EPiC Series in Computing

Volume 44, 2017, Pages 16–32

Vampire 2016. Proceedings of the 3rd Vampire Workshop

Theory-Specific Reasoning About Loops With Arrays Using Vampire*

YuTing Chen[1], Laura Kovács[1,2], and Simon Robillard[1]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] TU Wien, Austria
yutingc@chalmers.se, simon.robillard@chalmers.se
laura.kovacs@tuwien.ac.at

Abstract

We describe new extensions of the first-order theorem prover Vampire for supporting program analysis and proving properties of loops with arrays. The common theme of our work is the symbol elimination method for generating loop invariants. In our work, we improve symbol elimination for program analysis in two ways. First, we enhance the program analysis framework of Vampire by simplifying skolemization during consequence finding. Second, we extend symbol elimination with theory-specific reasoning, in particular in the theory of polymorphic arrays, and generate and prove program properties over arrays. We illustrate our approach on a number of challenging examples coming from program analysis and verification. Our experiments show that, thanks to our improvements, programs that could not be analyzed before can now be verified with our method.

1 Introduction

Ensuring the correctness of software programs is critical in our software-controlled modern days. However, due to the growing size and complexity of software systems, manual verification of computer systems is not a viable task. Ideally, one would like to employ a computer program for automating the verification process. Over the past decades, many rigorous approaches have been proposed for automatically analyzing and verifying software programs, starting with the pioneering works of [7, 4, 5, 11, 3]. While powerful and practically useful, these methods have a number of limitations when applied to programs with loops/recursion. In this paper, we focus on the analysis and verification of programs with loops over arrays. We rely on the symbol elimination method in first-order theorem proving [9] and propose new extensions of symbol elimination for handling loops with nested conditionals. In particular, our work allows generating and proving first-order program properties with quantifier alternation and exploits theory-specific reasoning in the polymorphic theory of arrays.

*This work has been carried out as a master thesis project at the Chalmers University of Technology. The work has been partially supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014- TheProSE, the Swedish VR grant D0497701 and the Austrian research project FWF S11409-N23.

As our motivating example, consider the following small imperative program, written in a Java-like syntax:

```
int [] A, B, C;
int a, b, c;
a = 0; b = 0; c = 0;
while (a < A.length) {
  if (A[a] >= 0) {
    B[b] = A[a];
    b = b + 1; a = a + 1;
  }
  else {
    C[c] = A[a];
    c = c + 1; a = a + 1;
  }
}
```

Figure 1: An imperative loop over three arrays. This program, which we refer to as the `partition` example, separates the non-negative array elements from the negative ones by element-wise copying array elements into two resulting arrays.

The program in Figure 1 copies the non-negative integer values of array `A` into array `B`, and the negative ones into array `C`. While the program is a relatively simple one, it employs arithmetic over array indexes and implements array operations depending on arithmetic conditions over the array content. Capturing the behavior of this program therefore requires reasoning in the first-order theory of arrays and integer arithmetic and needs to establish program properties, i.e. invariants, that hold at arbitrary loop iterations. For example, some of the properties fulfilled by our running example are as follows:

1. Each element of the array `B`, starting from `B[0]` to `B[b-1]`, is a non-negative integer and equal to one element in the array `A`.
2. Each element of the array `C`, starting from `C[0]` to `C[c-1]`, is a negative integer and equal to one element in the array `A`.
3. Each non-negative element of the array `A` is also one element in the array `B`.
4. Each negative element of the array `A` is also an element in the array `C`.
5. Any array element positioned in `B` (respectively, `C`) after the final value of `b` (respectively, `c`) is not updated during the loop.

The above enumerated properties capture the algorithmic nature of the loop. Note that these properties hold at every loop iteration, and hence they are *loop invariants*. However, inferring these properties requires deep understanding of the program semantics and reasoning in full first-order logic; for example, formalizing the first property needs quantifier alternation over the array elements of `A` and `B`. Nevertheless, knowing such invariants can prevent programmers introducing errors while making changes in the code and hence such deriving such properties fully automatically is invaluable in automating program analysis and verification.

In this paper we address the challenge of generating quantified invariants of loops with arrays. We rely on the work of [9] where first-order theorem proving has been proposed not only to prove, but also generate program properties. In particular, the work of [9] introduced the *symbol elimination* method in first-order theorem proving, becoming the first ever approach able to derive loop invariants with alternating quantifiers in a fully automatic way. Symbol elimination is already supported in the first-order theorem prover Vampire [10]. Symbol elimination was further extended for its application in program analysis in [2], enabling symbol elimination to also prove program correctness by using the program specification. The approach in [2] is interfaced with real world programming languages via an intermediate language called *simple guarded command language* and is used within the KeY verification framework [1]. When evaluating symbol elimination on 20 challenging examples, [2] reports that 11 benchmarks were proven correct. When analyzing the other failing examples, one sees that more sophisticated program analysis on top of symbol elimination is required in order to guide the first-order theorem prover in the process of invariant generation.

