# Reasoning About Loops Using Vampire in KeY[*]

Wolfgang Ahrendt, Laura Kovács, and Simon Robillard

Chalmers University of Technology

**Abstract.** We describe symbol elimination and consequence finding in the first-order theorem prover Vampire for automatic generation of quantified invariants, possibly with quantifier alternations, of loops with arrays. Unlike the previous implementation of symbol elimination in Vampire, our work is not limited to a specific programming language but provides a generic framework by relying on a simple guarded command representation of the input loop. We also improve the loop analysis part in Vampire by generating loop properties more easily handled by the saturation engine of Vampire. Our experiments show that, with our changes, the number of generated invariants is decreased, in some cases, by a factor of 20. We also provide a framework to use our approach to invariant generation in conjunction with pre- and post-conditions of program loops. We use the program specification to find relevant invariants as well as to verify the partial correctness of the loop. As a case study, we demonstrate how symbol elimination in Vampire can be used as an interface for realistic imperative languages, by integrating our tool in the KeY verification system, thus allowing reasoning about loops in Java programs in a fully automated way, without any user guidance.

## 1 Introduction

Reasoning about the (partial) correctness of programs with loops requires loop invariants. Typically, loop invariants are provided by the user as annotations to the program, see e.g. [1, 4, 13]. Providing such annotations requires a considerable amount of work by highly qualified personnel and often makes program analysis prohibitively expensive. Therefore, automation of invariant generation is invaluable in making program analysis scale to large, realistic examples.

In [10], the symbol elimination method for generating invariants was introduced. The approach uses first-order theorem proving, in particular the Vampire prover [11]. Symbol elimination allows the generation of quantified invariants, possibly with quantifier alternations, for programs with unbounded data structures, such as arrays. While experiments of invariant generation in Vampire show that symbol elimination generates non-trivial invariants, the initial implementation [6] of program analysis and invariant generation in Vampire has various disadvantages: it can only be used with programs written in C, the number of generated invariants is too large, and generating relevant invariants did not take

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

into account the program specification. Moreover, the process of invariant generation was not integrated, nor evaluated in a verification framework, making it hard to assess the quality and practical impact of invariant generation by symbol elimination. In this paper we address these limitations, as follows.

We provide a new and fully automated tool for invariant generation, by using symbol elimination in Vampire (Section 2). To this end, we re-implemented program analysis and invariant generation in Vampire. Our implementation is fully compatible with the most recent development changes in Vampire. It is designed to be independent of any particular programming language: inputs to our tool are program loops written in a simple guarded command language. We also improved program analysis in Vampire by generating loop properties that are more easily handled by the saturation engine of Vampire. We also show that symbol elimination can be used not only to produce invariants, but also as a direct (incomplete) method to prove the correctness of the loop. Our work provides an alternative approach to Hoare-style loop verification and cancels the need for explicitly stated invariants as program annotations.

Reasoning about real programming languages poses several challenges, e.g. using machine integers instead of mathematical ones or reasoning about out-of-bound array accesses. In order to showcase the relevance of our implementation in real applications, we integrated our approach to loop reasoning in Vampire into the KeY verification system [1], thus allowing automatic reasoning about loops in Java programs (Section 3). We experimentally evaluate invariant generation in Vampire on realistic examples (Section 4).

The main advantage of our tool comes with its full automation for generating invariants, possibly with quantifier alternations. Unlike [8, 7], where user-given invariant templates are used, we require no user guidance and infer first-order invariants with arbitrary quantifiers. Contrary to [3], we do not use specialized abstract domains, but use saturation theorem proving to generate quantified invariants. Theorem proving, in the form of SMT solving, is also used in [12] to automatically compute loop invariants, however only with universal quantifiers.

Our implementation of invariant generation in Vampire[1] required 3000 lines of C++ code. The integration of Vampire with KeY required about 1000 lines of Java code.

## 2 Invariant Generation in Vampire

In this section, we describe our tool to generate quantified loop invariants in a fully automatic manner. Our work uses symbol elimination and consequence finding in Vampire and extends Vampire with a new framework for reasoning about loops. Compared to the earlier implementation [6] of invariant generation in Vampire, our tool is independent of the language in which the loops are expressed, simplifies symbol elimination in saturation theorem proving, and provides various ways to generate a relevant set of loop invariants. The overall workflow of our tool is given in Figure 1 and detailed below.

