Theory-Specific Reasoning About Loops With Arrays Using Vampire

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

Chalmers University of Technology, Gothenburg, Sweden
yutingc@chalmers.se
laura.kovacs@chalmers.se
simon.robillard@chalmers.se

Abstract

The search for automated loop invariants generation has been popularly pursued due to the fact that invariants play a critical role in the verification process. Invariants with quantifiers are particularly interesting for these quantified invariants can be used to express relationships among the elements of array variables and other scalar variables. Automated invariant generation using a first-order theorem prover was first introduced by the work of Kovács and Voronkov [6] in 2009. This novel idea was further developed into a robust implementation introduced by the work of Ahrendt et al. [2]. Our work originated from the idea of enhancing the existing implementation by adding in domain-specific theory. Particularly, we extended the work of Ahrendt et al. [2] with first-order array theory reasoning.
In this paper, we describe new extensions of Vampire for supporting reasoning and proving properties of loops with arrays. The common theme of our work is the symbol elimination method for generating loop invariants. We first discuss a small change in the program analysis framework of Vampire which allows us to generate program properties with alternation of quantifiers, by simplifying skolemization during consequence finding. We then use the theory of polymorphic arrays to generate and prove program properties over arrays. We illustrate our approach on a number of examples coming from program verification.

1 Motivation

The correctness of programs could be critical in the software-controlled modern days. To be able to ensure the correctness and formally verify the desired properties has long been the goal of the research communities of computer science. Yet the formal verification and its derivation can be overwhelming and far too verbose for manual efforts. Ideally, one would like to employ a program for automated verification process. As our motivating example, here is a small imperative program with loop over arrays, written in 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 as the `partition` example, separates the non-negative values from the negative ones by element-wise copying into two result arrays.

The program copies the non-negative integer values of array `A` into array `B`, and the negative ones into array `C`. While the semantics of the program is rather trivial, the control over array indices introduces the extra complexity. Furthermore the unbounded nature of loops and arrays, the loop iteration can be any arbitrarily big number. Although the termination problem is not our focus here (the loop iteration can be infinite, in that case it does not terminate if given infinite resources), one still wishes to establish some properties of the loop for the derivation of correctness. Here are some correctness properties one wish to establish for the program above:

1. Each element of the array `B`, starting from `B[0]` to `B[b-1]`, is a non-negative integer. The value equals 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. The value equals to one element in the array `A`.
3. Each non-negative element of the array `A` equals to one of element in the array `B`.
4. Each negative element of the array `A` equals to one of element in the array `C`.
5. Any element with index beyond the final value of `b` and `c` is not updated at the end of loop.

These properties capture the algorithmic nature of the loop while expressing our semantics of the program. Also, these properties are obeyed regardless the total iteration of loop, therefore they all evaluate to `TRUE` without the knowledge of iteration count. One also refers these properties as the **loop invariants**, given these properties remain unchanged (always evaluate to `TRUE`) throughout the loop. However, to draw the connection between the program with these invariants often requires manual effort, demanding some insights of invariants extracting process itself and the semantics embedded in these invariants. Given another program with the same computational intention as the program above but with a different implementation, the invariants may no longer be true. Therefore research efforts had been made aiming to provide an automated process of reasoning these invariants without any pre-defined guidance of the user.

One of the automated invariants generation approaches has been developed by the research team within our department. Using a first-order theorem prover, this approach is capable to

derive complex loop invariants with alternating quantifiers. The line of development of our existing approach dated back to 2009, when the work of Kovács and Voronkov [6] first applied the symbol elimination method on automated invariant generation. This method relies on an efficient logical inferencing engine. In our case, it is one of the champions [8] in first-order theorem proving, Vampire [7]. In continuation of the initial work, Ahrendt et al. [2] further extended the idea to a more robust system. Not only can the new implementation generate invariants for a given program, it is now also capable to prove the correctness given the pre- and post-condition of the loop. Additionally, the new approach is interfaced toward real world programming languages via a intermediate language called *simple guarded command language*. Their work also included the translation tool from Java to the simple guarded command language, together with the connection with the JML verification tool, KeY [1].

