D. Gopan, T. W. Reps, and M. Sagiv. A Framework for Numeric Analysis of Array Operations. In *POPL*, pages 338–350, 2005.
4. S. Gulwani and A. Tiwari. Combining Abstract Interpreters. In *Proc. of PLDI*, pages 376–386, 2006.
5. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *Proc. of POPL*, pages 232–244, 2004.
6. R. Jhala and K. L. McMillan. Interpolant-Based Transition Relation Approximation. In *Proc. of CAV*, pages 39–51, 2005.
7. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *Prc. of TACAS*, pages 459–473, 2006.
8. R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 193–206, 2007.
9. L. Kovacs and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.
10. L. Kovacs and A. Voronkov. Interpolation and Symbol Elimination. In *Proc. of CADE*, pages 199–213, 2009.
11. M. Ludwig and U. Waldmann. An Extension of the Knuth-Bendix Ordering with LPO-Like Properties. In *Proc. of LPAR*, pages 348–362, 2007.
12. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *Proc. of CAV*, pages 1–13, 2003.
13. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.
14. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
15. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *Proc. of VMCAI*, pages 346–362, 2007.
16. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *PLDI*, pages 223–234, 2009.
17. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *J. of Automated Reasoning*, To appear, 2009.
18. C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System Description: SpassVersion 3.0. In *Proc. of CADE*, volume 4603 of *LNAI*, pages 514–520, 2007.
19. G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *Proc. of CADE*, pages 353–368, 2005.