Program Verification Based on SMT Solving and Templates of Invariants

Theory of Programming Languages, 2017/2018

Viktor Malík
imalik@fit.vutbr.cz

Abstract

This work presents a novel framework for automatic program verification that combines multiple analysis techniques together. A rather specific form of program representation is used—the semantics of the analysed program is over-approximated by a quantifier-free first-order logical formula. Then, a concept of templates is used to synthesise formulae describing invariants of loops and functions in the program, with the help of an SMT solver. These can be used to verify or refute various properties of the program. We formally present fundamental principles of the approach and demonstrate the verification on simple programs.

1 Introduction

Research in software verification led to development of a large number of tools and techniques capable of analysing various properties of programs. However, most of the tools are typically very narrowly focused on a single area of analysis. They usually fail to analyse complex properties of real-life programs (e.g. verifying termination of programs using numerical and pointer variables at the same time) while still being able to scale for realistic programs. On such complex properties and programs, the tools usually give up or produce imprecise results (false positives or even false negatives).

One of the tools trying to combine multiple approaches into a single, scalable framework is 2LS. It integrates different program analysis techniques to work simultaneously and exchange information, which allows it to both prove true properties as well as find errors in programs. Due to using multiple techniques, 2LS offers a possibility to analyse different classes of program properties. Currently, it is well usable to verify termination, data-flow among numerical variables and arrays (using domains of different price and precision), or equality between pairs of variables in the given program.

The verification approach of 2LS is based on approximating the semantics of the analysed program using logical formulae. This enables 2LS to use an external SMT solver to perform analysis of the program and to reason about its properties. In order to approximate sets of all reachable program states, a well-known concept of inductive

invariants is used. To infer such invariants, a novel algorithm reducing the problem from the second-order logic to the first-order logic is introduced. Inductive invariants can be used to describe invariants of loops and of functions in the analysed program and subsequently to prove and to refute program properties.

In this work, we formally present fundamental techniques that 2LS is built on. The rest of the paper is organised as follows. Section 2 presents traditional verification techniques that are used within 2LS. They are combined into a unique efficient algorithm called kIkI described in Section 3. Next, Section 4 shows how program semantics can be approximated by logical formulae. Finally, in Section 5, we introduce the approach that 2LS uses to infer inductive invariants of programs and demonstrate how these can be used to prove correctness of programs.

2 Overview of Used Techniques

The program verification approach that underlies 2LS is based on effectively combining three different static analysis techniques together. These are namely *abstract interpretation*, *bounded model checking (BMC)*, and *k-induction*. They are all based on some form of approximation of the set of all program states reachable by the analysed program. Two basic forms of approximation are common, namely *over-* and *under-approximation*.

Over-approximating the set of all reachable program states allows the analysis to soundly verify validity of program properties—if a property holds for a superset of all reachable states, it must hold for all states of the program as well. On the other hand, under-approximating reachable program configurations is typically used to discover violations of program properties—if a property is violated in a subset of all reachable program states, it must be violated in some reachable state, too.

All of the mentioned techniques are well-established in program verification. However, 2LS uses rather specific forms of these methods, especially of the abstract interpretation. This is caused by the fact that the analysed program is viewed as a logical formula and an automatic SMT solver is used to compute program invariants and to reason about program properties.

In this section, we formally define the standard approach to these techniques. To ease formalisation, we view the analysed program as a transition system. We show how concrete semantics of the program is defined within this representation and how individual techniques approximate this semantics.

2.1 Concrete Semantics of Programs

The *concrete semantics* of a program formalises a set of all possible behaviours of the program. For transition systems, this can be defined as a set of all states that the program can get into during its execution [5].

A *program state* $\boldsymbol{x}$ is the current value of all program variables (including the program counter) and related memory (i.e. the stack and the heap). Let S be a set of program

states, and let the *transition relation* $\tau \subseteq S \times S$ define for each state a set of its possible successors in the program execution.

