Case Studies on Invariant Generation Using a Saturation Theorem Prover [⋆]

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

[1] University of Manchester
[2] TU Vienna

Abstract. Automatic understanding of the intended meaning of computer programs is a very hard problem, requiring intelligence and reasoning. In this paper we evaluate a program analysis method, called symbol elimination, that uses first-order theorem proving techniques to automatically discover non-trivial program properties. We discuss implementation details of the method, present experimental results, and discuss the relation of the program properties obtained by our implementation and the intended meaning of the programs used in the experiments.

1 Introduction

The complexity of computer systems grows exponentially. Many companies and organisations are now routinely dealing with software comprising several millions lines of code, written by different people using different languages, tools and styles. This software is hard to understand and is integrated in an ever changing complex environment, using computers, networking, various physical devices, security protocols and many other components. Ensuring the reliability of such systems for safety-critical applications is extremely difficult. One way of solving the problem is to analyse or verify these systems using computer-aided tools based on computational logic.

In [9] a new method, called *symbol elimination*, has been proposed to automatically generate statements expressing computer program properties. The approach requires no preliminary knowledge about program behavior, but uses the power of a saturation theorem prover to derive and understand the *intended meaning* of the program. To undertake such a complex task, reasoning in the combination of first order logic and various theories is required as program components involve both bounded and unbounded data structures.

One can argue that automatic inference of program properties is a hard and creative problem whose solution improves our understanding of the relation between the computer reasoning and the human reasoning. Indeed, given a computer program, one can ask a computer programmer questions like "what are the essential properties of this program" or "what is the intended meaning of this program?" Answering such question requires intelligence. If the program is small and not highly sophisticated, one can expect that the programmer will be able to give some answers. For example, if the

⋆ This research was partly supported by Dassault Aviation. Kryštof Hoder is supported by the School of Computer Science at the University of Manchester and by an EPSRC Pathfinder grant. Laura Kovács is supported by an FWF Hertha Firnberg Research grant (T425-N23). Andrei Voronkov is partially supported by an EPSRC grant. This research is partly supported by the FWF National Research Network RiSE (S11410-N23).

program copies one array into another, one can expect the programmer to say that the intended meaning of the program is to copy arrays and that among the most essential properties of this program are the facts that the two arrays will be equal and the first array will not be modified. These two properties are first-order properties, that is, they can be expressed in first-order logic.

The properties generated by the symbol elimination method are first-order properties, therefore one can ask a question of whether a computer program can generate such properties and whether it can generate "the intended" properties (whatever it means). This paper tries to answer this fundamental question by taking a program annotated by humans, removing these annotations, generating program properties by a computer program, and comparing the generated program properties with the intended properties.

The first implementation of symbol elimination was carried out in the first-order resolution theorem prover Vampire [7]. This implementation could be used for symbol elimination, yet not for invariant generation. It was run on an array partitioning program to demonstrate that it can generate complex loop invariants with alternating quantifiers, which could not be generated by any other existing technique. Such complex loop properties precisely capture the intended first-order meaning of the program, as well as the programmer's intention while writing the program.

However, the practical power of symbol elimination was not clear since it required extensive experiments on programs containing loops. Such experiments were not easy to organise since they involved combining several tools for program analysis and theorem proving. Designing such a combination turned out to be non-trivial and error-prone. The first standalone implementation of program analysis in the theorem prover Vampire is described in the system abstract [8]. Experimental evaluation of the method was however not yet carried out.

This paper undertakes the first extensive investigation into understanding the power and limitations of symbol elimination for invariant generation. In addition to the fundamental AI problems mentioned above, we were also interested in the power of the method for applications in program analysis, which can be measured by the following characteristics:

1. [Strength] Is the method powerful enough to infer automatically invariants that would imply intended loop properties?
2. [Time] Is invariant generation fast enough?
3. [Quantity] What is the number of generated invariants?

The method we use to answer these questions in this work is described below.

- We use the invariant generation framework of [8] implemented directly in Vampire. Vampire can now accept as an input a program written in a subset of C, find all loops in it, and generate and output invariants for each of the loops. To this end, to make [8] practically useful, we extended the program analyser of [8] with a C parser.
- We took a number of annotated C programs coming from an industrial software verification project, as well as several standard benchmarks circulated in the verification community (for example, [2, 13]), and removed annotations corresponding to loop invariants.
- We ran Vampire with various time limits and collected statistics relevant to the questions raised above.

