Reasoning About Loops Using Vampire*

Laura Kovács and Simon Robillard

Chalmers University of Technology, Gothenburg, Sweden

Abstract

In 2009, the symbol elimination method for loop invariant generation was introduced [8], which used saturation theorem proving in first-order logic to generate quantified invariants of programs with arrays. Symbol elimination is fully automatic, requires no user guidance, and it is the first ever approach able to generate invariants with alternations of quantifiers. In this paper we describe a number of improvements and extensions to symbol elimination and invariant generation using first-order theorem proving, in particular the Vampire theorem prover. Rather than being limited to a specific programming language, our approach to reasoning about loops in Vampire relies on a simple guarded command language for its input, which can be used as an interface for more complex and realistic imperative languages. We propose new ways for extending quantified loop properties describing valid loop properties, by simplifying the properties over array updates and next state relations. We also extend symbol elimination with pre- and post-conditions of loops. We use the loop specification to generate only invariants that are relevant, that is, invariants that are needed for proving partial correctness of loops. Further, we turn symbol elimination into an automatic approach proving program correctness, providing an alternative method to Hoare-rule based loop verification or other deductive systems. We present our newly redesigned implementation of loop reasoning in Vampire and report on experimental results.

1 Introduction

In [8], the symbol elimination method for generating invariants was introduced. The approach uses saturation theorem proving to generate quantified invariants, possibly with quantifier alternations, for programs with unbounded data structures such as arrays. Implementations of symbol elimination using the first-order theorem prover Vampire [10] have been previously described in [6, 3]. However these implementations differ from the original description of symbol elimination in a number of ways. Most notably, while the original description of the method assumes that loops are given in a simple guarded command language, both implementations instead take as input loops written in a subset of C, and hence can be used only for analyzing loops written in C.

In this paper we describe a number of improvements and extensions to symbol elimination and invariant generation using first-order theorem proving, in particular Vampire. Rather than being limited to a specific programming language, our approach to reasoning about loops in Vampire relies on a simple guarded command language for its input, which can be used as an interface for more complex and realistic imperative languages. Compared to previous results [6, 3], our work makes symbol elimination more efficient, and more importantly, provides a clean interface for verifying various realistic imperative languages. Details on the guarded language representation used by our work are given in Section 2, whereas symbol elimination in Vampire is described in Section 3.

*This work was partially supported by the Wallenberg Academy Fellowship 2014, the Swedish VR grant D0497701 and the Austrian research project FWF S11409-N23.

Our work is compatible with recent developments in Vampire. In order to take advantage of these changes, the program analysis phase of symbol elimination – during which some lightweight static analysis techniques are used as a first step to symbol elimination – has been modified and improved. We propose new ways for extending quantified loop properties describing valid loop properties, by simplifying the properties over array updates and next state relations. These improvements result in properties that are more easily handled by the inference engine of Vampire; they are detailed in Section 4. We also extended symbol elimination by taking into consideration also the loop specification (contract), which may optionally be given by the user in the form of pre- and post- conditions. If available, pre-conditions are used to derive more precise invariants, and post-conditions can be used to select the subset of invariants relevant to the verification task. We also turn symbol elimination into an automatic (incomplete) way to directly prove the correctness of the loop w.r.t. to a contract. Our work provides an alternative to Hoare-style verification of loops and avoids the need for explicitly stated invariants. Generating relevant invariants and proving partial correctness of loops using symbol elimination are presented in Section 5. We experimentally evaluate our work and report on our result in Section 6.

In order to achieve the above improvements and extensions to symbol elimination, we completely re-implemented symbol elimination in Vampire. Our work provides a new and fully automated tool for invariant generation and proving partial correctness of loops. Our implementation required 3000 lines of C++ code, is fully compatible with the recent version of Vampire (version 3.0), and is available at `www.cse.chalmers.se/~simrob`.

