BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Improving precision of program analysis in the 2LS framework

Martin Smutný*

Abstract

The goal of this work is to propose a way to improve precision of program analysis in the 2LS framework, based on its existing concepts, mainly template-based synthesis of invariants. 2LS is a program analysis framework for C programs, which relies on the use of an SMT solver for automatic invariant inference. One of the techniques of 2LS' main algorithm is abstract interpretation, which is used for invariant inference and thanks to its over-approximations, makes the computation of invariants easier. The proposed solution analyses the computed invariants, and identifies the parts of the invariant that cause an undecidability of the verification. Using the obtained information, the designed method is able to identify the variables of the original program, which determine whether a verification is successful. The solution can be used to locate unbounded or imprecise variables inside loops of the original program, that might be the cause of program errors. Also the output of the designed method can be used in further analyses, now focused on further constraining or refining the values of imprecise variables.

Keywords: formal verification — static analysis — 2LS framework — abstract interpretation — invariant — template-based analysis

Supplementary Material: Downloadable Code

*xsmutn13@fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Most non-expert users tend to use specifically focused software verification tools rather for debugging, than to formally prove correctness. Users who need absolute verification invest in customizing a tool that is more suited to their product. For this reason there is a need for a general tool, that would make the debugging process easier and more convenient. The aim of 2LS is to offer a free wide range tool for proving various classes of program properties even for non-expert use. The verification process of 2LS is based on computing invariants of loops and of functions of the source program by utilizing an SMT solver. These invariants are used to reason about various program properties.

When analysing complex programs, verifiers often cannot decide whether the programs are correct. For 2LS, this undecidability is usually caused by imprecise parts of computed invariants. Such computed invariants might be even tens of lines long and are not very readable, since they consist of symbolic variables instead of the original program variables.

We propose a solution in the 2LS framework, to

identify parts of the verified programs that cause problems to the verifier. By identifying imprecise parts of the computed invariants, it is possible to map the symbolic variables back to the original program. Then, the user can refine values of problematic variables (e.g., using special "assume" constructions) in the analysed program, which can help the verification to succeed. At the same time, this may help the developers of the 2LS framework with debugging when e.g. adding new features.

There are many different literature approaches to inferring invariants, many of which are not efficient enough e.g., to infer strong loop invariants. Because of 2LS' template-based approach to computing invariants, it is not only efficient in producing all kinds of invariants but also allows us to design a viable solution for our problem.

Implementation of our solution in the 2LS framework requires a specifically designed algorithm to work with its computational approach of using various classes of templates and with its representations of analysed programs. The designed solution is described in Section 4.

The described solution was implemented in 2LS and currently supports the *template polyhedra domain*, specifically the *abstract interval domain* and the *heap domain*. It is able to identify and locate imprecise variables inside loop invariants, their line of definition, and the loops in which the imprecision is caused. Implementation details and examples of verification of sample C programs with our extension are presented in Section 5.

2. Program Analysis in 2LS

2LS is a static analysis and verification tool, built upon the *CPROVER* framework [1], for analyzing sequential C programs. 2LS is currently maintained by Peter Schrammel and the company *Diffblue Ltd* [2] and is currently able to verify pointer safety, user-specified assertions, and termination properties [3]

The core algorithm of 2LS, called *kIkI*, combines *bounded model checking*, *k-induction* and *abstract interpretation* techniques into a single efficient framework [4]. The main task of abstract interpretation in 2LS is to compute so-called *inductive invariants* using templates (described in Section 2.3) in various abstract domains. Because we want to analyse the computed invariants, we will work only with the abstract interpretation part.

2.1 Abstract Interpretation

Abstract interpretation is a static analysis technique used to approximate the concrete program semantics using *abstract semantics*. To prove certain program property only, it is sufficient to approximate the program states using elements of a simpler *abstract domain*.

Abstract domain Q consists of *abstract values*, each value represents a set of *concrete values* or program states from the *concrete domain* P. An abstract interpretation I of a program with the instruction set *Instr* is a tuple [5]:

$$I = (Q, \circ, \sqsubseteq, \top, \bot, \tau) \tag{1}$$

- Q is the *abstract domain* (domain of abstract contexts)
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\circ : Q \times Q \to Q$ is the *join operator* for accumulation of abstract contexts, $(Q, \circ, \top)$ is a complete semilattice,
- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \Leftrightarrow x \circ y = y$ in $(Q, \circ, \top)$,
- $\tau : Instr \times Q \to Q$ defines the *abstract transformers* for particular instructions.

