Lingva: Generating and Proving Program Properties Using Symbol Elimination

Ioan Dragan[1(✉)] and Laura Kovács[2]

[1] TU Vienna, Vienna, Austria
ioan@complang.tuwien.ac.at
[2] Chalmers, Gothenburg, Sweden

Abstract. We describe the Lingva tool for generating and proving complex program properties using the recently introduced symbol elimination method. We present implementation details and report on a large number of experiments using academic benchmarks and open-source software programs. Our experiments show that Lingva can automatically generate quantified invariants, possibly with alternation of quantifiers, over integers and arrays. Moreover, Lingva can be used to prove program properties expressing the intended behavior of programs.

1 Introduction

Safety verification of programs is a challenging task especially for programs with complex flow and, in particular, with loops or recursion. For such programs one needs additional information, in the form of loop invariants, pre- and postconditions, or interpolants, that express properties to hold at certain intermediate points of the program.

In this paper we present an automated tool for generating program properties, in particular loop invariants. Our tool, called Lingva, is based on the symbol elimination method of [9]. It requires no preliminary knowledge about program behavior, and uses symbol elimination in first-order theorem proving to automatically derive complex properties, as follows. Suppose we are given a loop L over scalar and array variables. Symbol elimination first extends the loop language L to a richer language L' by additional function and predicate symbols, such as loop counters or predicates expressing update properties of arrays at different loop iterations. Next, we derive a set P of first-order loop properties expressed in L'. The derived properties hold at any loop iteration, however they contain symbols that are not in L and hence cannot yet be used as loop invariants. Therefore, in the next step of symbol elimination, logical consequences of P are derived by eliminating the symbols from $L' \setminus L$ using first-order theorem proving. As a result, first-order loop invariants in L are inferred as logical consequences of P.

This work was partially supported by Swedish VR grant D0497701 and the Austrian research projects FWF S11410-N23 and WWTF ICT C-050.

© Springer-Verlag Berlin Heidelberg 2015
A. Voronkov and I. Virbitskaite (Eds.): PSI 2014, LNCS 8974, pp. 67–75, 2015.
DOI: 10.1007/978-3-662-46823-4_6

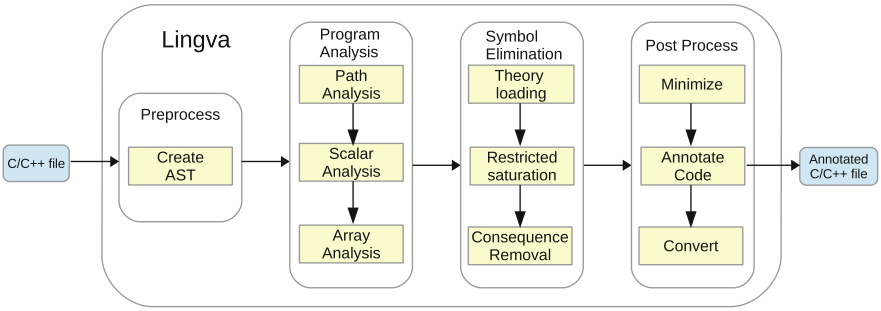

Fig. 1. The overall workflow of Lingva.

First implementation of symbol elimination was already described in [8], by using the first-order theorem prover Vampire [10]. This implementation had however various limitations: it required user-guidance for program parsing, implemented tedious translation of programs into a collection of first-order properties, had limited support for the first-order theory of arrays and the generated set of invariants could not yet be used in the context of software verification. In this paper we address these limitations and describe Lingva tool for generating loop invariants (Sect. 2). In addition to invariant generation, Lingva can also be used for proving program properties, in particular for proving program annotations from the generated set of invariants.

We evaluated Lingva on a large number of problems taken from recent research papers and open-source programs (Sect. 3). Our experiments addressed two evaluation criteria: (i) scalability, that is for how many programs Lingva successfully generated invariants; and (ii) expressiveness, that is can safety program annotation be automatically proved from the invariants generated by Lingva. The invariants inferred by Lingva are quantified properties over arrays and integers. Unlike [3,5,7,11], our invariants can express properties with quantifier alternations over the array content and exploit reasoning in the full first-order theory of arrays and integer arithmetic. In addition, in our experiments, program annotations were successfully proved by Lingva for all loops with nested conditionals. While other techniques, such as [1,6,13,14], can handle more general programs, we note that Lingva is fully automatic and requires no user guidance in the form of invariant templates, interpolants or apriori defined program properties.