2 Input Language

2.1 Syntax

Inputs to our approach are loops with nested conditionals, written in a simple guarded command language. Loops may contain scalar variables and arrays ranging over (unbounded) integers. In what follows, we use upper case letters $A, B, C, \ldots$ to denote array variables and lower case letters $a, b, c, \ldots$ for scalars. We use standard arithmetical function symbols $+, -, \cdot, \div$ and predicate symbols $\leq, \geq$. We write $A[p]$ to mean (an access to) the array element at position p in the array A.

We describe loops by a *loop condition* and an ordered collection of *guarded statements*; the loop condition is a quantifier-free Boolean formula over program variables. A guarded statement is a pair of a *guard* (also a Boolean formula) and a collection of assignments. In our setting, a guarded statement cannot contain two assignments to the same scalar variable v. If two array assignments $A[i] := e$ and $A[j] := e'$ occur in a guarded statement, the condition $i \neq j$ is added to the guard. These two restrictions ensure that each location is modified at most once by a given guarded statement.

In addition to the loop itself, pre- and post-conditions can also be specified, using the keywords `requires` and `ensures`, respectively. Pre- and post-conditions are Boolean formulas over program variables, possibly with quantifiers.

Figure 1 gives an example of a loop using the syntax supported by our work.

2.2 Semantics

We define the semantics of the guarded command language by the notion of *program states* mapping scalar variables to values of the correct type and arrays to functions. Note that arrays

```
requires (k == 0);
ensures forall int p, (0 <= p & p < n) ==>
                        (A[p] >= B[p]
                        & A[p] >= C[p]
                        & (A[p] == B[p] | A[p] == C[p]));
while (k < n) do
  :: B[k] >= C[k] -> A[k] = B[k]; k = k + 1;
  :: true         -> A[k] = C[k]; k = k + 1;
od
```

Figure 1: Example of an input to our work. This example loop is composed of two guarded statements; it computes the maximum of elements in arrays B and C at every position and writes it in the corresponding position in the array A. The program specification is given by the pre- (`requires`) and post-conditions (`ensures`).

bounds are not dealt with in the semantics: in a given state, an array storing values of type τ is treated as a total function of type $\mathbb{Z} \to \tau$. Array bounds checking may easily be encoded with the help of guards if required. Evaluation of program expressions in a given state is done in the standard way.

In our setting, there is exactly one program state for each loop iteration. The symbol n is used to denote the upper bound on the number of loop iterations, so that for any loop iteration i we have $0 \leq i < n$. We write σ_0 and σ_n to respectively speak about the initial and final state of the loop. If the loop condition is valid in a given program state σ_i, the first guarded statement whose guard is valid is executed: its assignments are applied simultaneously to σ_i, yielding the state σ_{i+1}. For example, executing the guarded statement

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

in a state where $x = 1$ holds, yields a state in which $y = 1$ and not $y = 0$.

If the loop condition is not valid, or if none of the guards hold, the loop is terminated: σ_i becomes the final state of the loop σ_n.

Note that while these semantics are deterministic, our method for invariant generation could be adapted to work with non-deterministic semantics with only minor changes.

2.3 Simulating Complex Languages

While previous implementations of symbol elimination [6, 3] used a syntax similar to the C programming language, only a subset of C programs could be analyzed. Many aspects of the semantics of C were not taken into account.

By using a guarded command language, our implementation clarifies the semantics of the input language. It is consequently easier to use the guarded command language as an representation of the semantics of a program given in another language. In our experiments, we demonstrated this possibility by using the KeY verification system [1] to translate Java programs with loops into our guarded command language.

Currently this translation is not complete: in order to fully encode the semantics of most imperative languages used in industry, one must be able to represent exception throwing and catching, abrupt termination and heap-related properties, among others. Many of those aspects can be easily and efficiently encoded by introducing additional Boolean variables in the program, however at the time of writing, Boolean variables are not supported by our tool. This

support should be added soon, thanks to the recent introduction of a first-class Boolean sort in Vampire [7].