The idea of automatic loop invariant generation using symbol elimination has been initially demonstrated in the work of Kovács and Voronkov [6]. This method was further improved by Ahrendt et al. [2]. The approach utilizes first-order theorem proving for loop invariant generation. With over half of the test cases listed in [2] solved (11 out of total 20), this approach for automated invariant generation shows promising potential. By building on the existing implementation of [2], **our work aims at enhancing the automated reasoning power of loops over arrays, hence improving the invariant generation**. From the previously failed test cases, we observed the theorem prover failing to provide the necessary logical consequences for invariant generations. Instead of treating the array variables as uninterpreted functions which take the indices as function arguments, we propose a novel approach, which directly encode array operations as their original semantics. The domain-specific knowledge of array operations are supplied via the axioms of first-order array theory. We wish to find out if one can further extend the reasoning power of this existing system, particularly using Vampire, with theory-specific reasoning.

2 Preliminary

2.1 Syntax: Simple Guarded Command Language

The input of our system is in a simple guarded command language. Differing from the famous guarded command language discussed in the work of Dijkstra [3], our simple 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 used. Given an array variable `A`, we denote `A[p]` as accessing the array element of `A` at position `p`, corresponding to array read-/select. The loop consists of a loop condition and an ordered collection of guarded statements as 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 statement is checked in sequential order and each guard is assumed to be mutually exclusive to 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 array write/store), the same position cannot be assigned to two values, i.e., if two array assignments `A[i] := e` and `A[j] := f` occur in a guarded statement, the condition `i` $\neq$ `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 formulae 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 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 which $0 \leq i < n$, 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`.

The syntax of our simple guarded command language can be best explained by Figure 4.1. 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 indexing variables `a`, `b` and `c` are limited to be initialized with their values equal (value equality is denoted by `==`) to zero. The last scalar variable, `alength`, is limited to positive values only.

The pre-conditions are followed by the post-conditions, also user specified. The keyword for denoting post-conditions is `ensures`. 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 utilize these contracts to provide better user experience, such as invariant filtering and direct 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: partition, expressed using our simple guarded command language. The second guard *true* functions as the final conditional guard, similiar to the *otherwise* construct found in other languages.

2.2 Invariant Generation Process

With the basic input syntax defined, we now introduce the entire processing flow of our approach. The entire invariant generation process can be split up into two parts: *static program analysis* and *logical inferences using the theorem prover*. This process flow is inherited from the previous implementation however the seperation provides nice modularity. Both the static analysis and theorem prover can be replaced by other analysis process or other theorem provers. The input program, written in simple guarded language, is first passed to the static program analysis process. The analysis aims to statically find out critical correlations between variables and express these correlations in first-order logic formulae. This analysis does not actually execute any part of the input program, hence it is categorized as *static* analysis. The logic formulae are bundled into one problem instance before being shipped to Vampire. The second part of logical inferences generation is then carried out by Vampire without any user interaction required. Our work enhanced the existing solution in both parts. We first introduced the theory-specific reasoning in Vampire by supplying the theory of arrays. This step involves the inclusion of axioms of array theory and associated implementation update, which is explained in section 4 then further enhanced the static analysis by providing an additional static property, the monotonic indexing. This enhancment is made to help the theorem prover by dropping some additional checking conditions. Detailed explaination can be found in section 3.