Motivated by the results of [2], in this paper we enhance symbol elimination in program analysis in two ways. First, we propose a simplified treatment of array properties, whenever updated positions in arrays can be described by monotonically increasing/decreasing functions. Second, we extend symbol elimination with theory-specific reasoning. Unlike [2], we do not treat arrays as uninterpreted functions but exploit the read/write semantics of array operations and use the polymorphic theory of arrays [8] for generating and proving program properties over arrays. Our work is implemented in Vampire and evaluated on many examples from program analysis and verification. When compared to [2], our experiments show that some examples that could not be handled before can now be proved by our method.

The rest of this paper is organized as follows. After fixing the notation and terminology in Section 2, we describe symbol elimination in program analysis in Section 3. Our extension to symbol elimination for program analysis is presented in Section 3.3.2. Next, Section 3 describes theory-specific reasoning in symbol elimination and details our contribution in Section 4.1. Our experimental results are summarized in Section 5 and Section 6 concludes the paper.

2 Preliminaries

This section fixes the notation and terminology used further in the paper. The material of this section is mainly based on [9, 2], adapted to our setting.

2.1 Programming Model

Our programming model is expressed using a simple guarded command language. Unlike the generic guarded command language of [6], our guarded command language is deterministic. We support both scalar variables and array variables with two primitive types, `int` for the integers and `bool` for the booleans. Standard arithmetical functions symbols such as $+$, `-` ,$*$, $/$ and predicate symbols $\leq$ and $\geq$ are also used. Given an array variable `A`, we denote `A[p]` as the array element of `A` at position `p`, corresponding to the result of an array read/select operation. We consider loops with nested conditionals; nested loops are not yet supported. A program loop is represented by a loop condition and an ordered collection of guarded statements representing the loop body. A loop condition is a quantifier-free boolean formula, while each guarded statement consists of a guard and a non-ordered collection of parallel assignment statements. The ordered collection of guarded statements is checked in sequential order and each guard is assumed to be mutually exclusive with other guards. Further, the non-ordered collection of parallel assignment statements are assumed to respect the following rules:

1. In case of scalar variable assignment, there cannot exist two parallel assignments to the same scalar variable within the collection.

2. In case of array variable assignment (which corresponds to an array write/store), the same position cannot be assigned to two values. That is, if two array assignments `A[i] := e` and `A[j] := f` occur in a guarded statement, the condition $\mathtt{i} \neq \mathtt{j}$ is added to the guard.

Pre- and post-condition can be specified apart from the loop, using *requires* and *ensures* keywords respectively. Conditions are boolean formulas over program variables and quantifiers are allowed. Finally, all types must be declared upfront of the program.

The semantics of our simple guarded language is defined using the notion of a program state. Each scalar and array variable is mapped to a value of correct type by the program states. In our setting, a single program state corresponds to each loop iteration. Assuming n denotes the upper bound of loop iterations, at any loop iteration i with $0 \leq i < n$, we have σ_i as the program state. σ_0 and σ_n represent the initial state and final state of the loop, respectively. Once the guard is valid, the associated collection of parallel assignment statements will be applied simultaneously to the program state. For example, executing the guarded statement:

```
true -> x = 0; y = x;
```

in a program state where `x = 1` holds will result in a new program state with `x = 0 & y = 1`.

Our programming model is illustrated in Figure 2. All variables, including arrays and scalars, are declared upfront. The declaration is followed by user specified pre-conditions using the keyword `requires`. In this example, all array index variables `a`, `b` and `c` are initialized with zero (value equality is denoted by `==`). In addition, the scalar variable `alength`, denoting the length of the array `A`, is required to be positive. The pre-conditions are followed by the user-specified post-conditions, using the `ensures` keyword. While in this particular example, both pre- and post-conditions are specified by the user, their presences are not required for our approach to generate loop invariants. Our approach can however utilize the program specification to provide better user experience, such as invariant filtering and proof of correctness.While most numerical operations take the usual representing syntax, logical implication uses the `==>` syntax. Inside the loop body, each guarded command starts with double-colon followed by the quantifier-free boolean guard. The guard is separated with the collection of parallel assignments by `->`. Finally, each assignment is terminated by the semicolon.

```
int [] A, B, C;
int a, b, c, alength;

requires a == 0;
requires b == 0;
requires c == 0;
requires alength > 0;

ensures forall int i, 0 <= i & i < b ==> B[i] >= 0;
ensures forall int i,
        exists int j, 0 <= i & i < b ==>
                                0 <= j & j < a & B[i] == A[j];

while (a <= alength) do
  :: A[a] >= 0 -> B[b] = A[a]; a = a + 1; b = b + 1;
  :: true -> C[c] = A[a]; a = a + 1; c = c + 1;
od
```