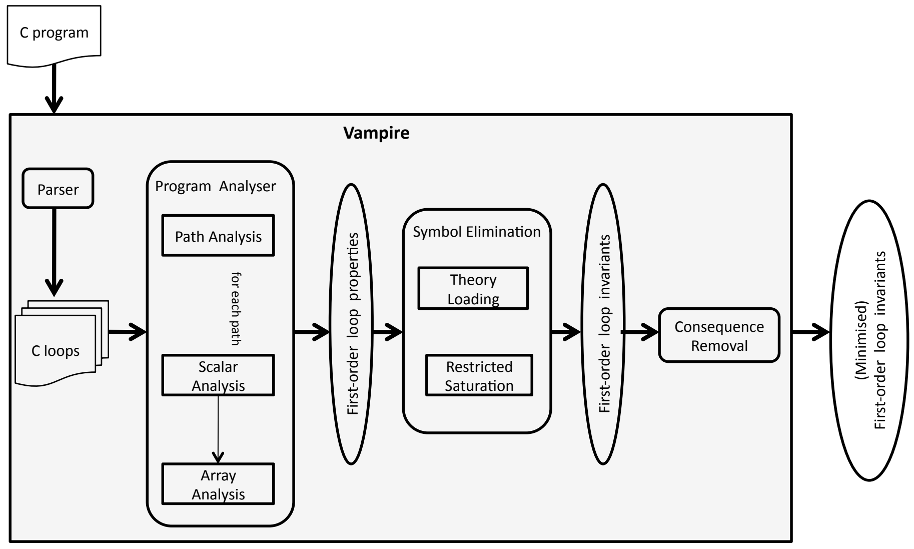

Fig. 1. Invariant Generation in Vampire.

The main contribution of this paper is an experimental evaluation of the symbol elimination approach, providing statistics and better understanding of the power of the method.

Our experiments show that, at least for small (but far from trivial) programs a computer program using the symbol elimination method turns out to generate in less than a second properties that imply the annotated properties, that is, the intended properties. Hence, symbol elimination confirms, in some way, the power of deductive methods in computer reasoning. However, even for relatively simple programs it also generates many other properties, which shows that further investigation may be required to bring the human and the computer understanding of the meaning of programs closer.

This paper is structured as follows. Section 3.1 overviews the program analysis framework of Vampire. Given a program in a subset of C as an input, Vampire now *automatically* generates a set of invariants for the loops occurring in the program.

Reasoning about programs requires reasoning in combined first-order theories, such as arithmetic, uninterpreted functions, arrays, etc. Since invariant generation by symbol elimination requires both theories and quantifiers, efficient handling of theories in Vampire was a major issue for us. We describe theory reasoning in Vampire in Section 3.2.

Since loop invariants are to be used in proving program properties, it is important that the generated set of invariants is not too large, yet powerful enough to imply important program properties. A framework for removing invariants implied by other invariants was introduced in [8], where invariants are removed in combination with invariant generation. In our work we use [8], as described in Sections 3.3 and 3.4. We further give an experimental evidence that such a removal is in practice "cheap" as compared to invariant generation (Section 4).

The "quality" of a set of automatically generated invariants refers to whether it can be used to easily derive program properties required for program verification or static analysis. We discuss the quality of invariants generated by our technique in Section 3.5.

The key section of this paper is Section 4 on experimental evaluation of the symbol elimination method. The reported experimental results provide empirical evidence of the power of symbol elimination for generating complex invariant.

We briefly consider related work in Section 5 and draw conclusions in Section 6.

2 Preliminaries

Programs, Variables and Expressions. Figure 1 illustrates our approach to practical invariant generation in Vampire. Figure 1 extends [8] by the use of a program parser. Hence, Vampire now can handle C programs over integers and arrays using assignments, loops and if-then-else conditional statements with standard (C language) semantics. Non-integer variables can also be considered but regarded as uninterpreted.

We assume that the language expressions contain integer constants, variables, and some function and predicate symbols. The standard arithmetical functions, such as $+$, $-$, $\cdot$ are considered as interpreted, while all other function symbols are uninterpreted. Likewise, the arithmetical predicate symbols $=$, $\neq$, $\leq$, $\geq$, $<$ and $>$ are interpreted while all other predicate symbols are uninterpreted. As usual, the expression $A[e]$ is used to denote the element of an array A at the position given by the expression e. The current version of Vampire does not handle nested loops. If the input program contains such loops, only the innermost loops will be analysed.

The programming model for invariant generation in Vampire is summarized below.

$$\underline{\textbf{while}}\ b\ \underline{\textbf{do}}\quad s_1; s_2; \ldots; s_m \quad \underline{\textbf{end do}} \tag{1}$$

where b is a boolean expression, and statements s_i ($i = 1, \ldots, m$) are either assignments or (nested) if-then-else conditional statements. Some example loops for invariant generation in Vampire are given in the leftmost column of Table 3. Throughout this paper, integer program variables are denoted by lower-case letters $a, b, c, \ldots$, whereas array variables are denoted by upper-case letters $A, B, C, \ldots$.

Theorem proving and Vampire. The symbol elimination method described and investigated in this paper is essentially based on the use of a first-order theorem prover. Moreover, one needs a theorem prover able to deal with theories (e.g. integer arithmetic), first-order logic, and generate consequences. Theorem provers using saturation algorithms are ideal for consequence generation. Saturation actually means that the theorem prover tries to generate all, in some sense, consequences of a given set of formulas in some inference system, for example, resolution and superposition [12].

In reality, saturation theorem provers use a powerful concept of *redundancy elimination*. Redundancy elimination is not an obstacle to consequence generation, since redundant formulas are logical consequences of other formulas the theorem prover is dealing with.