For abstract interpretation to be sound, an abstract value must describe at least all reachable concrete states in a given program location. This can be guaranteed using *Galois connection* between concrete and abstract domains [6]. Since the analysis approach of 2LS is to compute inductive invariants, each invariant is computed in a chosen abstract domain.

2.2 Verification using Inductive Invariants

As formalism to describe the analysed programs and reason about their properties, *symbolic transition systems* is used. A *program state* describes a logical interpretation of logic variables, which correspond to each program variable and the program counter. We can describe states in sets as the models of the formulae. Given a vector of variables x, we can describe the initial variable states as a predicate $Init(x)$. A *transition relation* between such states is described using a formula $Trans(x, x')$, where x is the current program state and x' is the next program state. The least fixed-point of the transition relation characterises the set of reachable states starting from the initial states $Init(x)$. An *inductive invariant* Inv is then a predicate that has the following property:

$$\forall x_0, x_1 . (Inv(x_0) \wedge Trans(x_0, x_1) \implies Inv(x_1)). \tag{2}$$

An inductive invariant describes a fixed-point of the transition relation, but not necessarily the least one. From an inductive invariant we are able to find *loop invariants* and *function summaries* by projecting it on to a subset of variables x.

Proving system safety can be seen seen as computing an inductive invariant (i.e. the set of reachable states) and checking that it is disjoint from the set of error states, denoted by the predicate $Err(x)$. Using existential second order quantification this can be formalised as [7]:

$$\exists_2 Inv.\forall x_0,x_1.(Init(x_0)\Rightarrow Inv(x_0))\wedge (Inv(x_0)\wedge Trans(x_0,x_1)\Rightarrow Inv(x_1))\wedge (Inv(x_0)\Rightarrow \neg Err(x_0)) \quad (3)$$

2.3 Template-based synthesis of invariants

For the formula 3 to be solved by iterative application of a first-order solver, the form of inductive invariant Inv is restricted to $\mathscr{T}(x,\delta)$, where $\mathscr{T}$ is known as *template*. Template is a fixed expression over program variables x and its parameters δ. Using templates, the second-order search for an invariant is turned into a first-order search for the template parameters:

$$\exists\delta\,.\,\forall x,x'\,.\,(Init(x)\Rightarrow\mathscr{T}(x,\delta))\wedge (\mathscr{T}(x,\delta)\wedge Trans(x,x')\Rightarrow\mathscr{T}(x',\delta)) \quad (4)$$

This restriction corresponds to the use of abstract domain, i.e. a template captures only relevant properties of the program state space for the analysis [8].

Negating the previous formula removes quantifier alternation, making it possible to be iteratively checked using SMT solver for different values of d as candidates for template parameters δ. The second conjunct of Formula 4 then has the form:

$$\exists x,x'\,.\,\neg(\mathscr{T}(x,d)\wedge Trans(x,x')\Rightarrow\mathscr{T}(x',d)) \quad (5)$$

Constant d corresponds to an abstract value in abstract interpretation, and represents, i.e. *concretises* to the set of all program states x, that satisfy the formula $\mathscr{T}(x,d)$ [7]. An abstract value $\bot$ corresponding to the infimum in abstract domain denotes the empty set ($\mathscr{T}(x,\bot)\equiv false$), and the supremum $\top$ denotes the whole state space ($\mathscr{T}(x,\top)\equiv true$).

The quantifier-free Formula 5 is used for the invariant inference, where the value of d is initialized to $\bot$ and the formula is iteratively solved by an SMT solver. If the formula is unsatisfiable, then an invariant has been found. If not, then the model of satisfiability is returned by the solver, representing a counterexample to the current instance of the template being an invariant. The value of the template parameter d is then refined by joining with the obtained model using the domain-specific join operator [8].

3. Encoding of Source Programs

Being built upon the CPROVER infrastructure, 2LS uses *GOTO programs* as internal representation of analysed programs. These are control flow graphs [4]. In order for 2LS to able to use an automated SMT solver, the GOTO programs are translated into an acyclic *single static assignment form*. The SSA is then a formula representing exactly the strongest post condition of running the program.