Assume a sequence of sets of states $S_0 S_1 \dots S_k$ such that $\forall 0 \leq i < k : (S_i, S_{i+1}) \in \tau$. We denote $S_k = \tau^k(S_0)$ the set of states *reachable from* S_0 *after* k *execution steps*. If I is the set of all possible initial states of a program, then the set of *all reachable states* S_r is the least fixed point of τ starting from I defined as:

$$S_r = \bigcup_{i \in \mathbb{N}} \tau^i(I). \tag{1}$$

With respect to the above, S_r defines the concrete semantics of the analysed program.

2.2 Programs As Logical Formulae

Since it is generally very hard to automatically analyse C programs directly, they are usually translated into a so-called *internal representation*. This is typically a simpler form of the program preserving the original concrete semantics that is easier to be used for further analysis. A very common representation are e.g. *control flow graphs (CFG)* used by the two most spread C compilers `gcc` and `clang`.

However, another form of program representation that is recently getting more and more popular in fields of program analysis and verification are *logical formulae*. The reason for this is that there exist strong automatic SAT and SMT solvers and it is very advantageous to use them to automatically reason about programs. This kind of representation is also used in the verification framework 2LS. Thus, we adapt the formalisation from the previous section to use logical formulae.

The state of a program is described by a logical interpretation of logical variables corresponding to each program variable. A set of states can be described using a formula—the states in the set are defined by models of the formula. Given a vector of variables $\boldsymbol{x}$, a predicate $\mathit{Init}(\boldsymbol{x})$ is the predicate describing the initial states. A transition relation is described as a formula $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$. From these, it is possible to determine the set of reachable states as the least fixed-point of the transition relation starting from the states described by $\mathit{Init}(\boldsymbol{x})$ as already shown in Formula 1. This is, however, difficult to compute, so instead an *inductive invariant* is used. A predicate Inv is an inductive invariant if it has the property:

$$\forall \boldsymbol{x}, \boldsymbol{x}'.(\mathit{Inv}(\boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathit{Inv}(\boldsymbol{x}')). \tag{2}$$

An inductive invariant defined as above is a description of a fixed-point of the transition relation. However, it is not guaranteed to be the least one, nor to include $\mathit{Init}(\boldsymbol{x})$. Moreover, there are predicates which are inductive invariants, but are not sufficient to be used for proving any properties of the source program (such as the predicate *true*, which describes the complete state space) [3]. That is why it is useful to compute such invariants that approach the least fixed-point, so that it is enough to use them to check a given property.

A verification task does often require showing that the set of all reachable states does not intersect with the set of error states denoted $Err(\boldsymbol{x})$. Using the concept of inductive invariants and existential second-order quantification ($\exists_2$), we can formalise it as:

$$\begin{aligned} \exists_2 Inv.\ \forall \boldsymbol{x}, \boldsymbol{x}'.\ & (Init(\boldsymbol{x}) \implies Inv(\boldsymbol{x})) \wedge \\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}')) \wedge \\ & (Inv(\boldsymbol{x}) \Rightarrow \neg Err(\boldsymbol{x})). \end{aligned} \tag{3}$$

2.3 Abstract Interpretation

Abstract interpretation is a static analysis technique that soundly approximates the concrete semantics of programs using a so-called *abstract semantics*. Generally, the set of all reachable states is not computable. However, since it is usually needed to reason about a certain program property only, to prove this property it is sufficient to approximate program states as elements of a simpler domain, called the *abstract domain*.

Having the *concrete domain* P of program states, we create the abstract domain Q. An element of the abstract domain, called an *abstract value*, corresponds to an element from the concrete domain, which is typically a set of concrete program states. Along with the abstract domain, we define two functions [6]:

- The *concretisation function* defines a mapping from an abstract value to a value of the concrete domain. Formally $\gamma : Q \to P$ and $\gamma(q)$ is a concrete value represented by q.
- The *abstraction function* defines mapping from a concrete value to an abstract value from the abstract domain. Formally $\alpha : P \to Q$ and $\alpha(p)$ is the most precise abstract value in Q whose concretisation contains p.