All our experiments described in this paper are conducted using Vampire [11], which is a resolution and superposition theorem prover running saturation algorithms. Vampire is available from http://www.vprover.org.

3 Symbol Elimination and Invariant Generation in Vampire

In a nutshell, the symbol elimination method of [9] works as follows. One is given a loop L, which may contain both scalar and array variables.

(1) In the *first phase*, symbol elimination tries to *generate as many loop properties as possible*. This may sound as solving a hard problem using an even harder problem, yet the method undertakes an easy way. First, it considers all (scalar and array) variables of L as functions of the *loop counter* n. This means that for every scalar variable a, a function $a(n)$ denoting the value of a at the loop iteration n is introduced. Likewise, for every array variable A a function $A(n, p)$ is introduced, denoting the value of A at the iteration n in the position (or index) p. Thus, the language of loop is extended by new function symbols, obtaining a new, extended language. Note that some loop properties in the new language are easy to extract from the loop body, for example, one can easily write down a formula describing the values of all loop variables at the iteration $n + 1$ in terms of their values at an iteration n, by using the transition relation of the loop. In addition to the transition relation, some properties of *counters* (scalar variables that are only incremented or decremented by constant values in L) are also added. Further, the loop language is also extended by the so-called *update predicates for arrays* and their properties are added to the extended language. An update predicate for an array A essentially expresses updates made to A and their effect on the final value of A. After this step, a collection Π of valid loop properties expressed in the extended language is derived.

(2) Formulas in Π cannot be used as loop invariants, since they use symbols not occurring in the loop, and even symbols whose semantics is described by the loop itself. These formulas, being valid properties of L, have a useful property: all their consequences are valid loop properties too. The *second* phase of symbol elimination tries to *generate logical consequences of Π in the original language of the loop*. Any such consequence is also a valid property of L, and hence an invariant of L. Logical consequences of Π are generated by running a saturation theorem prover on Π in a way that the theorem prover tries to eliminate the newly introduced symbols.

(3) The *third phase* of the method, added recently to symbol elimination [8], tries to remove invariants implied by other generated invariants.

It is important to note that (the first phase of) symbol elimination can be combined with other methods of program analysis. Indeed, any valid program property can be added to Π, resulting hopefully in a stronger set of invariants.

The rest of this section describes the details of how these three phases are implemented in Vampire.

3.1 Program Analysis in Vampire

In this section we briefly overview the program analysis phase of invariant generation in Vampire, introduced in [8].

The analyser works with simple programs of the form described in Section 2, and generates loop properties for the first phase of symbol elimination. The analyser works as follows. First, it extracts from the program all non-nested loops and performs the following steps on every such loop.

1. Find all loop variables and classify them into variables updated by the loop and constant variables.
2. Find counters, that is, updated scalar variables that are only incremented or decremented by constant values. Save properties of counters.
3. Generate update predicates of updated array variables and save their properties.

```
Loops found: 1          || Counter: a
Analyzing loop...       || Counter: b
---------------------   || Counter: c
while (a < m)           || Path:
  {                     ||   false: A[a] >= 0
    if (A[a] >= 0)      ||   C[c] = A[a];
      {                 ||   c = c + 1;
        B[b] = A[a];    ||   a = a + 1;
        b = b + 1;      || Path:
      }                 ||   true: A[a] >= 0
    else                ||   B[b] = A[a];
      {                 ||   b = b + 1;
        C[c] = A[a];    ||   a = a + 1;
        c = c + 1;      || Counter a: 1 min, 1 max, 1 gcd
      }                 || Counter b: 0 min, 1 max, 1 gcd
    a = a + 1;          || Counter c: 0 min, 1 max, 1 gcd
  }                     || 7. ![X1,X0,X3]:(X1>X0 & c(X1)>X3 & X3>c(X0)) =>
---------------------   ||    ?[X2]:(c(X2)=X3 & X2>X0 & X1>X2) [program analysis]
Variable: B: (updated)  || 6. ![X0]:c(X0)>=c0 (0:4) [program analysis]
Variable: a: (updated)  || 5. ![X0]:c(X0)<=c0+X0 (0:6) [program analysis]
Variable: b: (updated)  || 4. ![X1,X0,X3]:(X1>X0 & b(X1)>X3 & X3>b(X0))
Variable: m: constant   ||    => ?[X2]:(b(X2)=X3 & X2>X0 & X1>X2)[program analysis]
Variable: A: constant   || 3. ![X0]:b(X0)>=b0 (0:4) [program analysis]
Variable: C: (updated)  || 2. ![X0]:b(X0)<=b0+X0 (0:6) [program analysis]
Variable: c: (updated)  || 1. ![X0]:a(X0)=a0+X0 (0:6) [program analysis]
```

Fig. 2. Partial output of Vampire's program analysis on the `Partition` program of Table 3.

4. Save the formulas corresponding to the transition relation of the loop.
5. Create a symbol elimination task for Vampire by putting together all saved formulas and marking symbols that have to be eliminated.