Generally, the SSA form satisfies a property that each variable is assigned to only once. In 2LS, the SSA form represents an over-approximation of the loops, by cutting the loops at the end of the loop body, thus enabling 2LS to reason about abstractions of a program with a solver.

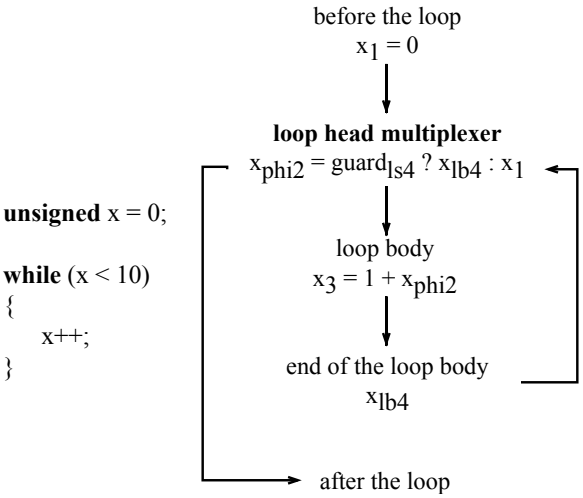

Figure 1. Encoding of the loop in SSA

Figure 1 shows the control flow expressed by SSA. The over-approximation is expressed by introducing *free variables*. Instead of passing a version of the variable x at the end of the loop (x_3) to the loop head, a free "loop-back" variable x_{lb4} is passed, describing the value of last assignment to the variable x. At the loop head, a free Boolean "loop select" variable $guard_{ls4}$ is introduced, to make a non-deterministic choice between the values coming from before the loop and from the end of the loop, thus representing a join of the values coming from before the loop and from the end of the loop body.

Precision can be improved by constraining the value of the loop-back variable x_{lb4} by means of a *loop invariant*, which is inferred during the analysis. Any property in a given abstract domain that holds at the loop entry (x_1) and at the end of the body (x_3), can be

also assumed to hold on the feedback variable x_{lb4} [7].

4. Imprecise Invariant Variables Identification and Localization

In this section, we propose a method to identify imprecise parts of computed invariants, and using existing structures in the 2LS framework, we propose an algorithm to map the identified imprecise parts back to the original program.

Firstly, for each supported abstract domain, that is the *abstract interval domain* and the *heap domain*, a method has to be devised that would enable us to identify domain-specific parts of the computed invariant that cause the undecidability of the verification, whenever the verification result is inconclusive. This involves determining for each domain its representative supremum $\top$ values and finding the template variables, that have that value in the given computed invariant.

After identifying the imprecise variables, which in essence are SSA variables, what is left, is to use the *GOTO programs* representation of the analysed program to locate the original variables that the SSA variables correspond to in the source program.

4.1 Imprecise Variable Identification Method

As described in Section 2.3, templates are used to efficiently compute loop invariants of the analysed program, which are used to constrain values of the loop-back variables. Each template domain has a defined template form, describing the program property that is being analysed. The templates of the domains we are interested in have a form of a conjunction of formulae called *template rows*. Each template row corresponds to a single SSA loop-back variable v. The row has a row parameter, called *row abstract value*. That parameter is computed during invariant inference and represents an abstraction of the value of v.

The loop invariant $Inv(x_{loop})$ is obtained by projecting the invariant, here template rows and its corresponding template row values, to a subset of variables $x_{loop} \subseteq x$ containing the loop back variables. Each template row values then are used to constrain the values of the loop-back variables. For each supported template domain in 2LS, we are looking for the row value representation of the whole domain (the supremum of the abstract domain) of the set of all x: $\mathscr{T}(x, \top) \equiv true$.

Template Polyhedra is a class of templates used for *numerical analysis* in 2LS. The analysis uses a system of linear inequalities to represet numerical values of variables [6]. The class' templates have the form $\mathscr{T} = (\mathbf{Ax} \leq \mathbf{d})$, where $\mathbf{A}$ is a matrix with fixed coefficients. Subclasses of such templates are *Intervals*, *Zones* (differences) and *Octagons*. For each r^{th} *row* of the template, a constraint is generated by the r^{th} row of matrix $\mathbf{A}$. In 2LS template expressions — variables $\mathbf{x}$ are *bit-vectors* representing signed or unsigned integers, where $\top$, denotes the whole domain of $\mathbf{x}$, and corresponds to the respective maximum values in the promoted type [7].