3 Invariant Generation Using Symbol elimination

The symbol elimination method aims at producing invariants for a given loop, i.e. first-order formulas in a language of assertions $\mathcal{L}_{asrt}$ that hold at arbitrary iterations of the loop. The central idea of symbol elimination is to use formulas expressed in a language of extended expressions $\mathcal{L}_{extd}$ during intermediate steps of the procedure. This language can express richer properties of the loop than is possible with $\mathcal{L}_{asrt}$: while any formula using symbols in $\mathcal{L}_{asrt}$ has a semantic equivalent in $\mathcal{L}_{extd}$, the converse is not true. During the procedure, we first deploy static analysis techniques to extract properties of the loop expressed in $\mathcal{L}_{extd}$. In a second phase, we use saturation theorem proving to discover consequences of those properties that can be expressed using only symbols from $\mathcal{L}_{asrt}$. Such properties are loop invariants.

In this section, we define $\mathcal{L}_{asrt}$ and $\mathcal{L}_{extd}$, then describe the symbol elimination procedure to generate loop invariants. The definitions assume a given loop, in particular they depend on the set of program variables used within that loop.

3.1 Assertions

We define $\mathcal{L}_{asrt}$, the language of assertions, as follows. For each scalar variable `v` of type τ in the loop, $\mathcal{L}_{asrt}$ includes two symbols $v : \tau$ and $v_{init} : \tau$. For each array A storing values of type τ, $\mathcal{L}_{asrt}$ includes a function symbol of type $\mathbb{Z} \to \tau$. Interpretation of a formula using symbols in $\mathcal{L}_{asrt}$ depends on a given program state σ. The symbol v is interpreted as the value of the program variable `v` in that state, while v_{init} is interpreted as the value of that variable at the start of the loop.

An invariant is a formula that uses symbols from $\mathcal{L}_{asrt}$ and is valid for any state σ_i. The pre- and post-conditions of the loops are formulas in $\mathcal{L}_{asrt}$ that are required to hold at the initial state σ_0 and the final state σ_n, respectively.

3.2 Extended Expressions

Unlike $\mathcal{L}_{asrt}$, symbols in $\mathcal{L}_{extd}$ do not depend on a particular program state for interpretation. Formulas using such symbols can express properties of the loop at arbitrary states, such as the relation between two successive program states.

For every variable `v` of type τ, $\mathcal{L}_{extd}$ includes a function of type $\mathbb{Z} \to \tau$[1]. For convenience, applications of these functions are noted $v^{(i)}$; they are interpreted as the value of `v` in the state σ_i. For each array A, $\mathcal{L}_{extd}$ includes a function of type $\mathbb{Z} \times \mathbb{Z} \to \tau$. Similarly, we use the notation $A^{(i)}[p]$ to represent the value stored at position p after the ith iteration. We call $v^{(i)}$ and $A^{(i)}[p]$ *extended expressions*. Note for any program expression E, we can build a term (or predicate, in the case of Boolean program expressions) by systematically replacing each variable by its extended expression. We may simply abbreviate such construction $E^{(i)}$.

$\mathcal{L}_{extd}$ also includes the symbol n which denotes the upper bound on the number of loop iterations. Formulas in $\mathcal{L}_{extd}$ that are valid for a given loop are called *extended loop properties*.

The following semantic equivalences relate $\mathcal{L}_{asrt}$ and $\mathcal{L}_{extd}$

[1] The type $\mathbb{N} \to \tau$ would perhaps be more accurate, but in practice it is more efficient to add predicates enforcing the non-negativity where needed.

$$\begin{aligned} v^{(0)} &\equiv v_{init} \\ v^{(n)} &\equiv v \\ A^{(0)}[p] &\equiv A_{init}[p] \\ A^{(n)}[p] &\equiv A[p] \end{aligned}$$

3.3 Loop Analysis and Symbol Elimination

In the first step of our invariant generation procedure, we perform simple static analysis to generate extended loop properties. For example, analyzing the program in Figure 1 would lead to generating the following property:

$$(\forall i)(0 \leq i < n \implies k^{(i+1)} = k^{(i)} + 1)$$

This property, which describes the assignment to the variable `k` at each iteration, is added to the list of extended properties as an assumption. A comprehensive description of the analysis performed by our tool and the resulting properties is given in Section 4. Note that this phase is quite flexible, and additional properties (user knowledge, invariants generated by other tools...) could potentially be added to the list of extended properties.