3 Program Analysis

3.1 Preprocessing of Invariant Generation

Prior to the logical inferences with saturation theorem prover, one requires to analyze the given program instance and create the logical equivalence of the program instance's semantics using the formal logic of the saturation theorem prover. In our particular case, Vampire performs logic inferences on first-order logic formulae, hence we must be able to express the semantics using first-order logic formulae. From the formulae formulated, Vampire can then infer more logic consequences on its own, hoping to generate the loop invariants or the consequences implying the post-conditions. This preprocessing step looks into the program instance and formulate the relations between program variables and the program states using first-order formulae. This step is often referred as program analysis in a correctness verification context (while program analysis in general can also discuss the optimization of programs, we only focus on the program correctness here). The goal of this preprocessing step is to automatically analyze and extract properties of the given program. Program analysis can be performed either dynamically or statically. Unlike dynamic analysis, our approach does not actually execute program statements, hence belongs to static program analysis. The phase of program analysis includes static program analysis for the properties extraction. The aim is to construct the connection between syntax (variable symbols) and the semantics (the semantics of array index operations are captured by the static properties). Specifically, our preprocessing can be further separated into the following tasks.

- Lex and parse the input program. In our implementation, we used standard C++ scanner generator flex and parser generator Bison for the simple guarded command language. A detailed definition of the syntax can be found in a later section.
- Formulate the pre- and post-conditions into first-order formulae if they are specified by the user. These formulae are created as first-order formulas (using Vampire terms). Associate the variable symbols with corresponding sorts while encode the program variables

into *extended expressions*. Extended expression language is an extension on the assertion language, which helps us capturing the program state in which the update of a particular variable occurs. More detailed introduction can be found in the later section of this chapter.

- Perform static analysis over the input program instance and extract static properties of the variables. In our approach, we do not execute any part of the actual code from the input program instance. These properties are encoded using the extended expressions.
- Bundle the extracted properties and send to theorem prover for invariant inferencing.

  From the work of [2], we learned that the formulated pre- and post-conditions can be further exploited during later stage of invariant generation. Also, the program analysis step described here can be extended with additional static property capture. One of our contribution in this master thesis work is the extension of a new static property capture, the *monotonic indexing* property which will be introduced in later section. The entire program analysis and the idea of capturing static program properties can be implemented as a stand-alone process; but in our approach, the program analysis shares the same code base with Vampire itself, making the development more convenient. Finally, the type of static properties captured can be adjusted as the different interests of program instances one would like to analyze; in our case, these static properties are targeting the operations and behaviors or array indices, allowing the logical inferences to reason about program variables and the array indices.

3.2 Extended Expressions

While the goal of invariant finding is to generate properties that hold at arbitrary iteration of the loop (think of the definition of invariants), the process of reasoning and analyzing the program instance could be improved with the help of indicators marking the program states. These program states are the critical states in which variable values get updated. With this intuition, we found the need to formulate a stronger language for describing the intermediate program states. This language must be also capable to facilitate the expressions relating program variable values in respect of the program state. With the extended language, which we called the extended expressions, one can express richer properties of the loops. Still, it is important to remind ourselves that the invariants we are after are meant to be the properties which hold at arbitrary iteration of the loop. This explains the need of the symbol elimination technique we used in our approach, which will be formally introduced in the next chapter. In this following section, we first introduce the basic assertion language, which is followed by the introduction of the extended expressions.

3.2.1 Assertion Language

The assertion language is meant to express the properties as classical first-order logic formulae, hence one should be able to correlate the two without additional knowledge. For each scalar variable v of type τ used in the program instance, we create two corresponding symbols in our assertion language: $v : \tau$ and $v_{init} : \tau$. The introduction of $v_{init} : \tau$ allows us to state the initial value of a scalar variable, while the $v : \tau$ allows us to state the general value of v. Like the interpretations in first-order logic, the interpretations in our assertion language also gives the value of each symbol. In the assertion language, the interpretations depends on a given program state α. Assuming the initial program state is represented by α_0 and the final program state is

represented by α_n, we can already know the following:

$$\begin{aligned} \forall i, \quad 0 \leq i \leq n \Rightarrow \\ v_{init} : \tau \quad of \quad \alpha_i == v : \tau \quad of \quad \alpha_0 \end{aligned} \tag{1}$$

An example of assertion language encoding can be drawn from the running example in Figure 4.1, since the value of a, b, c are assumed by the `requires` statements, one can translate the assumptions into their initial values:

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

In the previous work [2], the array variables are captured in our assertion language using the function symbols of type $\mathbb{Z} \to \tau$, where $\mathbb{Z}$ stands for integers which is used to capture the indexed access. This is no longer the case in our new implementation since we do not treat the arrays as functions taking indices anymore. The array variables are now captured using the same way as in scalar variables described above. This change also improves the readability in the codebase since now all the variables are equally handled. Finally, since the assertion language follows classic first-order logic, the pre- and post-conditions, which are in the first-order logic, are trivially describable using the assertion language.