Figure 2: Our running example from Figure 1, expressed using our simple guarded command language. The second guard *true* functions as the final conditional guard, similar to the *otherwise* construct found in other languages.

2.2 Invariant Generation Process

We now overview the main steps of symbol elimination for generating program properties, in particular loop invariants. Symbol elimination for invariant generation is composed of two main steps: *static program analysis* and *logical inferences using first-order theorem proving*. The input program, expressed in the simple guarded language of Section 2.1, is first passed to static program analysis. The analysis aims to statically find out valid relations between program variables and formulates these relations in first-order logic. This analysis does not actually execute any part of the input program, hence it is as a *static* analysis. Let P denote the set of first-order formulas derived in this step. The second part of invariant generation by symbol elimination uses P as input to a first-order theorem prover, in particular Vampire, and tries to saturate P, that is generate all logical consequences of P. Any logical consequence is a valid loop property as it is implied by P; in addition, the logical consequences that use only program variables are invariants. In our work, we extended both parts of symbol elimination: we improved static analysis over arrays and enhanced consequence finding by theory-specific reasoning in the polymorphic theory of arrays.

3 Program Analysis

We now describe the program analysis step of symbol elimination for generating loop invariants.

3.1 Preprocessing of Invariant Generation

Given a program as described in Section 2.1, in the first step of symbol elimination static analysis of the program is carried out in order to express the program semantics as a collection of first-order formulas that can further be used by saturation theorem proving. In this step of program analysis, we rely on the previous symbol elimination framework of [9, 2], but make a simple extension to this framework when it comes to reason about array properties. In more detail, within program analysis we perform the following steps:

- Lex and parse the input program. In our work, we used the standard C++ scanner generator flex and parser generator Bison for the simple guarded command language of Section 2.1.
- Express the (user-given) pre- and post-conditions of the input program as first-order formulas. On the implementation level, these formulas are formulated as first-order formulas using Vampire terms. We use the sorts of program variables in the first-order formulas to be generated and turn program variables into *extended expressions*, that is into functions of the loop counter. Extended expressions help us capturing the program state in which an update of a particular variable occurs – see Section 3.2 for details.
- Perform static analysis over the input program and extract first-order properties involving extended expression over scalar and array variables. When compare to the previous works of [9, 2], one contribution of our work comes with an extension of program analysis over array variables: under the additional condition that updates to array variables can be

described by monotonic function, we generate a new program property over arrays. We call this property the *monotonic indexing property*, as described later in Section 3.3.4.

- Use the collection of generated first-order properties as typed first-order formulas and feed further these formulas as input to a saturation theorem prover (second step of symbol elimination). Following the work of [2], we also use the (user-given) pre- and post-conditions in the consequence finding (invariant generation) part of symbol elimination. Therefore, program specifications are also given as input to the theorem prover.

3.2 Extended Expressions

The goal of invariant generation is to generate properties that hold at an arbitrary iteration of the loop. For doing so, we extend the language of our input program with an additional variable, called the loop counter, in order to reason about loop iterations. With a loop counter at hand, we can then reason about program states and express which variables are update in which program state. Using the loop counter, program variables become functions of the loop counter; these functions are called *extended expressions* and we refer to the language of extended expression as the *extended language*. Note however that loop properties formulated in the extended language are not necessarily loop invariants as they use symbols (extended expressions) which are not part of the input language of the program. Therefore, the goal of the consequence finding step of symbol elimination is to generate logical consequences of the loop properties expressed in the loop language such that the generated logical consequences do not contain extended expressions.

In what follows, we call program properties (not just invariants) containing only symbols from the input program as *program assertions*. Note that any program assertion is also a property in the extended language, but not every property expressed in the extended language is also a program assertion. In the sequel, we first present our program assertion language and then describe the loop properties we generate in the extended language.

3.2.1 Assertion Language

The assertion language is meant to express loop properties as classical first-order logic formulae. For each scalar variable v of type τ used in the program, we create two corresponding symbols in our extended language: $v : \tau$ and $v_{init} : \tau$. The introduction of $v_{init} : \tau$ allows us to state the initial value of the variable v, while the $v : \tau$ captures the final values of v (after the last loop iteration). An interpretation in our assertion language maps symbols to their values; the interpretations depend though on a given program state α.