While the properties extracted during that phase are valid at arbitrary loop iterations, they are not yet invariants as they use symbols extended expressions, symbols that are not in $\mathcal{L}_{asrt}$. The next step in our invariant generation process is to eliminate symbols that are not in $\mathcal{L}_{asrt}$. This is done by generating formulas that only use symbols from $\mathcal{L}_{asrt}$ and are logical consequences of the properties in $\mathcal{L}_{extd}$. To this end we use the prover to perform symbol elimination and generate invariants in $\mathcal{L}_{asrt}$. For more details on symbol elimination we refer to [9].

4 Extracting Loop Properties

In this section, we list the properties extracted from the loop during the first phase of invariant generation. It is important to note that there is no definitive way to chose which properties must be extracted from the loop, as long as those properties are indeed consequences of the loop semantics. The strength and the formulation of the properties play a great role in the quality of the invariants produced.

4.1 Properties of Scalar Variables

Program variables that are never updated by the loop body are treated as constant symbols during the analysis. For variables that are updated, simple static analysis techniques are used to characterize the behavior of those updates.

Let us call a scalar variable v *increasing* if, for all possible computations of the loop, it has the property

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \geq v^{(i)})$$

Similarly, we call v *decreasing* if

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \leq v^{(i)})$$

A variable is said to be *strict* if it is modified at every iteration, i.e.

$$(\forall i)(0 \leq i < n \implies v^{(i+1)} \neq v^{(i)})$$

Finally a variable is called *dense* if its value is increased or decreased by at most one during any iteration

$$(\forall i)(0 \leq i < n \implies |v^{(i+1)} - v^{(i)}| \leq 1)$$

Having detected those properties of the variables, the following properties are added to the list of extended properties:

1. If v is increasing, strict and dense, we add the property:

$$(\forall i)(v^{(i)} = v^{(0)} + i)$$

2. If v is increasing and strict, but not dense, we add the property:

$$(\forall i)(\forall j)(j > i \implies v^{(j)} > v^{(i)})$$

3. If v is increasing but not strict, we add the property:

$$(\forall i)(\forall j)(j \geq i \implies v^{(j)} \geq v^{(i)})$$

4. If v is increasing and dense, but not strict, we add the property:

$$(\forall i)(\forall j)(j \geq i \implies v^{(i)} + j \geq v^{(j)} + i)$$

Similar properties, with the required modifications, are generated for decreasing variables.

4.2 Update Properties of Arrays

In order to describe the behavior of arrays, for each array we analyze the guarded statements to collect:

1. the conditions under which the array is updated at position p by the value v during iteration i. Let us consider the example in Figure 1, for the array A (the only one to be updated), these conditions are

$$\begin{array}{ll} & (0 \leq i < n \ \wedge \ B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ v = B^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \\ \vee & (0 \leq i < n \ \wedge \ \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ v = C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \end{array}$$

   which we denote $upd_A(i, p, v)$

2. the conditions under which the array is updated at position p during iteration i, by any value. For the same example, they are

$$\begin{array}{ll} & (0 \leq i < n \ \wedge \ B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \\ \vee & (0 \leq i < n \ \wedge \ \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \ \wedge \ p = k^{(i)}) \end{array}$$

   these are noted $upd_A(i, p)$

After this analysis we can express the following properties of the array:

1. if the array is never updated at a position p, the value at this position remains constant

$$(\forall i\, p)\left(\neg upd_A(i, p) \implies B^{(n)}[p] = B^{(0)}[p]\right)$$

2. if the array is updated only once at a position p, the value associated with this update is the final value

$$(\forall i\, j\, p\, v)\left(upd_A(i,p,v) \wedge (upd_A(j,p) \implies j = i) \implies B^{(n)}[p] = v\right)$$

Note that compared to [8], the second property has been modified as it used to read

$$(\forall i\, j\, p\, v)\left(upd_A(i,p,v) \wedge (upd_A(j,p) \implies j \leq i) \implies B^{(n)}[p] = v\right)$$

While less general, the new property is more easily handled by the prover, since equality is a built-in predicate of the superposition calculus used by Vampire.