3.2.2 Extended Expression Language

In order to allow the theorem prover leverage on the findings of each program states and draw invariant conclusion from these findings, we must enrich the underlying language for this expressivity. In other words, the program state α described in the assertion language must be incorporated into the extended expressions directly, making the reasoning possible without additional interpretation needed. The extended expressions encode the program state directly into the symbols. For each variable v of type τ, the extended expressions includes a function symbol of type $\mathbb{Z} \to \tau$, where the first argument of the function denotes the program state. Although the function type takes an integer $\mathbb{Z}$, the actual implementation has a non-negative program state limitation. The program state represents the number of iteration in which an update takes place. v^i denotes the application of the introduced functions, and is interpreted as the value of variable v in the program state α_i. In the previous implementation, array variables are expressed with an additional arity denoting the index accessed. This is no longer in our new implementation as the array is only accessed via `select(array, index)`. Finally, the extended expressions includes an additional symbol n, denoting the upper bound of loop iteration count. Notice the introduction of iteration upper bound n does not correlate to a defined number of iteration limit, the symbol n is introduced merely for the symbolical reasoning. One can substitute the symbol with another such as $final$. Examples of extended expressions are listed as follows:

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

One can reason about the value of a variable between two program states. Two program states (first iteration and last iteration) are given special symbols ($init$ and n). The following example shows the final value of v remains the same as the initial value of itself.

$$v^n = v^{init}$$

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

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

Properties expressed using the extended expressions are called extended loop properties. These extended loop properties are the central idea of the process symbol elimination, which will be introduced in the next chapter. The idea is to first extract the static properties using this extended expressions and then ask the theorem prover for logical consequences which can be expressed in assertion language. Although the extended expressions can represent stronger properties, we only regard those invariants in the assertion language valid. This is due to the fact that the pre- and post-conditions are in the assertion language and the invariant in extended expressions can describe properties of intermediate steps. With the extended expressions introduced, we now look into the static property extraction and how the properties can be written in extended expressions.

3.3 Extracting Loop Properties

With the extended expressions, one can now express the variable value at a particular program state. The next step is to statically examine the input program and try to extract useful consequences from the program itself. We denote these consequences from static analysis the loop properties. Loop properties came from mathematical observations of the updates made to each variable. Properties of each variable are extracted and formulated in first-order formulae (in our implementation, these properties are formulated as additional axioms for the theorem prover) before sending to the first-order theorem prover. The properties extracted can be significantly helpful for deciding the final invariant generation as certain static properties (and the form of these properties) ease the work of the theorem prover. In this section, we overview the existing static properties extracted and introduce the new static property added in this master project, the monotonic indexing property. Asides from the static properties collected, if the user provides the pre-conditions, the pre-conditions are directly translated into extended expressions. Since any expression in the assertion language is also in the extended expressions with minor translation needed. Here is a concrete translation example:

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

is translated into the following property in extended expressions and added into the Vampire problem instance:

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

3.3.1 Static Properties of Scalar Variable

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

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

For a scalar variable which is either increasing or decreasing, we also call it monotonic. This analysis for monotonicity can be achieved via light-weight analysis, such as verifying that every assignment to `v` is of the form `v = v + e`, where `e` is a non-negative integer.

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

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

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

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

These static analyses are geared toward reasoning about the relation between indices and the array. However, some simple arithmetic properties can be derived by these analyses. The discovered extended properties and the added static properties are summarized in the following table:

Table 1: Added static properties for scalar variables

if scalar variable v is ...	we add the 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 is never updated by the loop, one can simply treat it as a constant symbol throughout all program states.