An example of a program assertion assertion can be obtained from our running example in Figure 2. As the values of a, b, c are assumed by the `requires` statements, one can translate these requirements into the following program assertion:

$$a_{init} = b_{init} = c_{init} = 0$$

In the previous work [2], array variables are captured in the assertion language using uninterpreted function symbols of type $\mathbb{Z} \to \tau$, where $\mathbb{Z}$ denotes the integer sort of array indexes. In our work, we extended [2] by a proper treatment of arrays instead of treating arrays as uninterpreted functions (see Section 4). Finally, pre- and post-conditions, which are in the first-order logic, are trivially expressible in our using the assertion language.

3.2.2 Extended Language

Extended expressions capture the behavior of program variables within program states. For each variable v of type τ, the extended expression of v is a function of type $\mathbb{Z} \rightarrow \tau$ capturing the value of v at various loop iterations. While the extended expression of v can take arbitrary integer arguments, in practice arguments are non-negative integers as loop iterations are non-negative integers as well. In the sequel, v^i denotes the application of the extended expression of v on the ith loop iteration, and represents the value of v in the program state α_i (where α_i is the program state corresponding to the ith loop iteration). In [2], the extended expressions of array variables have an additional argument denoting the array index at which the array is accessed. This is not the case in our work as we treat arrays as proper arrays and not as uninterpreted functions; that is, in our work, arrays are accessed via the interpreted `select(array, index)` operation.

Finally, the extended language of the program includes an additional symbol, denoted by n, representing the upper bound of loop iterations. In what follows, we consider n fixed and give all definitions relative to it.

We now list some examples of extended expressions. Consider the following property expressed in the extended language:

$$v^{i+1} = v^i + 1,$$

which states that the value of variable v at iteration $i + 1$ is the successor of the value of v at the ith loop iteration. Such a property holds, for example, in the case of variable a in our running example from Figure 2.

Using extended expression, one can capture the relation between the values of a variable at different program states. As already mentioned, we treat two program states, namely the states corresponding to the first and final loop iteration, differently than the other states and name these states as α_{init} and α_n, respectively . The following example expresses that the final value of variable v is the same as its initial value:

$$v^n = v^0$$

Following the semantics of extended expressions, the following equalities between the assertion language terms and the extended expressions are naturally true:

$$\begin{gathered} v_{init} \equiv v^0 \\ v_n \equiv v^n \\ \texttt{select(}\ A^{init}\ \texttt{, p)} \equiv \texttt{select(}\ A^0\ \texttt{, p)} \\ \texttt{select(}\ A\ \texttt{, p)} \equiv \texttt{select(}\ A^n\ \texttt{, p)}\ , \end{gathered}$$

where the capital letter variable A denotes an array variable.

For simplicity, in the sequel we call properties expressed using the extended expressions as *extended loop properties*. These extended loop properties are central for symbol elimination. During invariant generation by symbol elimination, extended loop properties are first inferred and then logical consequences of these extended loop properties are generated using saturation theorem proving.

We next detail how extended loop properties are generated in our work.

3.3 Extracting Loop Properties

Using extended expressions, we can capture the values of variables at various program states/loop iterations. Our next step is to generate valid extended loop properties. For doing so, we examine updates made to scalar and array variables and formulate properties on variable updates as first-order formulas with extended expressions.

Let us also note, that in addition to the generated extended loop properties, user-given pre- and post-conditions are also translated into extended expressions. We exemplify the translation of program specifications into extended loop properties below. Consider the pre-condition:

```
requires forall int i, 0 <= i & i < alength ==> A[i] > 0.
```

This pre-condition is translated into the following extended loop property:

$$(\forall i)(0 \leq i < \mathit{alength} \Rightarrow \mathit{select}(A(0), i) > 0)$$

In the remaining of this section, we first overview the program analysis framework of [9, 2] for extracting extended loop properties and then describe our improvement on this framework. Our contribution is a simpler and more efficient treatment of array variables, called the monotonic indexing property of array updates.

3.3.1 Static Properties of Scalar Variable

Given a scalar program variable `v`, we call it *increasing* or *decreasing* if the following extended property holds:

$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} \geq v^i \text{ /* v is increasing */}$$
$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} \leq v^i \text{ /* v is decreasing */}$$

where *iteration* is the set of loop iteration values, that is, $\mathit{iteration} = \{0, \dots, n\}$. We consider *iteration* part of the extended loop language.