In previous implementations, predicate symbols corresponding to upd_A were used in both properties, and assumptions giving the predicate definitions were also added. Those predicate symbols were then eliminated. The new tool replaces every occurrence of the predicate symbol directly by its definition, thus increasing efficiency and the quality of invariants produced.

4.3 Assignments

The relation between two consecutive states, and in particular the effects of assignments on states, can be described by extended expressions.

For the program in Figure 1, the following two properties (one for each guarded statement) are extracted and added to the extended properties.

$$\begin{aligned}(\forall i)(0 \leq i < n \wedge B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \implies\quad & A^{(i+1)}[k^{(i)}] = B^{(i)}[k^{(i)}] \\ & \wedge\, k^{(i+1)} = k^{(i)} + 1\end{aligned}$$

$$\begin{aligned}(\forall i)(0 \leq i < n \wedge \neg B^{(i)}[k^{(i)}] \geq C^{(i)}[k^{(i)}] \implies\quad & A^{(i+1)}[k^{(i)}] = C^{(i)}[k^{(i)}] \\ & \wedge\, k^{(i+1)} = k^{(i)} + 1\end{aligned}$$

4.4 Additional Properties

Finally the property indicating that the loop condition and one guard must hold at any given iteration is added to the assumptions.

$$(\forall i)(0 \leq i < n \implies \bigvee_j G_j^{(i)} \wedge C^{(i)})$$

In the original description of the symbol elimination method, arithmetic function and predicate symbols were introduced as needed and given an axiomatization. This is no longer necessary, as we use the default symbols now provided by Vampire. At the moment, any arithmetic reasoning in Vampire is still based on axiomatic theories, but symbol elimination would directly benefit from any further development concerning arithmetic reasoning in Vampire.

As noted before, the list of extended properties is not definitive. This makes our method flexible, as *ad-hoc* properties can potentially be added to the assumptions, whether it be user knowledge or properties gathered by other invariant generation techniques (e.g. [4, 5])

5 Loop Contract and Correctness

Previous works on symbol elimination [6, 3] report every property discovered during symbol elimination. This often results in hundreds of clauses being reported to the user in a few seconds, many of which are consequences of each other. To address this issue, a post-processing step was added during which some redundant clauses were eliminated. However minimizing a set of first-order clauses is an undecidable problem. Even if a minimal set of clauses is obtained, previous works on symbol elimination do not take into account a verification contract (specification) for analyzing and verifying loops. Therefore there is no realistic way to assess the quality of generated invariants in the process of verification. We also note that symbol elimination generates invariants that hold at any iteration of the loop, but may not be inductive. Using non-inductive invariants makes software verification harder.

By enabling the user to specify a post-condition of the loop, and using it to select relevant invariants within the set produced by symbol elimination, we address those issues. Unlike previous works, our work enables the user to specify optional pre- and post-conditions for the loop under analysis, using the keywords `requires` and `ensures`, respectively. They are expressions in $\mathcal{L}_{asrt}$ (quantified Boolean formulas over program variables).

5.1 Pre-conditions

Recall that any expression in $\mathcal{L}_{asrt}$ can be translated to an expression $\mathcal{L}_{extd}$. Pre-conditions given by the user as expressions in $\mathcal{L}_{asrt}$ are simply translated to $\mathcal{L}_{extd}$ and added to the extended properties. For example this precondition

```
requires forall int p, 0 <= p & p < l ==> A[p] != 0
```

results in the following property being added to the extended properties:

$$(\forall i)(0 \leq p < l \implies A^{(0)}[p] \neq 0)$$

Such additional information enables symbol elimination to derive stronger invariants.

5.2 Invariant filtering

Given a loop condition C, a post-condition P and a set of invariants $I_1, \ldots, I_k$ produced by symbol elimination, we attempt to prove P under the assumptions $I_1 \wedge \cdots \wedge I_k \wedge \neg C$. If the refutation proof succeeds, we can select the subset of invariants that were effectively used: they are among the leaves of the proof tree.