An abstract interpretation I of a program is then a tuple [7]:

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau^{\#}) \tag{4}$$

where

- Q is the abstract domain (along with well-defined abstraction and concretisation functions),
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\sqcup : Q \times Q \to Q$ is the *join operator*, $(Q, \sqcup, \top)$ is a complete semilattice,
- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering on $(Q, \sqcup, \top)$ defined as $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$,

- $\tau^{\#} : \mathit{Instr} \times Q \to Q$ defines the interpretation of *abstract transformers*.

The framework of abstract interpretation allows to approximate the original program semantics by computing the fixpoint of $\tau^{\#}$ in the abstract domain. The result is one abstract value (i.e. one abstract state) for each execution point of the source program. In case multiple abstract values are obtained (because of multiple execution paths entering the program location), these are accumulated into one using the join operator. The properties of the analysed program are then checked in the computed abstract values.

In order to be sound in proving program properties, an abstract value must describe at least (but not precisely) all concrete states that are reachable in the given program location. This property is ensured using a *Galois connection* between the concrete and abstract domains. We say that $(P, \leq, Q, \sqsubseteq)$ is a Galois connection if and only if $(P, \leq)$ and $(Q, \sqsubseteq)$ are partially ordered sets, and there is a following relation between abstraction and concretisation functions [7]:

$$\begin{aligned} &\forall p \in P, q \in Q : \\ &\qquad p \leq \gamma(q) \Leftrightarrow \alpha(p) \sqsubseteq q. \end{aligned} \tag{5}$$

Since the computed abstract value is an over-approximation of the set of all reachable concrete program states, abstract interpretation may generate a *false positive*. It is a situation when a property does not hold for the computed abstract semantics, but it holds for the set of all reachable program states. This incoherence is usually caused by the fact that an abstract value represents multiple concrete program states and may represent also states that are not reachable in the real program.

The analysis approach of 2LS uses inductive invariants instead of directly computing the fixed point of the transition relation. Abstract interpretation is used here in such way that the inductive invariant is computed in a chosen abstract domain. Hence, it describes a property that holds for a superset of reachable program states. Such property (and any other property logically implied by it) thus holds for the set of all reachable program states, too.

2.4 Bounded Model Checking

Bounded Model Checking (BMC) [2] is a static analysis technique that is in a way complementary to abstract interpretation. While the latter is based on an over-approximation to soundly prove program properties, BMC relies on under-approximation to find (mainly) property violations.

BMC is based on checking only program paths whose length is bounded by a certain integer k. To this end, it uses an *unwinding (unfolding) of the transition relation*. For a constant $k \in \mathbb{N}$, we introduce the k-th unwinding $T[k]$ representing k steps of the transition transition relation:

$$T[k] = \bigwedge_{i=0}^{k-1} \mathit{Trans}(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \tag{6}$$

Using the unwound transition relation as defined in Formula 6, it is possible to under-approximate the concrete program semantics by $Init(\boldsymbol{x}_0) \wedge T[k]$. This formula describes a subset of all reachable program states since only prefixes of program paths are considered. This makes BMC useful for finding property violations (i.e. bugs) in the original program.

To formalise the BMC problem, we first introduce a predicate $P[k]$ in Formula 7 describing k states being error-free.

$$P[k] = \bigwedge_{i=0}^{k-1} \neg Err(\boldsymbol{x}_i) \tag{7}$$

Using $T[k]$ and $P[k]$, we formally define Bounded Model Checking as a problem of picking a bound k and solving Formula 8 [3].

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . Init(x_0) \wedge T[k] \wedge \neg P[k+1] \tag{8}$$

If the formula is satisfiable, a property is violated in some state reachable after at most k steps. Moreover, the model of the formula represents a concrete counterexample (a witness to the property violation). On the other hand, an unsatisfiability of the formula does not necessarily imply that the property always holds since it might be violated after more than k steps. This means that BMC is generally unsound and it may produce so-called *false negatives*. It is a situation complementary to the false positive described in the previous section—a property holds for the k-th unwinding, but it does not hold for some l-th unwinding with $l > k$.