A scalar variable which is increasing (decreasing) is also called *monotonic*. The analysis for monotonicity can be achieved via light-weight program analysis, such as ensuring that every program assignment to `v` is of the form `v = v + e`, where `e` is a non-negative integer.

Apart from monotonicity checking, a scalar variable v is called *strict* if the following extended properties hold:

$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} > v^i \text{ /* v is strictly increasing */}$$
$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow v^{i+1} < v^i \text{ /* v is strictly decreasing */}$$

Since our guarded command language only supports integer sort for numerical scalars, we further call an increasing scalar variable v *dense* if the following property holds:

$$\forall i \in \mathit{iteration}, 0 \leq i < n \Rightarrow |v^{i+1} - v^i| \leq 1 \text{ /* v is dense*/}$$

Exploiting the definition of monotonic, strict and dense variables, extended properties relating variables values at different iterations can be expressed. The generated extended properties are summarized in the following table:

Table 1: Added static properties for scalar variables

Scalar variable v is:	We generate the extended property':
[increasing, strict, dense]	$\forall i, v^i = v^0 + i$
[increasing, strict]	$\forall i, \forall j, j > i \Rightarrow v^j > v^i$
[increasing, dense]	$\forall i, \forall j, j \geq i \Rightarrow v^i + j = v^j + i$
[increasing]	$\forall i, \forall j, j \geq i \Rightarrow v^j \geq v^i$

Finally, if a variable v is not updated at any loop iteration, one can simply treat v as a constant symbol throughout all program states.

3.3.2 Update Properties of Array Variable

For simplicity, in what follows we denote array variables by capital letters, such A, B, C. Given an array variable A, our approach analyzes the conditions which trigger the updates on array elements array at position `p` by the value `v` at the loop iteration `i`. We encode these conditions and their corresponding properties in additional properties denoted as $upd_A(i, p, v)$ and called *update predicates*. We also consider the predicate $upd_A(i, p)$ for encoding that the array A has been updated at position `p` and loop iteration `i`; we refer to the predicate $upd_A(i, p)$ also as an update predicate of array A. Update predicates are extended expressions and are part of our extended language. Hence, extended loop properties may contain update predicates.

The update predicates of arrays are used to generate the following properties over array variables:

- If the array `A` is only updated once throughout all possible program states, and the update happens at index `p` with the value `v`, then this value is associated with the final value at the same index during the last loop iteration.

$$\begin{aligned}(\forall i \in &iteration, j \in iteration, p \in index, v)\\ &(upd_A(i,p,v) \wedge (upd_A(j,p) \Rightarrow j \leq i)) \Rightarrow\\ &select(A(n), p(n)) = v)\end{aligned} \tag{1}$$

3.3.3 Array Non-Update

Given an array variable `A` which has not been updated during any loop iteration `i`, the approach of [9, 2] generates the following extended loop property over A:

$$\forall j, A(i+1, j) = A(i, j),$$

where $A(i, j)$ is the extended expression of A capturing the value of the array element $A[j]$ at iteration i. In our work we do not treat however arrays as uninterpreted functions (i.e. $A(i, j)$) but use the polymorphic theory of arrays to represent array operations. Based on the array theory, a naive translation of the property above into our work would yield:

$$\forall j, select(A(i), j) = select(A(i+1), j)$$

This property should however not be added as it is already assumed in the theory of array. Without array updates, that is without array `store` operations, the elements in an array are assumed to be unchanged from one program state to the next. Therefore, compared to [9, 2], we do not consider this extended loop property in our work.

3.3.4 Monotonic Indexing

We now describe our contribution for improving the program analysis framework of [9, 2]. Our contribution comes with a so-called monotonic indexing property, which is an extended loop property over arrays. The intuition behind this monotonic indexing property originated from the monotonicity of indexing scalars. If a scalar variable v is strict and monotonic, the value v^i of v at iteration i will only increase or decrease in later iterations. Therefore, once the value v^i is visited/updated, v variable will not have this value in later iterations. In our work we use this observation on strict and monotonic array index variables and derive extended loop properties with array update predicates.