2 Lingva: Tool Description

The general workflow of Lingva is summarized in Fig. 1 and detailed below. Lingva is a collection of C++ programs, glued together by Python scripts. Our implementation is available at: www.complang.tuwien.ac.at/ioan/lingva.html. Lingva can be done by executing the command: `Lingva problem.c`, where `problem.c` is a C/C++ program with loops. As a result, Lingva returns `problem.c` annotated with loop invariants.

When compared to initial implementation from [8], the preprocessing part and the code annotation and conversion parts of post processing are new features.

Further, Lingva extends that approach by more sophisticated path analysis methods and built-in support for reasoning in the first-order theory of arrays. These features allows Lingva to handle a programs with multiple loops and nested conditionals and derive quantified invariants that could not yet be obtained by [8], as arrays and integers, and their axiomatisation, were not yet supported as built-in theories in [8].

Preprocessing. Input programs of Lingva are first parsed using the Clang/LLVM infrastructure [12] and the abstract syntax tree (AST) of the input is created. Although Lingva front-end can parse arbitrary C/C++ programs, program analysis in the next step has implemented support for a restricted programming model, as follows. We only handle program loops with sequencing, assignments, and nested conditionals. Nested loops, recursive or procedure calls are thus not yet supported. Further, we only treat integers and arrays. Also we restrict program tests and assignments over integers to linear arithmetic expressions. If these restrictions are not met, Lingva terminates with an error message that provides information on the violation of the programming model.

After the AST construction, each program loop is analysed by default by Lingva. However, the user can also specify which loop or set of loops should be analysed by calling Lingva with the option `-f fn.loopNr`. Where `fn` is the name of the input's C/C++ function block and `loopNr` gives the loop number of interest within `fn`.

Example 1. Consider Fig. 2(a). It is written in C/C++ and contains multiple loops, each loop being annotated with a natural number starting from 0. For simplicity, we only show and describe Lingva on the kth loop of Fig. 2(a); analysing the other loops can be done in a similar manner. The kth loop of Fig. 2(a) takes two integer arrays aa and cc and creates an integer array bb such that each element in bb describes an array position at which the elements of aa and cc are equal. This loop is the `Partition_Init` program from Sect. 3. For running Lingva only on this loop, one should execute the command: `Lingva problem.c -f main.k`

Program Analysis. Program loops are next translated into a collection of first-order properties capturing the program behavior. These properties are formulated using the TPTP syntax [15]. Note that in TPTP, symbols starting with capital letters denote logical variables which are universally (!) or existentially (?) quantified. In the rest of the paper, we illustrate Lingva using the TPTP notation.

During program analysis, we extend the loop language with additional function and predicate symbols, as follows. For each loop, we use an extra integer constant $n \geq 0$ denoting the number of loop iterations and introduce an extra predicate $iter(X)$ expressing that the logical variable X is a valid loop iteration, that is $0 \leq X < n$. Loop variables thus become functions of loop iterations, that is a loop variable v becomes the function $v(X)$ such that $iter(X)$ holds and $v(X)$ denotes the value of v at the Xth loop iteration. For each loop variable v, we respectively denote by $v0$ and v its initial and final values. Finally, for each

Example of input file

```
void main() {
 //loop 0
 while (condition) {
       loop_body
 }
 ...
 //loop_k: Partition_Init
 int a, b, m;
 int *aa, *bb, *cc;
 while (a < m) {
    if (aa[a] == cc[a]) {
        bb[b] = a;
        b = b + 1; }
    a = a + 1;
 }
 ...
}
```

(a)