2.4.1 Incremental BMC

One of the main limitations of BMC is a need to specify an unwinding bound. This is often avoided by using a technique called *incremental bounded model checking* [8]. It uses repeated BMC checks with bound starting at 0 and being increased in a linear manner. In each step, this allows to assume that there are no errors in previous states and hence simplifies the problem to solving Formula 9.

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . Init(x_0) \wedge T[k] \wedge P[k] \wedge \neg Err(\boldsymbol{x}_k) \tag{9}$$

2.5 k-Induction

The *k-induction* technique [11] can be viewed as an extension of incremental Bounded Model Checking that is capable of proving program properties. Similarly to BMC, it uses unwinding of the transition relation which allows it to find witnesses of property violations. This approach is extended by computing a so-called *k-inductive invariant*. It is a generalisation of the concept of inductive invariants introduced in Section 2.2. A k-inductive invariant $KInv$ is a predicate having the property:

$$\forall \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k . K[k] \wedge T[k] \Rightarrow KInv(\boldsymbol{x}_k) \tag{10}$$

where

$$K[k] = \bigwedge_{i=0}^{k-1} KInv(\boldsymbol{x}_i) \tag{11}$$

Similarly to the inductive invariant described in Section 2.2, if a k-inductive invariant is implied by the k-th unwinding of the transition relation starting from the initial state, it represents a fixed-point of the transition relation. Therefore, it can be used to soundly verify a program property. Formally, a system is safe if and only if there is a k-inductive invariant $KInv$ that satisfies the property:

$$\begin{aligned} \forall \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k.(&Init(\boldsymbol{x}_0) \wedge T[k] \Rightarrow K[k]) \wedge \\ &(K[k] \wedge T[k] \Rightarrow KInv(\boldsymbol{x}_k)) \wedge \\ &(KInv(\boldsymbol{x}_k) \Rightarrow \neg Err(\boldsymbol{x}_k)) \end{aligned} \tag{12}$$

Moreover, k-inductive invariants have the following properties:

- Every inductive invariant is a 1-inductive invariant.
- Every k-inductive invariant is a $(k+1)$-inductive invariant.
- Showing that k-inductive invariant exists implies that an inductive invariant exists.
- k-inductive invariant is not necessarily an inductive invariant, usually a corresponding inductive invariant is much more complex.

Generally, finding a k-inductive invariant is simpler than finding an inductive invariant. However, it increases the complexity of the formula being checked (since unwinding of the transition relation significantly increases the formula size) [3]. In addition, it still remains to be a hard problem and hence it is useful to combine k-induction with abstract interpretation by computing k-inductive invariants in some abstract domain. Such combination is implemented by the kIkI algorithm described in the following section.

3 kIkI Algorithm

In Section 2, we presented three common techniques that are widely used for formal analysis and verification of programs. Each of the approaches is suitable for a different purpose and also contains different limitations. Generally, advantages and disadvantages of individual techniques can be summarized as follows:

Bounded Model Checking is suitable for finding violations of program properties, while providing counterexamples. However, only a small part of true properties can be proven since programs are explored up to a given bound only.

k-induction is capable of proving true properties as well as providing counterexamples for a part of property violations. However, it requires k-inductive invariants which are typically expensive to be computed.

Abstract Interpretation is designed to prove true properties of programs by computing over-approximative invariants in some abstract domain. However it suffers from a large number of false positives that cannot be distinguished from real violations of properties.

In order to make use of each technique's strengths and overcome their limitations in the same time, 2LS combines them together in a special algorithm called kIkI . The basic structure of the algorithm is illustrated by Figure 1 [3].