3.3.2 Update Properties of Array Variable

For the array variables, our approach analyzes the conditions which trigger the update of the array at position `p` by the value `v` during iteration `i`. These predicates are denoted as $upd_A(i, p, v)$. Another more general predicate is denoted as $upd_A(i, p)$, which states the update during iteration `i` at index `p` with any value. These predicate conditions are then collected for the following property of 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 last iteration.

$$\begin{aligned}&(\forall i \in iteration, j \in iteration, p \in index, v)\\ &\quad (upd_A(i, p, v) \wedge (upd_A(j, p) \Rightarrow j \leq i)) \Rightarrow\\ &\quad select(A(n), p(n)) = v)\end{aligned} \tag{2}$$

3.3.3 Array Non-Update

Up until this point, all the previous static properties can be reused (with some translations) for the new implementation. However, one of the previous property must be removed from the static analysis. Given an array variable `A` which has not be updated at any location during iteration `i`, the existing approach adds the following property:

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

A naive translation of this property into the new framework would look like the following:

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

However such property should not be added as it is already assumed in the theory of array. Without any specific update via function `store`, the elements in an array are assumed to be unchanged from one program state to the next. While most static properties of scalar variables are directly reusable from previous work, the properties of array variables should be reconsidered under the new reasoning framework.

3.3.4 Monotonic Indexing

The process of static analysis can be improved separately from the logical inferencing system. **This monotonic indexing property is one of the improvement we made in this master project**. The intuition behind this monotonic indexing property originated from the monotonicity of indexing scalars. If a scalar variable is monotonic, the value of this scalar will only develop into larger (in case of monotonic increasing) or smaller (monotonic decreasing) value in later iterations. Therefore, once the update of this scalar variable occurs, the scalar variable will never have the same value again. Combining with the strictness property, if the scalar variable is strictly monotonic, we know for sure the variable will not hold the same value in later iterations. Applying this knowledge to the array indices, we can learn more about the traverse of the array.

Suppose the array variable `A` is only accessed by indexing variable x throughout the entire program, and the variable x is a monotonic variable, the monotonic indexing property can be derived from the following derivation:

1. Assuming x is *monotonic* and *strictly increasing*. From the properties described above, we know x has following property (expressed in the extended expressions) added during the program analysis.

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

2. Suppose the array `A` is updated by the program statement `A[x] := val` at iteration α. We know for any iteration $\beta > \alpha \Rightarrow x^{\beta} > x^{\alpha}$. In other words, we never revisit array `A` at index x^{α} in later iterations. This means `val` is the final value of `A[x]`, since there will not be any update at the same location in later iterations. Hence one can safely assert the following property:

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

given the array `A` is updated at index x at iteration α and the final iteration denoted by `n`. This property can be further translated into array theory based property:

$$select(A(n), x(\alpha)) == select(A(\alpha), x(\alpha))$$

The same derivation also hold for monotonic strictly decreasing, this property only relies on the fact that the loop never revisit any index in later iterations. Hence this monotonic indexing property is added for all array updates with uniquely-accessing monotonic strict indices.

An example based on our running example `partition` can show this monotonic indexing property in action:

```
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
```

In both branches of the guards, the scalar variable `a` is increased by one. Hence one can learn that variable `a` is monotonic and strictly increasing. Further, we know that the array `A` is only accessed by `a`. All of the premises of our monotonic indexing property is satisfied in this analysis, hence the property is added to the array variable `A`.

$$select(A(n), a(\alpha)) == select(A(\alpha), a(\alpha))$$

4 Theory-Specific Reasoning

In the second invariant generation phase, we perform logical inferencing on the problem instance produced by the static analysis step. With the theorem prover Vampire, one can perform logical inferencing in following modes:

1. Allow Vampire to launch the consequence finding process on the problem instance, the results are sound logical consequences, hence the valid invariants for the loop. This process is referred as the *symbol elimination*, as the auxiliary symbols such as symbols containing extended expressions are eliminated by Vampire.
2. The invariants generated from the first operation mode can be large in quantity and may not be of the interest to the user who is proving certain post-conditions of the loop. Hence the user can, by invoking our system with a special operation option, apply a filter based on the post-conditions to restrain the invariants inferred by Vampire.
3. Finally, based on the idea of refutation, the user can also directly supply the post-condition to Vampire. If the post-condition is a logical consequence of the invariants generated, including the intermediate ones expressed using extended expressions, Vampire can prove the post-condition by refutation.