[1] available at `www.cse.chalmers.se/~simrob`

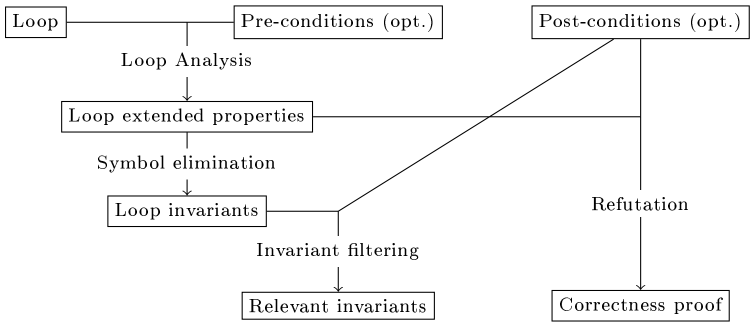

**Fig. 1.** Loop reasoning in Vampire.

**Input.** Inputs to our tool are loops with nested conditionals, written in a simple guarded command language. Optionally, pre- and post-conditions can also be specified. We use standard arithmetical function symbols $+, -, \cdot, \div$ and predicate symbols $\leq, \geq$.

Loops may contain scalar variables and arrays ranging over Boolean values and (unbounded) integers. 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 assignments*; the loop condition is a quantifier-free Boolean formula over program variables. A guarded assignment is a pair of a *guard* (also a Boolean formula) and a collection of assignments. In our setting, a guarded assignment 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 assignment, the condition $i \neq j$ is added to the guard. Figure 2 gives an example of a loop using the syntax supported by our tool.

**Loop semantics.** We assume basic knowledge about program states and transition relations. We use $n$ to denote the upper bound on the number of loop iterations and write $\sigma_0$ and $\sigma_n$ to respectively speak about the initial and final state of the loop. For any loop iteration $i$ we have $0 \leq i \leq n$. Given a program state $\sigma_i$ describing the value of each program variable after a loop iteration $i$, the semantics of the loop is as follows. If the loop condition is valid in $\sigma_i$, the first guarded assignment whose guard is valid is executed: its assignments are applied simultaneously to $\sigma_i$. This way the state $\sigma_{i+1}$ corresponding to the loop iteration $i+1$ is obtained from $\sigma_i$. For example, executing the guarded assignment `true -> x = 0; y = x;` in a state where $x = 1$ holds, yields a state in which $y = 1$. If the loop condition is not valid, or if none of the guards hold, $\sigma_i$ becomes the final state of the loop $\sigma_n$.

**Loop assertions and invariants.** For each loop variable $v$, we denote by $v_{init}$ the value of $v$ at the initial state $\sigma_0$ of the loop. By extending the language of quantifier-free Boolean expressions with universal and existential quantifiers over program variables $v$ and their initial values $v_{init}$, we obtain the language of *loop assertions* denoted by $\mathcal{L}_{asrt}$. An invariant is a formula that uses symbols

```
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
```

**Fig. 2.** Example of an input to our tool. This example loop is composed of two guarded assignments, 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`).

from $\mathcal{L}_{asrt}$ and is valid for any state $\sigma_i$. The pre- and post-conditions of the loops are formulas in $\mathcal{L}_{asrt}$ that are required to hold at $\sigma_0$ and $\sigma_n$, respectively.
**Extended loop properties.** For every (scalar and array) variable $v$, we introduce a function $v^{(i)}$ denoting the values of $v$ at states $\sigma_i$ corresponding to loop iterations $i$. Note that $v^{(0)}$ is $v_{init}$. We call $v^{(i)}$ an *extended expression* and denote the language of loop properties with extended expressions by $\mathcal{L}_{extd}$. Formulas in $\mathcal{L}_{extd}$ that are valid at any loop iteration are called *extended loop properties.* Compared to [6], we simplified $\mathcal{L}_{extd}$ as we do not use extended expressions describing loop iteration properties or update predicates over arrays. This simplification brought a significant performance increase to using symbol elimination for invariant generation (see Section 4).
**Loop analysis.** In the first step of our invariant generation procedure, we perform simple static analysis to generate extended loop properties. These formulas express (i) monotonicity properties of scalars; (ii) the transition relation of the loop by translating the guarded assignments into logical formulas; (iii) update properties of the array, and (iv) the validity of the loop condition at arbitrary loop iterations. For the loop in Figure 2, the following formula describing a monotonically increasing behavior of $k$ is one of the generated properties: $(\forall i)(0 \leq i < n \implies k^{(i+1)} = k^{(i)} + 1)$.

Compared to [6], we simplified and improved loop analysis in Vampire. In particular, array update properties expressing last updates to array positions and extended properties using loop conditions are now formulated in a way that makes them easier to handle by a first-order theorem prover, for example by introducing fewer Skolem functions. With these improvements, we generate a significantly smaller number of invariants, without loss of interesting properties (see Section 4).
**Symbol elimination.** While the properties in $\mathcal{L}_{extd}$ are valid at arbitrary loop iterations, they are not yet invariants as they use symbols that are not in $\mathcal{L}_{asrt}$ (they use extended expressions). The next step in our invariant generation process is to eliminate the symbols that are not in $\mathcal{L}_{asrt}$, 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 [10].