This filtering process is carried out in parallel of symbol elimination. One instance S_{gen} of the saturation algorithm is ran to generate invariants, possibly with a time limit. Another instance S_{filter} is started on a different thread, it initially tries to prove P assuming only $\neg C$. Each time a new invariant is discovered by S_{gen}, it is added to the list of assumptions in S_{filter}, and the proof attempt is restarted. This way, the process can stop as soon as the set of discovered invariants is strong enough to imply the post-condition. If the time limit of S_{gen} is reached however, the whole process is aborted.

This filtering mechanism also provides a good heuristic to select an inductive invariant. While this is not always true, our experiments (Section 6) show that the set of invariant selected is usually inductive.

5.3 Direct Proof of Correctness

During invariant filtering, we use invariants, which are consequences of the extended properties, to prove the post-condition. In any case where this succeeds, the post-condition is also a consequence of the extended properties.

As an alternative to invariant filtering, our tool offers the option of omitting the symbol elimination stage and proving the post-condition from the extended properties themselves. In this setting, no invariants are used or reported. This provides an alternative to classic Hoare-style verification of loops which, while incomplete, is fully automatic.

Finding a direct proof of correctness of the loop is faster than performing invariant filtering (see Section 6) and should succeed for every program where invariant filtering succeeds. In some cases, due to the fact that extended properties are stronger than the invariants they imply, a direct proof may even succeed where invariant filtering does not.

6 Experimental Results

6.1 Benchmarks

We evaluated our tool on 20 challenging array benchmarks taken from academic papers [2, 3] and the C standard library. Our benchmarks are listed in Table 1. The program `absolute` computes the absolute value of every element in an array, whereas `copy`, `copyOdd` and `copyPositive` copy (some) elements of an array to another. The example `find` searches for the position of a certain value in an array, returning -1 if the value is absent. The program `findMax` locates the maximum in an unsorted array. The examples `init`, `initEven`, and `initPartial` initialize (some) array elements with a constant, whereas `initNonConstant` sets the value of array elements to a value depending on array positions. `inPlaceMax` replaces every negative value in an array by 0, and `max` computes the maximum of two arrays at every position. `mergeInterleave` interleaves the content of two arrays, whereas `partition` copies negative and non-negative values from a source array into two different destination arrays. `reverse` copies an array in reverse order, and `swap` exchanges the content of two arrays. Finally, `strcpy` and `strlen` are taken from the standard C library. Each benchmark contains a loop together with its specification. All experiments were performed on a computer with a 2.1 GHz quad-core processor and 8GB of RAM.

6.2 Results

Table 1 summarizes our results. The second column indicates whether the benchmark loops contain conditionals. Column Δ_{direct} shows the time required to prove the partial correctness of the benchmarks, by proving the loop specification from the extended properties generated by program analysis in Vampire. On the other hand, column Δ_{filter} gives the time needed by our tool to generate the relevant invariants from which the loop post-condition can be proved. The time results are given in seconds. Where no time is given, a correctness proof/filtering of relevant invariants was not successful. Column N_5 shows the number of all invariants generated by our tool with a time limit of 5 seconds (before filtering of relevant invariants). The figure listed in parentheses gives the number of invariants produced by a previous implementation [3] of invariant generation in Vampire. Finally, column N_{filter} reports the number of invariants selected as relevant invariants; the conjunction of these invariants is the relevant invariant from which the loop specification can be derived.

Note that for all examples, our tool successfully generated quantified loop invariants. Moreover, when compared to the previous implementation [3] of invariant generation in Vampire,

Table 1: Experimental results on loop reasoning using Vampire.