Example 1. Figure 2 shows the output of Vampire corresponding to the first four steps of the program analysing process for the `Partition` program of Table 3. In the output, `![X]` (respectively `?[X]`) denotes $\forall$`X` (respectively, $\exists$`X`). The full example contains, apart from the loop shown in Figure 2, initialisation of some loop variables.

As shown in Figure 2, the program analyser of Vampire detects that variables `B`, `A`, `a`, `b`, `c` are updated in the loop, variables `A`, `m` are constants, and the variables `a`, `b`, `c` are counters.

Next, the path analysis of the program analyser reports that there are two program paths (listed in two blocks starting with `Path`), depending on the value of the test `A[a]` $\geq$ `0`. The values `min` and `max` denote the maximal and the minimal increment of the counter over all paths in the program. The value `gcd` is the greatest common divisor of all such increments.

Properties of counters are further generated, as listed between the lines enumerated by 1-7 of Figure 2. For example, Vampire generates axiom 1 expressing the following property: for *every loop iteration X0*, the value `a(X0)` of `a` at iteration `X0` is given by `a0+X0`, where `a0` denotes the initial value of `a`.

3.2 Theory Reasoning in Vampire

Standard resolution and superposition theorem provers are good in dealing with quantifiers but lack any support for theories, such as those of integers, reals, arrays, lists, etc. The standard way of adding a theory to such a theorem prover is by adding a first-order axiomatisation of the theory. There is no complete axiomatisation for all the above mentioned theories (assuming arbitrary quantifiers).

Adding incomplete axiomatisations is the approach used in [9]. The new version of Vampire [8] came further than [9] and also added integers as a data type in Vampire. The method of [8] is also the approach we follow in this paper. This means that integers can be used directly instead of representing them using, for example, zero and the successor function. Vampire "knows" several standard predicates and functions on integers: addition, subtraction, multiplication, successor, division, and standard inequality relations such as $\leq$. Since Vampire's users may not know much about combining arithmetic and first-order logic, automatic loading of relevant theories is taken care by Vampire. For example, if the user uses the standard integer addition function symbol $+$, then Vampire will automatically add an axiomatisation of integer linear arithmetic including axioms for $+$.

Generally, for loading existing theory axiomatisations of Vampire, the user should add to the input (in the TPTP syntax) a Vampire-specific declaration binding an input symbol to an interpreted theory symbol. For example, one can write:

```
  vampire(interpreted_symbol, geq, integer_greater_equal).
```

to declare that the input symbol `geq` denotes the inequality $\geq$ on integers. Given the above declaration, Vampire will add some theory axioms for this symbol to the input formulas. The user can also choose to use her own axiomatisation or to add more axioms to the axiomatisation loaded by Vampire.

The results reported in Section 4 show that Vampire's approach to reasoning with integers is good enough for proving properties of simple loops. However, the research into various approaches to reasoning with quantifiers and theories and their relative strength is still in its infancy and hindered by a lack of publicly available benchmarks.

3.3 Symbol Elimination in Vampire

If one just adds a collection of formulas obtained by program analysis to a theorem prover and expects the prover to generate consequences of these formulas using only a given subset of functions and predicates from the input, the result will most likely be disappointing. For example, suppose that p is a symbol that cannot occur in invariants, while q, r are symbols that can occur in them. Suppose that we are also given two clauses $p \vee q$ and $\neg p \vee r$. A theorem prover may decide to derive the invariant $q \vee r$ from these two clauses but may also decide not to do anything with them, depending on the term ordering and literal selection it uses (e.g. q and r might be selected before p).

The main ingredient of the symbol elimination technique in Vampire is the concept of a *well-colored derivation* [10, 7]. To define well-colored derivations (also called local proofs or split proofs) some predicate and/or function symbols are declared to have colors, while other symbols are uncolored. A symbol, term, literal or formula using a color are called colored, otherwise they are called transparent. A derivation is called *well-colored* if any inference can use symbols of at most one color. Any inference having at least one colored premise and a transparent conclusion is called a *symbol eliminating inference*.

Following the symbol elimination approach of [9], loop invariant generation can be thus be addressed using colors, as follows. One and the same color is assigned to all additional symbols introduced for formulating properties of loops (see Section 3), for example, the loop counter. All other symbols, that is the loop variables and the theory symbols, are transparent. A loop invariant is then a transparent formula describing a valid loop property. Since one is guaranteed that any transparent consequence of the input set of formulas is a valid loop property, the problem of invariant generation reduces to the problem of generating transparent consequences of this set. This means, in a way, that the colored symbols should be eliminated.

To make saturation more effective for deriving transparent consequences, the Knuth-Bendix term ordering used in Vampire was modified in [7], so that symbol weights are infinite ordinals and any colored ground term or atom is greater than any ground transparent term or atom.

Example 2. Consider the `Partition` program from Table 3. As presented in Figure 2, the program analyser of Vampire generates the following valid loop property:

`a(X0)=a0+X0` for every loop iteration `X0`.