Suppose the array variable `A` is only accessed by the array index variable x throughout the entire program, where x is monotonic and strict. In our framework, we derive the monotonic indexing property as follows:

1. Assume x is *monotonic* and *strictly increasing*. The following extended loop property over x holds:

$$(\forall\, \alpha \in \mathit{program\, state})\, (\forall\, \beta \in\ \mathit{program\, state})$$
$$(\beta > \alpha \Rightarrow x^{\beta} > x^{\alpha})$$

2. Suppose the array `A` is updated by the program statement `A[x] := val` at iteration α. For any iteration β, the implication $\beta > \alpha \Rightarrow x^{\beta} > x^{\alpha}$ holds. As argued above, we never revisit array `A` at index x^{α} in later iterations β. This means, that `val` is the final value of `A[x]`. Hence, we generate the following extended loop property:

$$A^n[x^{\alpha}] == A^{\alpha}[x^{\alpha}] == \mathit{val}$$

   Formulating this property using the theory of arrays, we have the so-called *monotonic indexing property*:

$$\mathit{select}(A(n), x(\alpha)) == \mathit{select}(A(\alpha), x(\alpha))$$

   With the same reasoning, we generate a similar monotonic indexing property for arrays whose array index variables are monotonic and strictly decreasing.

Consider our running example from Figure 2. In both program branches, the scalar variable `a` is increased by one and hence `a` is monotonic and strictly increasing. Further, we know that the array `A` is only accessed by `a`. All of the requirements of our monotonic indexing property are thus satisfied and the monotonic indexing property over the array variable `A` is:

$$\mathit{select}(A(n), a(\alpha)) == \mathit{select}(A(\alpha), a(\alpha))$$

In our work, we apply a similar reasoning monotonic increasing/decreasing array indexes which are not strict, and derive monotonic indexing properties of the corresponding arrays. In our example from Figure 2, we generate monotonic indexing properties of the arrays `B` (and similarly to `C`), as follows:

$$\mathit{select}(B(n), b(\alpha)) == \mathit{select}(B(\alpha), b(\alpha))$$

where α represents the program state in which the update to `B` takes place.

4 Invariant Generation by Theory-Specific Reasoning

Once program analysis is finalized, a set P of extended loop properties are generated. In the second step of invariant generation by symbol elimination, we saturate P using a saturation theorem prover in order to generate logical consequences of P. In our work, we rely on the Vampire theorem prover and apply the saturation algorithm of Vampire to generate logical consequences of P. To this end, we support three different settings of consequence finding with Vampire:

1. We try to saturate P using Vampire and generate logical consequences of P. Whenever a logical consequence of P is generated as a program assertion, that is without extended expression, a loop invariant is obtained by Vampire. Hence, for invariant generation by symbol elimination, symbols from the extended language that are not part of the input program language are eliminated and invariants are inferred as logical inferences of P.
2. The set of invariants generated by simply saturating P using Vampire is, in general, infinite as we generate logical consequences of a set of valid first-order formulas. Many of the generated invariants might therefore be not of much use when it comes to prove the user-specified post-conditions of the loop. Therefore, in our work, we can use the post-condition to guide Vampire while saturating P and generate/output invariants from which the post-condition can be proven.
3. Finally, the user-specified post-condition can be used to directly prove program correctness, without the burden of eliminating symbols from P and generate invariants as logical consequences of P. That is, Vampire uses P as input formulas and the user-given post-condition is as the conjecture to be proven, and tries to refute the post-condition while saturating P. This approach is already supported in [2].

In the rest of this section, we explain the logical inferences performed by Vampire and the symbol elimination method for invariant generation. Vampire is a fully automatic first-order theorem prover, hence it requires no user guidance during proving/saturation. This allows our invariant generation approach to be fully automated, the user is not required to provide any input or steer the generation of formulas during runtime. Vampire uses a logically sound inference system, i.e. the superposition inference system. This means that the generated logical consequences of P are also valid loop properties. The overall performance of Vampire for invariant generation is significantly affected by the input P. Additionally, Vampire has a sophisticated internal architecture and the performance of Vampire very much depends on what options are used for proving. As we describe the process of consequence finding in Vampire, we will also briefly explain critical concepts related to the choice of options.

4.1 First-Order Reasoning about Array Properties

The recent work of [8] introduces an extension of first-order logic, called FOOL. FOOL extends first-order logic with first-class boolean variables and polymorphic arrays. FOOL is already supported by Vampire and hence can be used for symbol elimination. In addition, FOOL allows the use of `if-then-else` and `let-in` and the theory of polymorphic arrays is encoded using array theory axioms with select/store operations.

In our work, we exploited the polymorphic theory of arrays built-in Vampire. We used this theory for both generating extended loop properties and later inferring loop invariants by symbol elimination. In the previous work of [2], array variables have been encoded as uninterpreted functions and one cannot explicitly express the differences between array reads/selects and array writes/stores. In our work, arrays are now encoded as array theory symbols and we distinguish between the read/select and write/store operations over arrays. With this improvement, the semantics of array operations is captured. Let us exemplify some differences between [2] and our treatment of arrays:

- In [2], the array assignment of `a[i] = b[j]` at iteration `k` is internally encoded as:

```
equals(a(k+1,i(k)),
       b(k  ,j(k)))
```

where the function `equals()` creates internal equality over two terms for Vampire.