Partial result of program analysis

```
...
18. ![X1]: bb(0,X1) = bb0(X1)
17. ![X0, X2, X3]: updbb(X0,X2,X3) => bb(X2) = X3
...
9. ![X0]: iter(X0) => a(X0) = a0 + X0
8. a(0) =a0
...
2. iter(X0) =>(let a := a(X0) in (let b := b(X0)
    in ( let bb(X1) := bb(X0,X1) in a<m )))
1. ![X0, X1]: let a := a(X0) in (let b := b(X0)
   in (let bb(X1) := bb(X0,X1) in (aa(a) = cc(a) =>
   b(X0+1) = ( let bb(X1) := ite_t(b =
   X1,a,bb(X1)) in ( let b := b +1 in (let a :=
   a + 1 in b))))))
```

(b)

Fig. 2. Program analysis with Lingva on the `Partition_Init` program of Table 2.

array variable we introduce so-called update predicates describing at which loop iteration and array position the array was updated. For example, for an array *bb* we write $updbb(X, Y, Z)$ denoting that at loop iteration X the array was updated at position Y by the value Z.

For each loop, we next apply path and (scalar and array) variable analysis in order to collect valid loop properties in the extended loop language. Within path analysis, loops are translated into their guarded assignment representations and the values of program variables are computed using let-in and if-then-else formulas and terms. Unlike [8], the use of let-in formulas (`let...in`) and if-then-else terms (`ite_t`) allow us to easily express the transition relations of programs. Further, (i) we determine the set of scalar and array program variables, (ii) compute monotonicity properties of scalars by relating their values to the increasing number of loop iterations, (iii) classify arrays into constant or updated arrays, and (iv) collect update array properties. As a result, for each program loop a set of valid loop properties is derived in the extended loop language.

Example 2. Consider the kth loop of Fig. 2(a). A partial set of first-order properties generated by Lingva in the extended loop language is given in Fig. 2(b). Properties `1` and `2` are derived during path analysis. They express the value of the scalar b during the program path exhibiting the then-branch of the conditional within the loop and, respectively, the loop condition. Properties `8` and `9` are derived during scalar analysis. They state that the values of a are monotonically increasing at every loop iteration; moreover, these values are exactly defined as functions of loop iterations and the initial value $a0$ of a. Properties `17` and `18` are inferred during array analysis, and express respectively, the initial and final values of the array *bb*.

Symbol Elimination. Within symbol elimination, for each loop we derive loop invariants. For doing so, we rely on Vampire [10] and compute logical consequences of the properties derived during program analysis. To this end, we first load the built-in theories of integers and arrays. Properties with let-in and

Generated invariants using symbol elimination

```
:
:

tff(inv3,claim,![X0 :$int]:
    aa(sk1(X0))=cc(sk1(X0)) |
    ~$less(X0,$sum(b,$uminus(b0))) |
    ~$lesseq(0,X0)).

:
:

tff(inv10,claim,![X0:$int, X1:$int,X2:$int]:
    ~(sk1(X0)=X1) | ~($sum(b0,X0)=X2) |
     ~$less(X0,$sum(b,$uminus(b0))) |
    ~$lesseq(0,X0) | bb(X2) = X1) ).

:
:
```

(c)

Annotated output program

```
...
loop invariant
\forall integer X0;
  aa[sK1(X0)]==cc[sK1(X0)] ||
  !(X0<(b-b0)) || !(0<=X0);
loop invariant
\forall integer X2, integer X1;
  !(sK1(X0)=X1) || !((b0+X0)=X2)  ||
  !(X0<(b-b0)) ||
  !(0<=X0) || || bb[X2]==X1;
...
  while (a < m) {
       if (aa[a] == cc[a]) {
           bb[b] = a;
           b = b + 1;}
       a = a + 1;}
...
```

(d)

Fig. 3. Invariants and annotated code corresponding to Fig. 2(a).

if-then-else expressions are then translated into first-order properties with no let-in and if-then-else terms. Unlike the initial work from [8], Lingva supports now reasoning in the first-order theories of arrays and uses arrays as built-in data types. By using the theory axiomatisations of arrays and integers arithmetic within first-order theorem proving, Lingva implements theory-specific reasoning and simplification rules which allows to generate logically stronger invariants than [8] and to prove that some of the generated invariants are redundant (as explained in the post processing step of Lingva).