Using the intervals domain, we are looking for each pair of template rows x_i:

$$\begin{pmatrix} +1 \\ -1 \end{pmatrix} x_i \leq \begin{pmatrix} d_{i1} \\ d_{i2} \end{pmatrix} \tag{6}$$

where d_{i1} is the maximum row value of the type x_i at the first row and d_{i2} is the minimum value of the type x_i at the second row.

Heap Domain in 2LS is used by *heap analysis* for modelling the shape of the heap, which is the description of the reachable shapes of dynamic data structures. The heap analysis only assumes two types of memory objects — statically allocated objects (variables) and dynamically allocated objects (on the heap), where one dynamic object corresponds to one allocation site (e.g. multiple concrete heap objects allocated in a loop). Since only two types of memory objects describe the shape of the heap, the template is a conjunction of its two parts:

- The *pointer* part p describes the relation between pointer and its pointed objects. Its row value specifies either a set of addresses of objects in the memory or *null*.
- The *object* part o describes the shape of dynamic data structures on the heap using a set of reachable dynamic objects from its pointer. Its row value, leading from the object via its pointer field, is specified by a destination object and a set of (abstract) objects that it passes through [8].

In both cases, we are looking for the non-deterministic row value of each template row, which in heap domain means the $\top$ value. This can be caused either because the p or o is not initialized, therefore its value is *unknown* or is initialized but either points to *null* or its set of addresses it points to is not terminated by *null*.

4.2 Localization of Source Variables

Since 2LS is built upon the CPROVER infrastructure, the only information about the source program is in its internal representation — *GOTO programs*. The translation to SSA form is made upon the GOTO programs, and because the SSA form is over-approximation of

GOTO programs, it is not possible to use only SSA variables when referring to source program variables and lines in source code at which they are defined or used.

The task is to provide the user with a real feedback in form of a specification, at which source code lines which variables are the cause of the imprecise loop invariant. The way of such "source" information retrieval is depends on the type of the loop-back SSA variables. Generally, original variable name *Var* is converted to a loop-back SSA variable name — $Var\#lbN$, where N is the n-th SSA node where its assignment to occurs. The only exception are the dynamic variables in the heap domain, since normally dynamically allocated objects on heap are unnamed. Hence heap object name is defined as:

$$\&dynamic\_object\$i\#lbN \tag{7}$$

where i is the i-th SSA node where the dynamic object was allocated. Additionally, structured-typed heap objects' name is enriched by its field Fld via it points to its objects and in case of allocation site, by its #y-nth instantiation, then the name is defined as:

$$\&dynamic\_object\$i\#y.Fld\#lbN \tag{8}$$

Using the n-th (static variables) and the i-th (dynamic variables) occurrence of the SSA node, we can get respectively the line in GOTO program at which the SSA variable was created (was assigned to in source program) and allocated. In case of the loop-back variable that means we always get the line number after the last statement (assignment) in the loop, which ultimately due to its semantics is the line of the loop head.

5. Implementation and Examples

The designed solution in Section 4 was implemented in 2LS. The implementation of *Imprecision Invariant Identification* method is generally available for any template-based domain in 2LS. Currently the only supported domains are the abstract interval domain and the heap domain. In both domains, we are able to identify imprecise SSA variables and locate the loop in which this imprecision occurs. Additionally, in case of the dynamic objects, we are able to also locate their allocation location in the source code.

Finally, we propose for each supported domain the algorithm of the designed method from Subsection 4.1 followed by illustrative examples of its use on simple C programs.

5.1 Abstract Interval Domain

Algorithm 1 shows the designed implementation in the abstract interval domain. At line 1 template row values are initialized with values of computed invariant. Since a pair of template rows (from now on, only *rows*) corresponds to the same variable, the first row value at line 5 is saved. Then at the second row at line 8, both row values are compared to the maximum row value (*MaxRowVal*) and to the minimum row value (*MinRowVal*), respectively. The condition being true indicates the imprecise variable has been found and its name is then added to the set of variable names *VarNameSet* at line 11, which is then returned.