In our work, with the array encoding, we represent the above array assignment statement as:

```
store(a(k+1),
      i(k),
      select(b(k),j(k)))
```

In our work, whenever a `select` and/or `store` operation over arrays is encountered, the array theory axioms are automatically loaded within Vampire. The polymorphic array fragment of FOOL is more general than our simple guarded command language when it comes to the sorts of array indexes: FOOL supports arbitrary sorts of array indexes, whereas in our work array indexes are always (non-negative

5 Experimental Results

Our work is implemented in Vampire and evaluated on 20 interesting examples, including the benchmarks of [2]. In this section we report on our experimental results and compare our results with [2]. All our results are collected using a computer with quad-core i7 CPU equipped with 16GB of RAM. For a fair comparison, we reproduced the results of [2] on our hardware setup.

Results of all test cases Our benchmarks were annotated with pre- and post-conditions. We evaluated invariant generation by symbol elimination in Vampire, but we were also interested in proving the specified post-condition. For this reason, we used the pre- and post-conditions to guide Vampire in the process of invariant generation and tried to prove the correctness of the post-conditions during consequence finding.

Table 2 shows the experiment result without array theory reasoning and monotonic indexing property; these table summarizes the results we obtained by evaluating the work of [2] on our benchmarks. Table 3 reports on the results we obtained using our implementation, where array theory reasoning and the monotonic indexing property is used. All test case, in both cases, are performed with the time limit set to 300 seconds. For both tables, the first column names the benchmark problem. Column Δ_{direct} shows the time (in seconds) Vampire needed to prove the user-given post-condition while generating logical consequences and invariants of the extended loop properties. An empty entry in the Δ_{direct} column means that Vampire failed to find a proof. Finally, the total number of generated clauses during consequence finding is given the third column: whenever Vampire finds a proof, the number of generated clauses corresponds to the number of generated clauses until the proof was found. Whenever Vampire failed to find a proof, the number of generated gives the number of clauses generated by Vampire within the 300 seconds time limit.

Table 2: Reasoning **without** the array theory and monotonic indexing property

Test case	$\Delta_{direct}(sec)$	generated clauses
absolute	0.374	2095
copy	0.057	495
copyOdd	0.208	1571
copyPartial	0.047	426
copyPositive		530669
find		412821
findMax		324456
init	0.052	415
initEven		430518
initNonConstant	0.117	909
initPartial	0.060	495
inPlaceMax		362783
max	0.348	2140
mergeInterleave		376322
partition		622830
partitionInit		488387
reverse	0.079	593
strcpy	0.048	373
strlen	0.019	139
swap		812284

Table 3: Reasoning with the array theory and monotonic indexing property

Test case	$\Delta_{direct}(sec)$	created clauses
absolute	0.484	2614
copy	0.079	654
copyOdd	0.181	1098
copyPartial	0.104	800
copyPositive	46.238	89280
find		413352
findMax		398548
init	0.069	592
initEven		391735
initNonConstant	0.128	940
initPartial	0.069	593
inPlaceMax		530098
max	0.481	2634
mergeInterleave		543746
partition	97.519	210837
partitionInit	28.217	72989
reverse	0.098	733
strcpy	0.081	538
strlen	0.031	168
swap	11.218	61786

When compared to [2], Table 3 shows that our work managed to solve four examples that could not be proven before in [2]: `copyPositive`, `partition`, `partitionInit`, and `swap`. We note that all examples that were proven in [2] were also proven in our work. For the `partition`, `copyPositive`, and `partitionInit` examples, our contribution on the monotonic indexing property was the key part, whereas for the `swap` our contribution on reasoning with the array theory was essential.