However, this property cannot be used as an invariant as it makes use of the additional unary function symbol `a(X0)` denoting the value of the loop variable `a` at an iteration `X0`. However, since we declare the unary symbol `a` colored, Vampire tries to eliminate `a(X0)` from the set of valid properties generated by its program analyser, and derives, for example, the following transparent formula by a symbol eliminating inference: [3]

$\texttt{a-a0} \geq \texttt{0}$, where the constant `a` denotes the value of `a` at the end of the loop.

This property is a loop invariant, as it uses only the transparent symbols. Similarly,

$\texttt{B(X0,b(X0))} \geq \texttt{0}$ for every loop iteration `X0`

is a valid loop property, but not an invariant. However, by making the unary symbol `b` and the binary symbol `B` colored, Vampire generates the following invariant from this and other loop properties:

$$\texttt{0} \leq \texttt{X} < \texttt{b} \Rightarrow \texttt{B(X)} \geq \texttt{0},$$

where the unary symbol B and the constant b are the corresponding transparent symbols denoting the final values of loop variables with the same names.

3.4 Pruning Generated Invariants

Symbol elimination can generate invariants implied by other generated invariants. For example, any inference applied to two invariants gives an invariant. For this reason, [7] only stores invariants obtained by an inference having at least one colored premise, that is, a symbol-eliminating inference. However, even among conclusions of symbol-eliminating inferences there are typically many invariants implied by others.

To improve invariant generation, a new mode, called the *consequence-elimination mode*, was added to Vampire in [8]. In this mode, Vampire obtains a set S of clauses (i.e. invariants) as an input and tries to find its proper subset S' equivalent to S. In the process of computing S', Vampire is run with a small time limit. Naturally, one is interested in having S' as small as possible.

In the experiments reported in this paper, we made use of [8] and ran Vampire in the consequence elimination mode using four different strategies with a 20 seconds time limit. Our experimental results show that typically between 80% and 90% of all invariants obtained by symbol elimination are redundant, and hence discarded. It is usually the case that all or nearly all of the discarded invariants are discovered in a few milliseconds already by the first strategy.

[3] As described in [7], for every program variable v two transparent variables $v0$ and v are used, denoting the initial and the final values of this loop variable. Further, a colored unary function symbol v is introduced, such that $v(X)$ denotes the value of the loop variable v at iteration X.

Loop	Symbol Elimination within 1s					Symbol Elimination within 10s				
	♯ SEI	♯ Min SEI	% Redundancy	∀-Inv	∀∃-Inv	♯ SEI	♯ Min SEI	% Redundancy	∀-Inv	∀∃-Inv
`Initialisation`	15	5	67%	yes	no	40	5	88%	yes	no
`Copy`	24	5	79%	yes	no	25	5	80%	yes	no
`Find`	151	13	91%	yes	no	474	21	96%	yes	no
`Vararg`	1	1	0%	yes	no	1	1	0%	yes	no
`Partition`	166	38	77%	yes	yes	849	59	93%	yes	yes
`Partition_Init`	168	24	86%	yes	yes	692	127	82%	yes	yes
`Shift`	41	12	71%	yes	no	111	16	86%	yes	no

Table 1. Symbol elimination on programs from [2, 13], by running Vampire with 1 and 10 seconds of time limit.

3.5 Proving Invariants, Postconditions, and Assertions

The set of invariants and loop properties resulting from Vampire's program analysis and symbol elimination can be used to prove loop properties. Note that proving a loop property can be done in at least two different ways.

(a) First, we can add the negation of the property to the formulas obtained by program analysis and try to prove that the resulting set of formulas is unsatisfiable. It is easier than invariant generation since one does not have to take care of colors and can use arbitrary proofs, ordering and strategies, including goal-oriented ones.

(b) Second, one can prove that the property is an inductive invariant, which is a much simpler problem and can be reduced to proving a few formulas with respect to the theory.

However, both approaches assume that every loop is already annotated. Providing such annotations manually requires a considerable amount of work by highly qualified persons and thus often makes verification prohibitively expensive. Therefore, generation of invariants without using any annotations is invaluable in making verification and static analysis of programs economically feasible.

Evaluating the quality of an invariant generation technique is not easy since it cannot be measured using simple measures, such as the number of generated invariants or the speed of their generation. One can say that such a technique is powerful, if the set of invariants generated in small time implies the *intended* loop invariants, or invariants that humans would use to annotate this loop for verification purposes.

To evaluate our method, we used annotated code both from academic benchmarks and an industrial verification project. In this code every loop was *annotated by its intended property* in the form of loop invariants and/or postconditions. So we ran Vampire on the code as follows.

1. First, we generated a set of invariants using symbol elimination, as described in Sections 3.1–3.4;
2. Then we checked, also using Vampire, whether the intended loop property is implied by this set of invariants and whether the intended property described a postcondition, if the latter was provided.

Example 3. In the fourth column of Table 3, we show one of the intended invariant of the `Partition` program. This invariant follows from the two invariants generated by Vampire, which are presented in the fifth column of the table.