Algorithm 1: Interval domain, Invariant identif.

```
1  RowValues ← DomainValues
2  VarNameSet ← ∅
3  forall row ∈ TemplateRows do
4    if row is first row then
5      FirstVal ← GetRowVal(row, RowValues)
6    else
7      SecndVal ← GetRowVal(row, RowValues)
8      if FirstVal = MaxRowVal(row − 1) and
         SecndVal = MinRowVal(row) then
9        RowExpr ← expr of row
10       Name ← GetName(NameSet, RowExpr)
11       VarNameSet ← VarNameSet ∪ {Name}
12 return VarNameSet
```

5.2 Heap domain

Algorithm 2 shows the implementation in the heap domain. At line 6, each template row value *RowVal* of corresponding template row is checked whether it is *true*, which is the $\top$ value of the abstract domain. The row value is set to *true* whenever the value of the row object might be non-deterministic. Only if the template row value at line 6 is *true*, the name of the non-deterministic row (SSA variable name) is added to the set of names *VarNameSet* at line 8, which is the product of the algorithm.

Algorithm 2: Heap domain, invariant identif.

```
1  RowValues ← DomainValues
2  VarNameSet ← ∅
3  forall row ∈ TemplateRows do
4    RowExpr ← expr of row
5    RowVal ←
       RowValues[row].GetRowVal(RowExpr)
6    if RowVal is true then
7      Name ← ExprName(NameSet, RowExpr)
8      VarNameSet ← VarNameSet ∪ {Name}
9  return VarNameSet
```

5.3 Examples

Below are listed simple examples of C programs, on which we show the inferred loop invariant in 2LS along with the output of our implemented method from Section 5. The method identifies the original variables that are the cause of inconclusive verification. Using the output from our method, we can refine the values of the imprecise variables using "assume" constructions in the source program, thus making the verification successful.

Listing 1 shows operations upon two numerical variables of different types in a "never-ending" loop. Using the interval domain, the computed invariant has a form:

$$(x \leq 2147483647) \wedge (-x \leq 2147483648) \wedge \quad (9)$$
$$(y \leq 4294967295) \wedge (-y \leq 0)$$

The formula above indicates that both variables are unbounded — their values are in the whole value interval defined by their types.

```
1  void main() {
2    int x = 0;
3    unsigned y = 0;
4
5    while (x || y < 10) {
6      --x;
7      ++y;
8      x = -y - x;
9    }
10 }
```

Listing 1. Loop in C, numerical variables

Listing 2 shows the output from our implemented method in intervals domain. It found both imprecise SSA loop-back variables and specified the starting line of the loop in which now corresponding original variables have imprecise values.

```
Invariant Imprecision Identification
------------------------------------
Variables:
1: x#lb23
2: y#lb23
------------------------------------
1: Imprecise value of variable "x" at the
    end of the loop, that starts at line 5
2: Imprecise value of variable "y" at the
    end of the loop, that starts at line 5
```

Listing 2. Output of running the Listing 1

Listing 3 shows an allocation of a list in a loop (also known as *allocation site*, lines 8-11) of user-defined types and then looping through the elements of the list. The example shows both dynamic and static variables, therefore we can use the heap domain to compute loop invariants. Because the last element of the list in the loop at line 14 does not point to *null*, the address of the next variable *elem* in the loop is non-deterministic.

```
1  typedef struct elem {
2    struct elem *next;
3  } *elem_t;
4
5  void main() {
6    elem_t head, elem;
7
8    for (unsigned i = 0; i < 2; i++) {
9      elem = (elem_t) malloc(sizeof(struct
           elem));
10     elem->next = head;
11     head = elem;
12   }
13   elem = head;
14   while (elem)
15     elem = elem->next;
16 }
```

Listing 3. List of dynamically allocated objects

Using the heap domain in 2LS, we can compute loop invariants for both loops. In this case the invariant is rather complicated and long, therefore the cut-down invariant has a form:

$$(head = obj\#0 \vee head = obj\#1 \vee head = obj\#2) \wedge$$
$$(elem = obj\#0 \vee elem = obj\#1 \vee elem = obj\#2) \quad (10)$$