Next, we collect the additional function and predicate symbols introduced in the program analysis step of Lingva and specify them to be eliminated by the saturation algorithm of Vampire; to this end the approach of [9] is used. As a result, loop invariants are inferred. Symbol elimination within Lingva is run with a 5 s default time limit. This time limit was chosen based on our experiments with Lingva: invariants of interests could be generated by Lingva within a 5 s time limit in all examples we tried. The user may however specify a different time limit to be used by Lingva for symbol elimination.

Example 3. The partial result of symbol elimination on Fig. 2(b) is given in Fig. 3(c). The generated invariants are listed as typed first-order formulas (`tff`) in TPTP. The invariants `inv3` and `inv10` state that at every array position $b0 + X0$ at which the initial array $bb0$ was changed, the elements of aa and cc at position $bb(b0 + X0)$ are equal; recall that $b0$ is the initial value of b. Note that the generated invariants have skolem functions introduced: `sk1(X0)` denotes a skolem function of $X0$.

Post Processing. Some of the loop invariants generated by symbol elimination are redundant, that is they are implied by other invariants. In the post processing part of Lingva, we try to minimize the set of invariants by eliminating redundant ones. As proving first-order invariants redundant is undecidable, minimization in Lingva is performed using four different proving stratgies, with a 20 s default time limit for each of the strategy. The chosen strategies and their time limit

Table 1. Overview of experimental results obtained by Lingva.

Program	♯ loops	♯ analysed loops	Average ♯ invariants	Average ♯ minimized invariants
Academic benchmarks [4,8,14]	41	41	213	80
Open source archiving benchmarks	1151	150	198	62

were carefully selected based on our experiments, and they involve theory-specific simplifcation rules as well as special literal and selection functions within first-order reasoning.

After invariant minimization, Lingva converts the minimized set of invariants in the ACSL annotation language of the Frama-C framework [2]. The input program of Lingva is then annotated with these invariants and returned. The use of the ACSL syntax in Lingva, allows one to integrate the invariants generated by Lingva in the overall verification framework of Frama-C, formally annotate program loops with their invariants, and verify the correctness of the annotated program using Frama-C.

Example 4. Fig. 3(d) shows the kth loop of Fig. 2(a) annotated with its partial set of minimized invariants generated by symbol elimination.

Proving Program Properties. In addition to the default workflow given in Fig. 1, Lingva can be used not only for generating but also for proving properties. That is, given a program loop with user-annotated properties, such as postconditions, one can use Lingva to prove these properties as follows: (i) first, loop invariants are generated as described above, (ii) second, Lingva tries to prove the user-annotated property from the set of generated invariants. For proving program properties in the combined first-order theories of integers and arrays, Lingva uses Vampire.

```
1 a = b = 0;
2 while (a < m) {
3   if aa[a] == cc[a] {
4     bb[b] = a; b = b+1;}
5   a = a+1;}
6 for j=0 to b-1 do {
7   assert(aa[bb[j]]=cc[bb[j]]);
8   j= j+1;}
```

Fig. 4. Program with assertion.

Example 5. Consider the simplified program given in Fig. 4.

Note that the loop between lines 2–5 corresponds to the kth loop of Fig. 2(a). The code between lines 6–8 specifies a user-given safety assertion, corresponding to the first-order property $\forall j : 0 \leq j < b \implies aa[bb[j]] = cc[bb[j]]$. This safety assertion can be proved from the invariants generated by Lingva (see Table 2).

3 Experiments with Lingva

We evaluated Lingva on examples taken from academic research papers on invariant generation [4,8,14] as well as from open source archiving packages. Our results

Table 2. Experimental results of Lingva on some academic benchmarks with conditionals.