Initially, k is set to 1. At the beginning, initial program states are checked whether they contain errors. Then, kIkI computes a k-invariant ($KInv$) in some chosen abstract domain (i.e. using abstract interpretation) and adds assumptions that the checked property holds for all previous states (this is expressed by the predicate $P[k]$). Then, the invariant is checked whether it is sufficient to prove safety. In case a violation of the property is found, BMC is used to check if the error state is truly reachable within k steps in the original program. In case it is not reachable, the counterexample can be spurious and hence the procedure is repeated with an increased k.

Even though kIkI eliminates most of the limitations of the three approaches, there still remain a need to specify some maximal k, otherwise it might not terminate. In case the maximal k is reached and the property was neither proved nor refuted, the algorithm ends with an inconclusive result.

3.1 Incremental Solving

In order for kIkI to be efficient, it is based on a so-called *incremental solving* [9]. This technique aims at checking whether satisfiability of a problem is preserved when the clause set is incremented with new clauses. Instead of re-solving the whole problem, the information from the original problem is used to speed up the solution of the new one. The original problem (before adding the clauses) is thus considered satisfiable, and only the impact of the new clauses is checked.

In kIkI , this concept is used as follows. Instead of giving the whole $P[k]$ to the solver in each iteration of the algorithm, incremental solving allows to give only $\neg Err(\boldsymbol{x}_{k-1})$, since $P[k-1]$ is already in the clause set of the solver from the previous iteration and $P[k] = P[k-1] \wedge \neg Err(\boldsymbol{x}_{k-1})$. Analogously, $Trans(\boldsymbol{x}_{k-1}, \boldsymbol{x}_k)$ can be given instead of the whole $T[k]$ in each iteration.

4 Representation of Programs Using Logical Formulae

In order to seamlessly combine various analysis techniques together, 2LS uses logical formulae to represent analysed programs. In this section, we show how a program can be

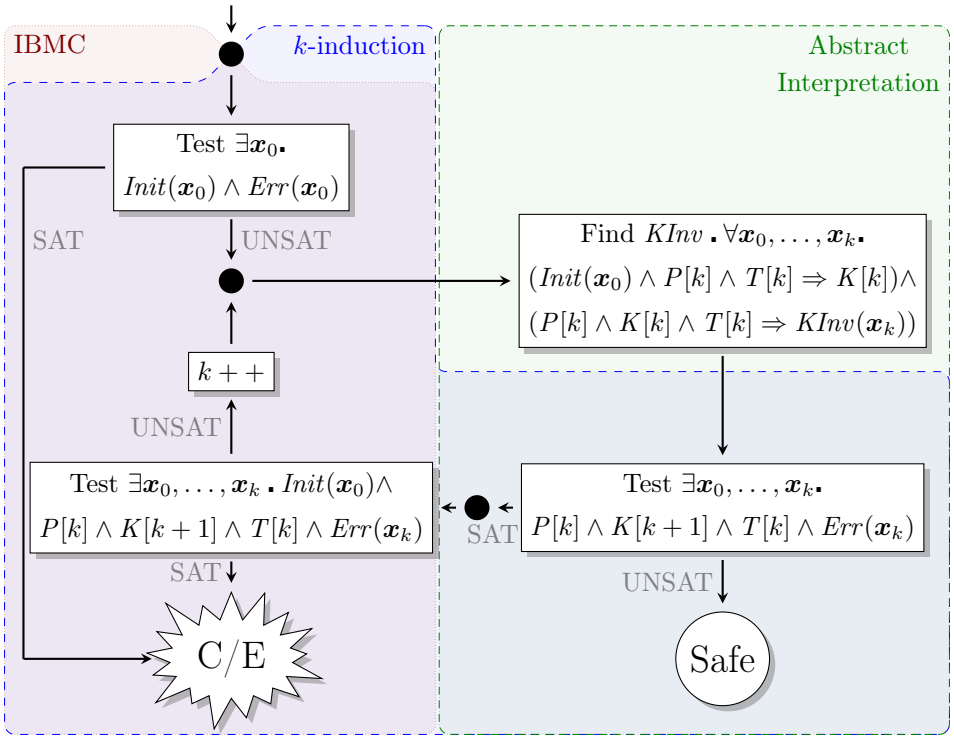


Figure 1: The *k*I*k*I algorithm [3]