**Invariant filtering.** As a result of symbol elimination, a set of loop invariants is computed. While [6] returned all invariants discovered during symbol elimination, we note that not all generated invariants are relevant to the user when proving the partial correctness of the loop. In our work, we provide additional options to control the process of invariant generation, as follows. If the user provides a loop post-condition $\phi$, we add an invariant filtering step to symbol elimination by proving $\neg\psi \wedge I_1 \wedge \ldots \wedge I_k \implies \phi$, where $\psi$ is the loop condition and $I_1, \ldots, I_k$ are the invariants generated so far by symbol elimination. If proving this implication succeeds, the invariants that were effectively used in the proof are reported to the user.

Recall that invariants are logical consequences of extended loop properties, hence the loop post-condition can be proved directly from the extended loop properties. We therefore also extended loop analysis in Vampire by proving partial correctness of loops using extended loop properties, without the need for generating loop invariants by symbol elimination.

**Output.** We provide three options regarding the output of our tool. It can consist of (i) the set of all invariants generated by symbol elimination, (ii) the set of relevant invariants after filtering using the loop specification, or (iii) a partial correctness proof of the loop.

## 3 Integration with the KeY System

In this section we describe the integration of our invariant generation method to the KeY verification system. We discuss the modularity afforded by our tool and its applicability to realistic examples.

**Dynamic logic.** KeY [1] is a deductive verifier for functional correctness properties of Java source code. It uses dynamic logic (DL), a modal logic for reasoning about programs. DL extends first-order logic with the modality $[p]\phi$, where $p$ is a program and $\phi$ is another DL formula; $[p]\phi$ is true in a state from which running the program $p$, in case of termination, results in a state where $\phi$ is true.

**Symbolic execution.** KeY uses symbolic execution. For that, DL is extended by "explicit substitutions", called updates. During the symbolic execution of a program $p$, the effects of $p$ are *gradually*, from the front, turned into updates, and applied to each other. After some proof steps, an intermediate proof node may look like $\Gamma \vdash \mathcal{U}[p']\phi$, where a certain prefix of $p$ has turned into update $\mathcal{U}$, representing the effects so far, while a "remaining" program $p'$ is yet to be executed. Note that most proofs branch over case distinctions, usually triggered by Boolean expressions in the source code. The semantics of the $\Gamma \vdash \mathcal{U} \ldots$ part of a sequent is in many ways close to those of a guarded assignment in Vampire's programming model. $\Gamma$ can be understood in the same way as Vampire's guards, while updates and Vampire's assignments share the same semantics of simultaneous application. We therefore use symbolic execution to perform the translation of Java programs to Vampire's guarded command language, as follows. Given a program $p$ containing a loop, we apply symbolic execution to all instructions preceding the loop, leading to a sequent:

$$\Gamma \vdash \mathcal{U}[\texttt{while (}se\texttt{) \{ }b\texttt{ \}; }p']\phi$$

where $se$ is a *side effect-free* Java expression[2]. As a step towards employing Vampire, we launch a separate KeY proof at this point, starting from the sequent: $\Gamma, se' \vdash \mathcal{U}\mathcal{V}[b]\psi$. Here, $se'$ is the result of applying $\mathcal{U}$ to $se$, $\mathcal{V}$ is an anonymizing update [2] meant to remove information on variables modified by the loop body $b$, and $\psi$ is an uninterpreted predicate. This side proof is not meant to prove anything, but only to carry out symbolic execution of *any* iteration (hence $\mathcal{V}$) of the loop body $b$. Since $\psi$ is uninterpreted, the side proof started with this sequent cannot be completed; however, assuming that they do not themselves contain an unannotated loop, instructions of $b$ can be symbolically executed. We are then left with a proof tree containing one or more open nodes, all of which have the form: $\Gamma' \vdash \{v_1 := e_1; \ldots; v_k := e_k\}[\,]\psi$. Each of these nodes corresponds to a possible path of symbolic execution, which is transformed into a guarded assignment:

```
Gamma' -> v1 = e1; ... ; vk = ek;
```

The translation of Java programs to Vampire's model has limitations however. It is for example not yet possible to fully express heap-related properties in Vampire. Another limitation is the lack of support for unannotated loops within $b$.