Name	Cond.	Δ_{direct}	Δ_{filter}	N_5	N_{filter}
absolute	yes	0.271	2.358	19	3
copy	no	0.043	2.194	9 (37)	1
copyOdd	no	0.122	2.090	9 (214)	1
copyPartial	no	0.042	3.145	9	1
copyPositive	yes			9	
find	yes			123	
findMax	yes			3	
init	no	0.035	2.059	9 (35)	1
initEven	no			10	
initNonConstant	no	0.114	2.054	9 (104)	1
initPartial	no	0.042	3.129	9	1
inPlaceMax	yes			39	
max	yes	0.696	3.535	20	2
mergeInterleave	no			20	
partition	yes			164 (647)	
partitionInit	yes			98 (169)	
reverse	no	0.038		9 (42)	
strcpy	no	0.036	2.126	9	1
strlen	no	0.018	2.023	2 (26)	1
swap	no			26	

our tool brings a significant performance increase: in all examples where the implementation of [3] succeeded to generate invariants, the number of invariants generated by our tool is much less than in [3]. For example, in the case of the program `copyOdd`, the number of invariants generated by our tool has decreased by a factor of 24 when compared to [3]. This increase in performance is due to our improved program analysis for generating extended loop properties. For the examples where the number of invariants generated by [3] is missing, the approach of [3] failed to generate quantified loop invariants over arrays. We also note that invariants generated by [3] are logical consequences of the invariants generated by our tool.

When evaluating our tool for proving correctness of the examples, we succeeded for 11 examples out of 20, as shown in column Δ_{direct} of Table 1. For these 11 examples, the partial correctness of the loop was proved by Vampire by using the extended loop properties generated by our tool. Further, for 10 out of these 11 examples, our tool successfully selected the relevant invariants from which the loop specification could be proved. For the example `reverse` the relevant invariants could not be selected within a 5 seconds time, even though the partial correctness of the loop was established using the extended properties of the loop.

When analyzing the 9 examples for which our tool failed to generate relevant invariants and to prove partial correctness, we noted that these examples involve non-trivial arithmetic and array reasoning. We believe that improving reasoning with full first-order theories in Vampire would allow us to select the relevant invariants from those generated by our tool.

7 Conclusion

We provide a new and fully automated tool for invariant generation, by re-implementing and improving program analysis and symbol elimination in Vampire. One of these improvements is the dedicated parser for the guarded command language, which can now be used a simple way to describe the semantics of a loop. We also introduce a number of simplifications during the generation of extended properties of loops, leading to an increased quality in the invariants produced. We allow the possibility of specifying a verification contract for the loop being analyzed, and we add a filtering stage to output only invariants that are relevant to the partial correctness of the loop w.r.t. to that contract. We also extend symbol elimination to directly prove partial correctness of loops, without the need for explicitly stating invariants. We experimentally evaluated our tool on a number of examples.

For future work, we intend to improve theory reasoning in Vampire; this should benefit program analysis as well as more traditional applications of the theorem prover. The analysis of programs that we perform generates first-order problems, which we believe are challenging benchmarks for reasoning with quantifiers and theories. They would be an interesting addition to the CASC theorem proving competition [11]. We are also interested in analyzing more complex programs and support the translation of the full semantics of a programming language such as Java into our program analysis framework.

References

[1] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[2] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *Programming Languages and Systems*, pages 246–266. Springer, 2010.

[3] Ioan Dragan and Laura Kovács. Lingva: Generating and proving program properties using symbol elimination. In *Perspectives of System Informatics*, pages 67–75. Springer, 2014.

[4] Juan Pablo Galeotti, Carlo A Furia, Eva May, Gordon Fraser, and Andreas Zeller. Dynamate: dynamically inferring loop invariants for automatic full functional verification. In *Hardware and Software: Verification and Testing*, pages 48–53. Springer International Publishing, 2014.

[5] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification*, pages 634–640. Springer, 2009.

[6] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in vampire. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 60–64. Springer, 2011.

[7] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class boolean sort in first-order theorem proving and TPTP. In *Conference on Intelligent Computer Mathematics (CICM)*, 2015. To appear.

[8] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[9] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In *Automated Deduction–CADE-22*, pages 199–213. Springer, 2009.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification*, pages 1–35. Springer, 2013.

[11] G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.