5.1 Case Study: swap

```
int [] a, b, olda, oldb;
int i, alength, blength;

requires blength == alength;
requires i == 0;
requires forall int i, 0 <= i & i < alength ==> a[i] == olda[i];
requires forall int i, 0 <= i & i < blength ==> b[i] == oldb[i];

ensures forall int i, 0 <= i & i < blength ==> a[i] == oldb[i];
ensures forall int i, 0 <= i & i < blength ==> b[i] == olda[i];

while (i < alength) do
:: true -> a[i] = b[i]; b[i] = a[i]; i = i + 1;
od
```

Figure 3: Test case: `swap`

We now look into more detail for the `swap` example. The program takes two integer arrays of equal length and element-wise swaps the two arrays, as shown in Figure 3. The pre-conditions requires equal length of the input arrays and keeps a copy of the values inside each array (`olda` and `oldb`). The post-conditions ensures that every element in the modified array `b` is indeed an element-wise equal to `olda`. This test case has rather straightforward semantics for human reader, yet its algorithm heavily relies on the array operations `select` and `store`. While this example could not be proved by the approach of [2], using our work, we managed to prove the post-conditions thanks to theory-specific reasoning over arrays.

```
38747.  C1 $select(oldb,sK6) = $select(a,sK6)
| $lesseq(alength,$sum(-1,sK6))
<- {80, 185, 192, 194}
[subsumption resolution 38698,30268]

38749.  C1 $lesseq(alength,$sum(-1,sK6))
<- {19, 80, 185, 190, 192,194}
[subsumption resolution 38747,30169]

38750. 262 | 19 | ~80 | 185 | ~190 | ~192 | ~194
[AVATAR split clause 38749,30139,30132
,30125,30101,1660,307,38738]

63634.  C0 $false [AVATAR sat refutation
48074,47723,371,180,186,30241,340,341,3414,
63200,339,38750,30127,30141,30134,381,208,
214,44449,309,1780,1681,330,48112,45944,48885,
51852,47081,3177,3247,332,323,48525,55659,331,
63319,1668,1696,316]
```

Figure 4: Critical proof steps in the test case: `swap`

Some critical steps in the successful proof of correctness are contributed by the array theory axioms, as shown in Figure 4. This shows the array theory derived inferences are indeed used in the final proof by refutation. Using our approach, Vampire can automatically generate the following loop invariant, written in the Vampire native format:

```
! [X2 : $int] : ($select(oldb,X2) = $select(a,X2) |
              ~($less(X2,blength) & $lesseq(0,X2))).
```

Writing the above invariant using standard first-order logic notation, we have the invariant in conjunctive normal form:

$$\forall i \in Int\,, oldb[i] = a[i] \lor \neg((i < blength) \land (0 \leq i))$$

which is essentially states the following property:

$$\forall i \in Int\,, (0 \leq i < blength) \Rightarrow oldb[i] = a[i]$$

6 Conclusion

We described improvements to invariant generation by symbol elimination and demonstrated the practical impact of our work on a collection of example. Our work is implemented using Vampire and our contributions come with theory-specific reasoning in the polymorphic theory of arrays and enhancing the program analysis framework of Vampire with a special treatment of arrays with monotonic indexes. Our approach of invariant generation using Vampire requires no user guidance nor any predefined templates of invariants.

Future Work Our work can further be extended by a number of ways.

1. **Boolean array test cases and experiments**: using the FOOL framework, boolean arrays can be incorporated into our array theory reasoning approach. Using boolean arrays could, for example, be used to reason about sorted properties arrays.
2. **Vampire with SMT solvers**: The AVATAR architecture of Vampire [12] enables the communication between first-order provers and SAT solvers. The AVATAR architecture can also be used with SMT solvers instead of SAT solvers. Despite the fact that Vampire has built-in approximation of arithmetic axiomatization, complex properties involving arithmetic require dedicated solvers, such as theory-specific SMT solvers. We believe that program properties with arithmetic operations, such as even/odd predicates, could be generated/proven using AVATAR with SMT solving.
3. **More language constructs**: Apart from the polymorphic array theory, the FOOL logic also supports `if then else` and `let-in` constructs, allowing user to encode programs directly into FOOL logic. Changing our input language and program analysis framework to support and generate FOOL formulas is therefore an interesting venue to investigate.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY Tool. *Software and System Modeling*, 4(1):32–54, 2005.

[2] Wolfgang Ahrendt, Laura Kovács, and Simon Robillard. Reasoning about loops using vampire in key. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 9450 of *LNCS*, pages 434–443, 2015.

[3] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principle of Programming Languages*, pages 238–252, 1977.

[5] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principle of Programming Languages*, pages 84–97, 1978.

[6] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[7] C. A. R. Hoare. An Axiomatic Basis for Computer Programming (Reprint). *Commun. ACM*, 26(1):53–56, 1983.

[8] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The vampire and the FOOL. *CoRR*, pages 37–48, 2016.

[9] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 470–485, 2009.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 1–35, 2013.

[11] J. Sifakis. A Unified Approach for Studying the Properties of Transition Systems. *Theor. Comput. Sci.*, 18:227–258, 1982.

[12] Andrei Voronkov. AVATAR: The Architecture for First-Order Theorem Provers. In *International Conference on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014.