**Integration.** If the user is satisfied with delegating the proof of correctness of the loop to Vampire, when the Vampire proof succeeds, it is possible to simply complete the main KeY proof by applying a dedicated axiomatic rule. If more transparency is desired, it is of course possible to import the invariants produced by Vampire (with or without invariant filtering) into KeY and use these invariants in the KeY inference rule normally used with user-annotated invariants. KeY will however need to prove that the invariants generated by Vampire are indeed invariants.

## 4 Experimental Results

We evaluated our tool on 19 challenging array benchmarks taken from academic papers [5, 6] 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

[2] More complex Boolean expressions are transformed away by KeY rules.

**Table 1.** Experimental results on loop reasoning using Vampire.

| Name | Cond. | $\Delta_{direct}$ | $\Delta_{filter}$ | $N_5$ | $N_{filter}$ |
|---|---|---|---|---|---|
| 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 | |

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. Our benchmarks are available at the URL of our tool.

We carried out two sets of experiments: (i) invariant generation, by using a guarded command representation of the benchmarks as inputs to our tool; (ii) loop analysis of realistic Java programs, by specifying the examples as Java methods with JML contracts as inputs to our tool and using our integration of invariant generation in KeY. All experiments were performed on a computer with a 2.1 GHz quad-core processor and 8GB of RAM.

Table 1 summarizes our results. The second column indicates whether the benchmark loops contain conditionals. Column $\Delta_{direct}$ 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 $\Delta_{filter}$ 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 [6] 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.

**Invariant generation.** Note that for all examples, our tool successfully generated quantified loop invariants. Moreover, when compared to the previous implementation [6] of invariant generation in Vampire, our tool brings a significant performance increase: in all examples where the implementation of [6] succeeded to generate invariants, the number of invariants generated by our tool is much less than in [6]. 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 [6]. 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 [6] is missing, the approach of [6] failed to generate quantified loop invariants over arrays. We also note that invariants generated by [6] are logical consequences of the invariants generated by our tool.

**Invariant filtering.** When evaluating our tool for proving correctness of the examples, we succeeded for 11 examples out of 19, as shown in column $\Delta_{direct}$ 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. The reason why the relevant invariants were not generated lies in the translation of the Java method into our guarded command representation: due to the limited representation of heap-related properties, the post-condition given to Vampire is weaker than the original proof obligation in KeY. This causes the invariant relevance filter to miss properties required to carry out the proof within KeY, even though the relevant invariants were generated by our tool.

When analyzing the 8 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.

## 5 Conclusion

We provide a new and fully automated tool for invariant generation, by re-implementing and improving program analysis and symbol elimination in Vampire. We also extend symbol elimination to prove partial correctness of loops. We integrated our tool with the KeY verification system, allowing automatic reasoning about realistic Java programs using first-order proving. We experimentally evaluated our tool on a number of examples coming from KeY. For future work, we intend to improve theory reasoning in Vampire. We believe that our examples coming from invariant filtering are challenging benchmarks for reasoning with quantifiers and theories, and intend to add these examples to the CASC theorem proving competition. 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. For doing so, new features and extensions of the TPTP language supported by first-order theorem provers are needed, for example the use of a first class Boolean sort as described in [9].

**Acknowledgments.** We thank Martin Hentschel for his help with KeY.

## References

1. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of object-oriented software: The KeY approach. Springer-Verlag (2007)
2. Beckert, B., Schlager, S., Schmitt, P.H.: An Improved Rule for While Loops in Deductive Program Verification. In: FMSE. LNCS, vol. 3785, pp. 315–329 (2005)
3. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL. pp. 105–118 (2011)
4. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A program analysis perspective. In: SEFM, pp. 233–247. Springer (2012)
5. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: ESOP. LNCS, vol. 6012, pp. 246–266 (2010)
6. Dragan, I., Kovács, L.: Lingva: Generating and Proving Program Properties Using Symbol Elimination. In: PSI. LNCS, vol. 8974, pp. 67–75 (2014)
7. Galeotti, J.P., Furia, C.A., E.May, Fraser, G., Zeller, A.: DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification. In: Proc. of HVC. LNCS, vol. 8855, pp. 48–53 (2014)
8. Gupta, A., Rybalchenko, A.: InvGen: An Efficient Invariant Generator. In: CAV. LNCS, vol. 5643, pp. 634–640 (2009)
9. Kotelnikov, E., Kovács, L., Voronkov, A.: A first class boolean sort in first-order theorem proving and TPTP. In: CICM (2015), to appear
10. Kovács, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays using a Theorem Prover. In: FASE. LNCS, vol. 8044, pp. 470–485 (2009)
11. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: CAV. LNCS, vol. 5503, pp. 1–35 (2013)
12. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-Based Array Invariant Generation. In: VMCAI. LNCS, vol. 7737, pp. 169–188 (2013)
13. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR. pp. 348–370. Springer (2010)