4 Experimental Results

The experiments described in this section were carried out using two benchmark suites. One is a collection of 6 loops taken from various research papers (Tables 3 and 1).

Loop Shape	♯ of Loops	Average ♯ of SEI	Average ♯ of Non-Redundant SEI	% of SEI Redundancy	∀-Inv	∀∃-Inv
Simple	33	168	18	89.3%	yes	no
Multi-path	5	340	46	86.4%	yes	yes

Table 2. Symbol elimination on programs sent by Dassault Aviation.

The other one is a collection of 38 loops taken from programs provided by Dassault Aviation. We used a computer with a 2 GHz Intel Core i7 CPU processor and 4GB RAM, and ran experiments using the Vampire version 0.6. The consequence elimination phase of Vampire was run with a 20 seconds time limit.

To analyse C programs, we had to extend Vampire by a C parser. Inputs to the parser are (large) C programs. After parsing, Vampire finds all loops in the program and checks, for each loop, if it is as given in (1) and thus can be analysed. Vampire outputs a set of loop invariants for each loop (1) under analysis. Figure 1 illustrates the invariant generation process within Vampire.

To use Vampire for generating invariants of arbitrary C code, one should use it in the program analysis mode as follows:

```
vampire --mode program_analysis < filename.c
```

4.1 Challenging Benchmarks

Table 3 describes the effect of symbol elimination on 6 programs. The names of the first 5 programs and their origins (that is, the papers where they were described) are given in column 1. The program given in the last row of Table 3 is taken from our own case studies, and illustrates the possibility of generating invariants for loops using read-*and*-write arrays. Columns 2 contains the number of invariants generated in 1 second, while column 3 the number of invariants that remain after consequence elimination.

Column 4 contains the intended invariant (or the invariant of interest): for this invariant we checked whether it is implied by invariants generated by Vampire, and if yes, show (in column 5) the subset of the generated invariants that imply the intended one. Checking that the intended invariant follows from the generated invariants was done using Vampire, so clauses in column 5 are those extracted from the corresponding Vampire proof. Invariants are generated by Vampire in a certain order. We used this order to enumerate clauses in column 5, since it gives the reader an idea how fast the required invariants were found. For example, `inv81` for the `Partition` example means that this invariant was the 82nd generated invariant (counting invariants starts from 0). Note that some of the formulas in this column have skolem functions introduced by Vampire's clausifier. For example, sk_1 denotes a skolem function. They can be de-skolemised to give invariants with quantifier alternations.

For this benchmark suite, *all the intended invariants turned out to be logical consequences of the invariants generated by Vampire,*. However, one of the intended invariants could not be proved by Vampire. Namely, for the `Shift` example, Vampire generated the invariant $\forall x(x \geq 0 \land x < a \Rightarrow A[x] = A[x+1])$, while the intended invariant was $\forall x(x \geq 0 \land x \leq a \Rightarrow A[x] = A[0])$. These two invariants are equivalent in arithmetic, however, to prove the intended one from the generated one in first-order logic one needs induction. By adding a simple induction axiom, specific to the `Shift` example, we could also prove the intended invariant.

We were also interested in checking the number of generated invariants depending on the time spent on invariant generation. Table 1 contains statistics about invariant generation with 1 and 10 seconds time limits on symbol elimination, respectively. It also shows the percentage of generated invariants shown to be redundant. As one can see, on the average over 80% of the generated invariants were proved to be redundant. Moreover, Table 1 reports whether quantified invariants with only universal quantifier ($\forall$-Inv) and with quantifier alternations ($\forall\exists$-Inv) have been generated. The relative size of the minimised set varies from example to example. Also, the intended invariant is always implied by the invariants generated in the first second (as reported in Table 3). For example, the intended (and rather complex) invariant for the `Partition` problem is implied by invariants 1 and 81 (see Table 3), while the first 166 invariants are generated in 1 second (Table 1). This suggests that the symbol elimination method generates increasingly sophisticated invariants, while natural and simple invariants are generated quickly.

4.2 Industrial Examples

We also ran Vampire's symbol elimination on 48 annotated array examples provided by Dassault Aviation. Since Vampire does not deal with pointers, we safely replaced pointers by arrays in 5 examples, and structures by arrays in 3 example loops. The 48 annotated array examples involve array copying, initialisation and shifts, and used arithmetical operations (e.g. addition, minus, plus, multiplication) and comparisons (e.g. greater, not equal) over the array content.

Vampire failed to find sufficiently strong invariants for 10 of these loops, for the following reasons. 6 loops were nested (all related to sorting algorithms) and thus cannot be analysed by the current version at all. Of the remaining 4 loops, two traversed sorted arrays using a logarithm-time search, one accessed the array using logically complex manipulation with array indexes, and the last one computed the sum of all array elements.