Where obj is $\&dynamic\_object\$27$, $head$ is the variable in Listing 3 at line 11, $elem$ is the variable at line 9. Because only these two variables have defined set of pointed objects, the other loop-back variables in both loops are non-deterministic. The names of the non-deterministic loop-back variables can be seen in the first half of the output shown in Listing 4. The other half shows the fields of structured dynamic objects (numbers 1 to 3) through which the non-deterministic values are accessed, along with the line of allocation site of these dynamic objects. The last line of output shows the imprecise static variable *elem* and the line of the start of the loop in which its imprecision is caused.

```
...
Variables:
1: dynamic_object$27#2.next#lb50
2: dynamic_object$27#1.next#lb50
3: dynamic_object$27#0.next#lb50
4: elem#lb55
------------------------------------
1: Imprecise value of "next" field of
    dynamic object "dynamic_object$27#2.
    next#lb50" allocated at line 9
2: Imprecise value of "next" field of
    dynamic object "dynamic_object$27#1.
    next#lb50" allocated at line 9
3: Imprecise value of "next" field of
    dynamic object "dynamic_object$27#0.
    next#lb50" allocated at line 9
```

```
4: Imprecise value of variable "elem" at
    the end of the loop, that starts at
    line 14
```

Listing 4. Output of running the Listing 3

The information presented from the output of implemented method can be then further used in verification to refine the variables inside the loop using 2LS' "assume" function. In case of Listing 1 both variables x and y can be assumed to hold certain values at the end of the loop at line 8, thus resulting in successful verification. In case of Listing 3, the last element in the list can be assumed to be *null* at the end of the loop at line 15, resulting in successful verification of both the dynamic objects and the static variable *elem* at the end of the loop at line 14.

6. Conclusion

Users nowadays use software verification tools rather for debugging, than to formally prove correctness. 2LS is a wide range tool, used even by non-experts for proving various classes of program properties and is able to verify a large scale of programs. 2LS successfully participates in the Software Verification Competition every year and even won a gold medal in one of the competition's category.

The verification process of 2LS is based on computing invariants, that are used to reason about various program properties. Analysis of complex program is often undecidable. For 2LS that is mainly caused by imprecise parts of computed invariants and since the invariants tend to be long and consist of symbolic variables instead of the original program variables, it is complicated to identify the problematic parts in the analysed program.

Hence we proposed a solution to identify parts of the verified programs by identifying imprecise parts of computed invariants and mapping them back to the analysed program. The solution was implemented into two 2LS abstract domains, namely in the *abstract interval domain* and in the *heap domain*. In the enclosed examples, we have shown that we are able to identify the exact imprecise variables in the source code and locate the parts of the program in which these imprecisions occur.

In the future, we would like to propose integration of this extension to the 2LS framework. This could help mainly 2LS developers in their feature development process. Also users of 2LS could use this extension on a daily basis to identify problematic parts of their programs caused by unrefined values of variables inside invariants.

Acknowledgements

I would like to very much thank my supervisor Viktor Malík for his consultations that gave me both theoretical and practical insights into this matter.

References

[1] *CPROVER - Systems Verification Group*. [Online; seen 21.01.2019].

[2] Daniel Kroening. *Diffblue*, 2018. [Online; seen 21.01.2019].

[3] Daniel Kroening and Peter Schrammel. *2LS - Static Analyzer and Verifier*. [Online; seen 21.01.2019].

[4] Peter Schrammel and Daniel Kroening. *2LS for Program Analysis (Competition Contribution)*. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *LNCS*, pages 905–907. Springer, April 2016.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th*, pages 238–252. ACM, 1977.

[6] Ondřej Lengál and Tomáš Vojnar. *Formal Analysis and Verification - Abstract Interpretation*, 2018. [Online; seen 21.01.2019]. Brno University of Technology Faculty of Information Technology.

[7] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. *Safety Verification and Refutation by k-Invariants and k-Induction*. In *Static Analysis: 22nd International Symposium*, volume 9291 of *LNCS*, pages 145–161. Springer, 2015.

[8] Viktor Malik. *Template-Based Synthesis of Heap Abstractions*. Master's thesis, Brno University of Technology, Faculty of Information Technology, Brno, 2017.