In the work of Ahrendt et al.[2], the last two operation modes are introduced to extend the original work of [6]. Together with the user provided post-conditions, one can utilize the result of the symbol elimination method or even constructing a novel automation for the proof of correctness differing from the classic Hoare-rule[4] based approach.

In 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 the theorem proving. This allows our invariant

generation approach to be fully automated, the user is not required to provide any input or steer the generation during runtime. Also, Vampire is a logically sound inference system. This means all logical consequences generated would be valid consequence of the result of program analysis. Similar to other first-order theorem provers, the final performance of Vampire is significantly affected by the input. Additionally, Vampire has a sophisticated internal architecture for its unparalleled proving speed. This internal architecture is however subjected to the choice of options provide to Vampire. As we describe the process of logical inferencing in Vampire, we will also briefly explain critical concepts related to the choice of options.

Theory of Polymorphic Arrays At the time of the initiation of our project, one of the developing branches of Vampire has been published in the work of Kotelnikov et al. [5]. This particular branch supports the need of our previous implementation. The new introduction of first-class boolean variables and polymorphic arrays made the foundation needed for our project. This FOOL branch aims at mitigating the complexity of translation into first-order logic, on top of translation aid, the FOOl branch also provided different new expressions such as `if-then-else`, `let-in` and the direct support of *polymorphic arrays* with the associated theory axioms. The needed optimizations during translation are built-in as part of the FOOL extension, including the optimization over the array axioms inclusion process. Our work benefits largely by these new features and optimization.

4.1 First-Order Reasoning about Array Properties

Our main contribution to this new extension for the logical inference step is the translation from previous encoding into array theory encoding. Previously, the array variables are encoded as uninterpreted functions taking the indices as their argument. In the case of extended expression, the additional argument representing the program state can be included. In the previous encoding, one cannot explicitly express the differences between array reads and array writes. In our new reasoning framework, arrays are encoded as constant symbols instead of function taking indices. We also differentiate the operations of reading and writing in the program analysis phase. This results in a clearer semantics of array operations and disambiguates the equalities from array assignments. The concept can be more precisely explained using the following example:

- Previously, 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.

- Now with the array encoding, we represent the same array assignment statement as:

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

The array theory axioms (*read-over-write* and *extensionality*) are automatically and optimally added by Vampire once the `select` and `store` functions are invoked. The polymorphic array

introduced in the FOOL extension is more general than our guarded command language in terms of array sorts. In our language, arrays are only indexed by non-negative integers and contains either integers or booleans. But the more general underlining framework means possibility for future extension of our guarded command language.

Another alternative approach is to insert the needed array axioms into Vampire directly, as this alternative was used in the original work in [6]. In the previous versions of Vampire, the theories of integer arrays and arrays of integer arrays are coded internally using this approach. Apart from providing less support for arrays of different sorts, this approach also lose the chance of potential optimization. During the automated inclusion of axioms, Vampire optimally avoids the axioms which are not necessary. For instance, the read-over-write axioms are not added in cases where the `store` function never occurs.

5 Experiment Results

In this section we present the experiment results from our implementation for all 20 test cases, the exact same set of test cases as experimented in [2]. The numerical experiment results are followed by detail examination of critical proof steps of the newly proved examples. Finally, this section ends with the generated invariants of selected test cases, showing the capacity and improvement of our proposed approach.