Loop	Min. Inv.	Program annotation	Generated invariants implying annotation

`Partition` [14]
$a = 0; b = 0; c = 0;$
`while( a < m ){`
`if( aa[a] >= 0){`
`bb[b] = aa[a];`
`b = b+1;}`
`else{`
`cc[c] = aa[a];`
`c=c+1;}`
`a = a+1;}`
	647	$\forall x : 0 \leq x < b \rightarrow$
		$bb[x] \geq 0 \wedge$
		$\exists y : 0 \leq y < a \wedge$
		$bb[x] = aa[y]$
			`inv1:`
			$\forall x_0 : \quad aa(sk4(x_0)) \geq 0 \vee$
			$\neg(0 \leq x_0) \vee b \leq x_0$
			`inv42:`
			$\forall x_0 : \quad 0 \leq sk4(x_0) \wedge sk4(x0) < a$
			`inv81:`
			$\forall x_0 : \quad \neg(0 \leq x_0) \vee b \leq x_0 \vee$
			$aa(sk4(x_0)) = bb(x_0)$

`Partition_Init` [8]
$a = b = 0;$
`while( a < m ){`
`if(aa[a] == cc[a]){`
`bb[b]=a; b=b+1;}`
`a = a+1;}`
	169	$\forall x : 0 \leq x < b \rightarrow$
		$aa[bb[x]] = cc[bb[x]]$
			`inv3:`
			$\forall x_0 : \quad \neg(0 \leq x_0) \vee \neg(x_0 < b) \vee$
			$aa(sk1(x_0)) = cc(sk1(x_0))$
			`inv10:`
			$\forall x_0, x_1, x_2 : \neg(sk1(x_0) = x_1) \vee$
			$\neg(x_0 = x_2) \vee \neg(x_0 < b)$
			$\vee \neg(0 \leq x_0) \vee bb(x_2) = x_1$

were obtained using a Lenovo W520 laptop with 8 GB of RAM and Intel Core i7 processor. All experimental results are also available on the Lingva homepage.

Table 1 summarizes our experiments. The first column lists the number of examples from each benchmark suite. The second column gives the number of problems that could be analysed by Lingva; for all these problems invariants have been generated. The third column shows the average number of generated invariants, whereas the fourth column lists the average number of invariants after minimization. Note that minimizing invariants in the post processing part of Lingva considerably decreases the number of invariants, that is 63 % in the case of academic examples and by 69 % for open source problems. In the sequel, we detail our results on each benchmark set.

Academic Benchmarks. Tables 2 and 3 describe the results of Lingva on program loops from [4,8,14], with and without conditionals. All these examples were already annotated with properties to be proven. Due to the page limit, we only list some representative examples. The first column of both tables shows the programs with their origins. The second column gives the number of generated invariants after the minimization step of Lingva. The third column states the program annotation to be proven for each program. Finally, the fourth column lists the invariants generated by Lingva which were used in proving the property of column three (similarly to Example 3). Tables 2 and 3 show that Lingva succeeded to generate complex quantified invariants over integers and arrays, some of these invariants using alternation of quantifiers[1]. We are not aware of any other tool that is able to generate invariants with quantifier alternations. We further note that all user-provided annotations were proved by Lingva, in essentially no time, by using (some of) the generated invariants.

Open Source Benchmarks. We evaluated Lingva on open source examples taken from archiving software, such as GZip, BZip, and Tar. All together we used 1151 program loops, out of which only 150 could be analysed by Lingva,

[1] De-skolemising skolem functions give invariants with quantifier alternations.

Table 3. Experimental results of Lingva on some academic benchmarks without conditionals.

Loop	Min.Inv.	Program annotation	Generated invariants implying annotation

`Initialisation` [8]
`a = 0;`
`while( a < m ){`
`aa[a]=0; a=a+1;}`
	35	$\forall x : 0 \leq x < a \rightarrow$
		$aa[x] = 0$
			`inv90:`
			$\forall x_0 : \neg(0 \leq x_0) \vee$
			$a \leq x_0 \vee aa[x_0] = 0$

`Copy` [14]
`a = 0;`
`while( a < m ){`
`bb[a]=aa[a]; a=a+1;}`
	37	$\forall x : 0 \leq x < a \rightarrow$
		$bb[x] = aa[x]$
			`inv104:`
			$\forall x_0, x_1 : \neg(0 \leq x_0) \vee a \leq x_0 \vee$
			$\neg(bb[x_0] = x_1) \vee$
			$aa[x_0] = x_1$