The results for the remaining 38 loops are analysed in Table 2. The first row of Table 2 shows the performance of Vampire on loops having only a single path, whereas the second row gives the results for multi-path loops. The second column shows the number of such loops. The third column gives the average number of invariants generated by Vampire with a 1 second time limit. The fourth and the fifth columns show, respectively, the number of invariants in the minimised set and the percentage of invariants proved to be redundant. The last two columns show whether any quantified invariants with universal quantifiers ($\forall$-Inv) (respectively with quantifier alternations $\forall\exists$-Inv) have been generated. We note that *for all 38 examples the intended invariants have been implied by the ones generated by Vampire*.

4.3 Analysis of Experiments

By studying the minimised sets of generated invariants we discovered that it still contains many redundancies and that many generated clauses could have been further improved by a better theory reasoning or algebraic simplifications.

Example 4. For the `Partition_Init` program, 692 invariants were generated in 10 seconds (see Table 1), out of which 127 invariants were kept in the minimised set. By

Loop	♯ SEI	♯ Min SEI	Inv of interest	Generated invariants implying Inv
`Initialisation` [13] $a = 0;$ **while** $(a < m)$ **do** $A[a] = 0; a = a + 1$ **end do**	15	5	$\forall x : 0 \leq x < a \Rightarrow$ $A[x] = 0$	`inv7:` $\forall x_0, x_1, x_2 : 0 \neq x_0 \vee x_1 \neq x_2 \vee$ $A(x_1) = x_0 \vee$ $\neg a > x_2 \vee \neg x_2 \geq 0$
`Copy` [13] $a = 0;$ **while** $(a < m)$ **do** $B[a] = A[a]; a = a + 1$ **end do**	24	5	$\forall x : 0 \leq x < a \Rightarrow$ $B[x] = A[x]$	`inv8:` $\forall x_0, x_1 : A[x_0] = B[x_1] \vee x0 \neq x_1 \vee$ $\neg a > x_0 \vee \neg x_0 \geq 0$
`Vararg` [13] $a = 0;$ **while** $(A[a] > 0)$ **do** $a = a + 1$ **end do**	1	1	$\forall x : 0 \leq x < a \Rightarrow$ $A[x] > 0$	`inv0:` $\forall x_0 : \neg a > x_0 \vee \neg x_0 \geq 0 \vee$ $A(x_0) > 0$

`Partition` [13]
$a = 0; b = 0; c = 0;$
while $(a < m)$ **do**
if $(A[a] >= 0)$
then $B[b] = A[a]; b = b + 1$
else $C[c] = A[a]; c = c + 1$
end if;
$a = a + 1$
end do
	166	38	$\forall x : 0 \leq x < b \Rightarrow$
			$B[x] \geq 0 \wedge$
			$\exists y : B[x] = A[y]$
				`inv1:`
				$\forall x_0 : \quad A(sk_2(x_0)) \geq 0 \vee$
				$\neg b > x_0 \vee \neg x_0 \geq 0$
				`inv81:`
				$\forall x_0 : \quad \neg b > x_0 \vee \neg x_0 \geq 0 \vee$
				$A(sk_2(x_0)) = B(x_0)$

`Partition_Init` [13]
$a = 0; c = 0;$
while $(a < m)$ **do**
if $(A[a] == B[a])$
then $C[c] = a; c = c + 1$
end if;
$a = a + 1$
end do
	168	24	$\forall x : 0 \leq x < c \Rightarrow$
			$A[C[x]] = B[C[x]]$
				`inv0:`
				$\forall x_0 : A(sk_1(x_0)) = B(sk_1(x_0)) \vee$
				$\neg c > x_0 \vee \neg x_0 \geq 0$
				`inv30:`
				$\forall x_0, x_1, x_2 : sk_1(x_0) \neq x_1 \vee$
				$x_0 \neq x_2 \vee \neg c > x_0 \vee$
				$\neg x_0 \geq 0 \vee C(x_2) = x_1$

`Shift`
$a = 0;$
while $(a < m)$ **do**
$A[a + 1] = A[a]; a = a + 1$
end do
	24	5	$\forall x : 0 \leq x \leq a \Rightarrow$
			$A[x] = A[0]$
				`inv5:`
				$\forall x_0, x_1, x_2 : x_0 + 1 \neq x_1 \vee$
				$A[x_0] \neq x_2 \vee A[x_1] = x_2 \vee$
				$\neg a > x_0 \vee \neg x_0 \geq 0$
				`inv13:`
				$\forall x_0, x_1 : \quad A[0] \neq x_0 \vee$
				$x_1 \neq 1 \vee A[x_1] = x_0 \vee$

Table 3. Invariant generation by symbol elimination with Vampire, within 1 second time limit.

further inspection of these 127 invariants, we noticed the following 2 invariants:

`inv30:`
$$\forall x_0, x_1, x_2 : \; sk_1(x_0) \neq x_1 \vee x_0 \neq x_2 \vee \neg c > x_0 \vee \neg x_0 \geq 0 \vee C(x_2) = x_1$$

`inv677:`
$$\forall x_0, x_1 : \qquad C(x_1) = sk_1(x_0 + 2) \vee 0 \geq x_0 \vee x_0 + 2 \neq x_1 \vee \neg c > x_0 + 2$$

When running Vampire only on these two formulas (without symbol elimination and consequence generation), Vampire proves in essentially no time that `inv30` $\Rightarrow$ `inv677`. Hence `inv677` is redundant, but could not be proved to be redundant using the consequence elimination mode with a 20 seconds time limit.