In order to experiment and measure the time consumption and general ability to reason about specific loop invariants, we ran the test cases with the target invariants specified as the post-conditions. Once Vampire can refute the post-conditions, it will stop the clock and produce the entire proof steps of refutation to the user. This step can be seen as the derivation of provability of the post-conditions, also, it can be seen as the derivation of our desired loop invariants. In cases where Vampire fails to refute the post-conditions within the time limit, the program will return "refutation not found" together with the internal statistics.

All the results are collected using a computer with quad-core i7 CPU equipped with 16GB of RAM. For better comparison, the experiment from the previous paper are reproduced using the new hardware setup.

Results of all test cases Table 2 shows the experiment result *without* array theory reasoning, while Table 3 shows the final result after our array theory extension. All test cases are performed with the time limit set to 300 seconds. This time limit ensures sufficient time for the harder proofs. Other options such as splitting strategies are left as their default settings. The readings Δ_{direct} stands for the required time for Vampire to directly proof the desired post-condition from the extended properties, with unit in second. In the case where Δ_{direct} reading is missing, Vampire fails to prove the post-condition within the 300 seconds time limit and ends with refutation not found. As none of the examples reach to saturation within our time limits, we cannot formally conclude the answer to satisfiability. However, reaching to the saturation for any of the test cases is virtually impossible in our particular application. Apart from required proving time, the total number of created clauses is also featured in the table. In the cases where Δ_{direct} reading is missing, the created clauses count total created clauses within 300 seconds.

Table 3 shows the four newly proved test cases: `copyPositive`, `partition`, `partitionInit`, and `swap`. These cases witness the enhancement in reasoning about arrays in loops of the newly proposed approach. While the newly proved cases show the improvement, the experiment also showed that all the previously provable cases are still provable in array theory reasoning.

Table 2: Test cases reasoning **without** array theory

Test case	$\Delta_{direct}(sec)$	created 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: Test cases reasoning with array theory

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

With these two observations combined, we demonstrated the overall improvement over existing approach. By reasoning in the domain specific theory, in our case the theory of array, the theorem prover is capable of deriving more complex invariants automatically.

Besides showing the improvement in reasoning ability, the two tables also provide a closer comparison between two approaches. One can observe the general trend of less time used for the invariant generation *without* array theory. In other words, the implementation of treating array variables as uninterpreted functions has marginal advantage in computation complexity. This observation can be witnessed by the fact that in all commonly provable cases the time consumed in the original implementation are marginally shorter than the array theory reasoning implementation. Also, the total number of created clauses agrees on such trend, typically the number of clauses generated indicates the level of complexity needed for Vampire to reach to refutation. However, in all the commonly provable cases, both implementations manage to

derive the post-condition as the logical consequence within a fraction of a second, suggesting the new approach still remains a competitive solution for automatic invariant generation.

Although the newly proved cases cannot be directly compared with the results of previous implementation in terms of computation complexity, we found a reference time from the work of Kovács et al. [6]. In the result section of [6], the test case similar to our `partition` took around 56 seconds to generate the same invariant which our proposed approach takes about 97 seconds.