`Init_non_constant` [4]
`i = 0;`
`while( i < size ){`
`aa[i]=2*i+c;i=i+1;}`
	104	$\forall x : 0 \leq x < i \rightarrow$
		$aa[x] = 2 * x + c$
			`inv128:`
			$\forall x_3, x_4 : i \leq x_3 \vee \neg(0 \leq x_3)$
			$c + (2 * x_3) = aa[x_3]$

`Copy_odd` [4]
`i = 0; j = 1;`
`while( i < size ){`
`aa[j]=bb[i];`
`j++;i+=2;}`
	214	$\forall x : 0 \leq x < j \rightarrow$
		$aa(x) = bb(2 * x + 1)$
			`inv206:`
			$\forall x_3, x_4 : j \leq x_3 \vee \neg(0 \leq x_3) \vee$
			$aa[x_4] = bb[2x_3 + 1]$

`Reverse` [4]
`i = 0;`
`while( i < size ){`
`j=size-i-1;`
`aa[i]=bb[j];`
`i++;}`
	42	$\forall x : 0 \leq x < i \wedge \rightarrow$
		$aa[x] = bb[size - x - 1]$
			`inv111:`
			$\forall x_4 : \quad i \leq x_4 \vee \neg(0 \leq x_4) \vee$
			$bb[size - x_4 - 1] = aa[x_4]$

`Strlen` [4]
`i = 0;`
`while( str[i] ≠ 0 ){`
`i=i+1;}`
	26	$\forall x : 0 \leq x < i \rightarrow$
		$str(x) \neq 0.$
			`inv5`
			$\forall x_0 : i \leq x_0 \vee \neg(0 \leq x_0) \vee$
			$str(x_0) \neq 0$

as given in Table 1. The reason why Lingva failed on the other 1001 loops was that these programs contained nested loops, implemented abrupt termination, bitwise operations, used pointer arithmetic or procedure calls. We believe that extending and combining Lingva with more sophisticated program analysis methods, such as [6,7,14], would enable us to handle more general programs then we currently do.

The 150 loops on which Lingva has been successfully evaluated implemented array copy, initialization, shift and partition operations, similarly to the ones reported in our experiments with academic benchmarks. For these examples, Lingva generated quantified invariants, some with alternations of quantifiers, over integers and arrays. We were also interested to see the behavior of Lingva on these examples when it comes to proving program properties. To this end, we manually annotated these loops with properties expressing the intended behavior of the programs and used Lingva to prove these properties from the set of generated invariants. In all these 150 examples, the intended program behavior was proved by Lingva in essentially no time, underlining the strength of Lingva for generating complex invariants in a fully automated manner.

4 Conclusion

We described the Lingva tool for automatically generating and proving program properties. We reported on implementation details and presented experimental

results on academic and open-source benchmarks. Our experiments show that Lingva can generate quantified invariants, possibly with quantifier alternations, in a fully automated manner. Moreover, the generated invariants are strong enough to prove program annotations expressing the intended behavior of programs. Further work includes extending our approach in order to better integrate theory-specific reasoning engines for improving invariant minimization.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012)
2. Correnson, L., Cuoq, P., Puccetti, A., Signoles, J.: Frama-C user manual. In: CEA LIST (2010)
3. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of POPL, pp. 105–118 (2011)
4. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
5. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 813–829. Springer, Heidelberg (2013)
6. Gupta, A., Rybalchenko, A.: InvGen: an efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
7. Halbwachs, N., Peron, M.: Discovering properties about arrays in simple programs. In: Proceedings of PLDI, pp. 339–348 (2008)
8. Hoder, K., Kovács, L., Voronkov, A.: Invariant generation in vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011)
9. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
10. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013)
11. Larraz, D., Rodríguez-Carbonell, E., Rubio, A.: SMT-based array invariant generation. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 169–188. Springer, Heidelberg (2013)
12. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In Proceedings of CGO, pp. 75–88 (2004)
13. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
14. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: Proceedings of PLDI, pp. 223–234 (2009)
15. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reasoning **43**(4), 337–362 (2009)