The above example suggests thus that further refinements of integer reasoning in conjunction with first-order theorem proving are crucial for generating a minimal set of interesting invariants. We leave this issue for further research.

Based on the experiments described here, we believe that we are now ready to answer the three questions raised in Section 1 about using symbol elimination for invariant generation.

1. [Strength] For each example we tried, (i) Vampire generated complex quantified invariants as conclusions of symbol eliminating inferences (some with quantifier alternations), (ii) using the invariants inferred by Vampire, the intended invariants and loop properties of the example could be automatically proved by Vampire in essentially no time. Hence, symbol elimination proves to be a powerful method for automated invariant generation.
2. [Time] Symbol elimination in Vampire is very fast. Within a 1 second time limit a large set of complex and useful quantified invariants have been generated for each example we tried.
3. [Quantity] Symbol elimination, even with very short time limits, can result in a large amount of invariants, ranging from one to several hundred. By interfacing symbol elimination with consequence elimination, one obtains a considerably smaller amount of non-redundant invariants: in practice, about 80% of invariants obtained by symbol elimination are normally proved to be redundant. We believe that the generated minimised set of invariants makes symbol elimination attractive for industrial software verification. It seems that the set of remaining invariants can be further reduced by better reasoning with quantifiers and theories.

5 Related Work

To the best of our knowledge, symbol elimination is a new approach that has not been previously evaluated. A related approach to symbol elimination is presented in [10] where theorem proving is used for generating interpolants as quantified invariants that imply given assertions. Predefined assertions and predicates are also the key ingredients in [4, 1, 3, 13] for quantified invariant inference. For doing so, predicate abstraction is employed to derive the strongest boolean combination of a given set of predicates [4, 3], or invariant templates are used over predicate abstraction [13] or constraint solving [1]. Abstract interpretation is also used in [2, 5], where quantified invariants are automatically inferred by an interval-based analysis over array indexes, without requiring user-given assertions. Unlike the cited works, in our experiments with symbol elimination to invariant generation, we generated complex invariants with quantifier alternations without using predefined predefined templates, predicates and assertions, and without using abstract interpretation techniques.

Quantified array invariants are also inferred in [6], by deploying symbolic computation based program analysis over loops. Although symbolic computation offers a more powerful framework than symbol elimination when it comes to arithmetical operations, all examples reported in [6] were successfully handled by using symbol elimination for invariant generation. However, [6] can only infer universally quantified invariants, whereas our experiments show that symbol elimination can be used to derive invariants with quantifier alternations.

6 Conclusions

We describe and evaluate the recent implementation of symbol elimination in the first-order theorem prover Vampire. This implementation includes a program analysis framework, theory reasoning, efficient consequence elimination, and invariant generation.

Our experimental results give practical evidence of the strength and time-efficiency of symbol elimination for invariant generation. Furthermore, we investigated quantitative aspects of symbol elimination.

We can answer affirmatively the question whether a computer program can automatically generate powerful program properties. Indeed, the properties generated by Vampire implied the intended properties of the programs we studied. However, our research also poses highly non-trivial problems. The main problem is the large number of generated properties. On the one hand, one can say that this is an accolade to the method that it can generate many more properties than a human would ever be able to discover. On the other hand, one can say that the majority of generated properties are uninteresting. This poses the following problems that can help us understand computer reasoning and intelligence better:

1. what makes some program properties more interesting than others from the viewpoint of programmers (or applications);
2. how can one automatically tell interesting program properties from other properties generated by a computer?

Answering these fundamental questions will also help us to improve program generation methods for the purpose of applications in program analysis and verification.

References

1. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *Proc. of VMCAI*, pages 346–362, 2007.
2. D. Gopan, T. W. Reps, and M. Sagiv. A Framework for Numeric Analysis of Array Operations . In *Proc. of POPL*, pages 338–350, 2005.
3. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *Proc. of POPL*, pages 235–246, 2008.
4. S. Gulwani and A. Tiwari. An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 379–392, 2007.
5. N. Halbwachs and M. Peron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI*, pages 339–348, 2008.
6. T. Henzinger, T. Hottelier, L. Kovacs, and A. Rybalchenko. Aligators for Arrays. In *Proc. of LPAR-17*, pages 348–356, 2010.
7. K. Hoder, L. Kovacs, and A. Voronkov. Interpolation and Symbol Elimination in Vampire. In *Proc. of IJCAR*, pages 188–195, 2010.
8. K. Hoder, L. Kovacs, and A. Voronkov. Invariant Generation in Vampire. In *Proc. of TACAS*, pages 60–64, 2011.
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. K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, pages 413–427, 2008.
11. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
12. A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*, volume 1. Elsevier Science, Amsterdam, 2001.
13. S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *Proc. of PLDI*, pages 223–234, 2009.