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 one of the newly proved test cases, `swap`. The program takes two integer arrays of equal length and element-wise swapping the two arrays. In Figure 3, the pre-conditions ensure equal length of the input arrays and keep a copy of the values inside each array (`olda` and `oldb`). The post-conditions for correctness of the loop check if the element in the modified array `b` is indeed 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`. This particular test case was not among the provable cases in the previous approach, however, our new approach managed to prove the post-conditions (hence generating the invariants) with only the inclusion of array theory reasoning.

```
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: `partition`

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. The previous implementation without array theory cannot prove this example, this suggests the improvement in reasoning ability of the new extension. Also, the steps show that complex invariant over array `select` and `store` with quantifier alternation can be derived using our extension. The final step `63634.` is a unification of AVATAR splitting.

With our approach, the system can automatically generate the following loop invariants:

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

which can be further de-Skolemized into more human-reader friendly form:

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

by applying the rule of inference on the righthand side of $\vee$, one can further derive:

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

6 Conclusion

The inclusion of array theory in loop invariant generation has shown to be improving the invariant reasoning ability. In our experiment results, the test case `swap` works as the witness of such improvement comparing to the previous implementation. As the nature of `swap` is heavily dependent on the array operations `select` and `store`, the array theory pays off for the derivation of correctness. Apart from the improvement brought by the array theory, we also demonstrated that the process of invariant reasoning can be enhanced by the static program analysis. The monotonic indexing property we introduced helps the theorem prover by dropping

the additional premises dealing with index bounds. This makes the theorem prover's task much simpler hence the successful proof of test cases such as `partition` and `partitionInit`. This experiment result shows the light-weight static analysis can be simple yet critical in the later logical inference process. Crafting the suitable static properties can be non-trivial. In our case the monotonic indexing property was devised after observing the additional premises during inferencing steps, which lead to difficulties for the theorem prover to derive the post-condition. These static properties are largely originated from a mathematical context. A single proper property can be crucial for the loop verification. It is very interesting to further explore more static analyses and experiment with their impact on other problem instances.

The improvement in reasoning power is certainly promising, however, we also observed marginal increase in both reasoning time and the internal clauses generated for the set of already proved cases. Comparing to the implementation back in 2009, the test case `partition` takes around twice as much computation time to derive, however, the new approach now directly handles the array operations. Despite the fact that most of the proving time required are longer than the standard timeout for Vampire in the newly proved cases, the result still indicates improvement in invariant reasoning, especially with the array-specific invariants.

On top of the enhanced reasoning power, we also found the encoding based on the array theory could benefit the human readability of the proof steps. Instead of equality over two general array terms, the new proof step clearly indicates the difference between reading from an array and writing into an array. This improvement gives a much better trace for user to examine the derivation.

In our work, we demonstrated the idea of using a theorem prover for automated invariant generation is promising. The capacity to derive complex invariants including both quantifier alternations and domain specific theory reasoning can be used to ease the manual efforts needed for loop annotation. Particularly, our approach of invariant generation requires no user guidance nor any predefined templates of invariants‘.

Future Work

1. **Boolean array test cases and experiments**: with the foundation of FOOL and the polymorphic arrays, boolean arrays can be incorporated into our array theory reasoning approach. Due to project time limit, we have not yet included test cases containing boolean arrays. In the future work, boolean array can be used to explore interesting properties such as partial sortedness of the array. Furthermore, since Vampire with the FOOL extension already support first class boolean reasoning, it is interesting to experiment with the boolean array invariants.
2. **Vampire with SMT solvers**: The AVATAR [9] structure of Vampire makes it possible to delegate satisfiability problems to a SAT solver. This structure enables the superposition-based Vampire to reason about logical consequences much faster hence solving the problem within the time constraint. Furthermore, the AVATAR structure can accommodate other types of solvers such as SMT solvers. Despite the fact that Vampire has built-in approximation of arithmetic axiomatization, complex properties involving arithmetic require dedicated solvers. Problems involving arithmetic properties such as even / odd indices can possibly benefit from the arithmetic reasoning ability of SMT solvers such as Z3.
3. **More language constructs**: Apart from the polymorphic arrays, the paper of Vampire and the FOOL [5] also introduced the entire first-class boolean and corresponding constructs into the latest Vampire branch. Among the constructs introduced, the `if then else` construct can be directly benefiting program analysis. Our input language

syntax can be extended with the condition construct and internally mapped to the new feature from the FOOL infrastructure. This would also mean new static properties can be explored specifically for the condition construct.

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 *Lecture Notes in Computer Science. 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2015, Suva, Fiji, 24-28 November 2015*, pages 434–443. Springer Berlin Heidelberg, 2015.

[3] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[4] C. A. R. Hoare. An axiomatic basis for computer programming (reprint). *Commun. ACM*, 26(1):53–56, 1983.

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

[6] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 470–485, 2009.

[7] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[8] Geoff Sutcliffe and Christian B. Suttner. The CADE ATP system competition. In *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, pages 490–491. Springer, 2004.

[9] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 696–710. Springer, 2014.