translated into a quantifier-free first-order formula that over-approximates its concrete semantics. The transformation is based on *static single assignment* (*SSA*) form. It is a well-known concept of an intermediate program representation that is usable in combination with an automatic solver. We first define the general principles of the SSA form in Section 4.1 and then we present specific modifications adopted by 2LS in Section 4.2. At the end, in Section 4.3, we give an example of a translation of a program into the SSA form.

4.1 Single Static Assignment Form

Generally, single static assignment form [1] is an intermediate program representation satisfying the property that each variable is assigned at most once. A translation into the SSA form thus involves separating each variable v into several variables v_i. Every time some program location i of the original program contains an assignment to v, it is replaced by an assignment to v_i. Every R-value usage of v (i.e. every occurrence of v at the right-hand side of an assignment or in a condition) is replaced by the appropriate variable v_j where j is the last program location in which v was assigned before the given use.

In order to fulfil the single assignment property, it is required that for each program location j and each variable v, there is a unique v_i such that there are no assignments to v between i and j. This is achieved by introducing additional assignments at join points of the translated program. These are called Φ (*phi*) *nodes* and have a form of an assignment $x = \Phi(y, z)$. This expression means that x is assigned the value of y if the control reaches this program location via the first entering edge, and x is assigned the value of z if the control reaches the node via the second entering edge. For simplicity, we may assume that each join point joins exactly two program branches.

The logical formula corresponding to the original program is then a conjunction of SSA formulae for all program statements. For an acyclic code, this formula represents exactly the *strongest post condition* of running the code. In 2LS, the standard SSA form is made acyclic even for programs containing loops by over-approximating their effect. Thanks to this, the corresponding logical formula implicitly encodes the transition relation $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$. Moreover, when the formula is used in the abstract interpretation procedure, it removes the need to explicitly define abstract transformers. Details of the loop approximation are described in the following section.

4.2 SSA Form Used in 2LS

The SSA form used in 2LS extends the general concepts introduced in the previous section. The most important extensions are a specific encoding of control-flow information, and an over-approximation of loops and function calls.

4.2.1 Encoding of Control-Flow Information

A logical formula obtained from the standard SSA form, as described above, implicitly encodes the data-flow among variables by enhancing the single assignment property. However, control-flow is lost after the transformation, and hence it must be encoded explicitly. To this end, special variables called *guards* are introduced. In particular, for each program location i, a Boolean variable g_i is introduced, and its value encodes whether the program location is reachable.

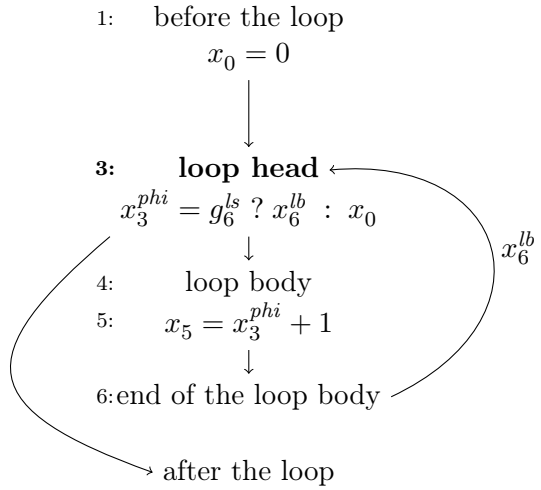
4.2.2 Over-Approximation of Loops

In order to be able to use the generated formula in an SMT solver, the SSA form used by 2LS is made acyclic by cutting loops at the end of the loop body. Then, the value of each variable x at the loop head is represented using a *phi variable* x^{phi} whose value is defined by a non-deterministic choice between the value coming from before the loop, and the value coming from the end of the loop. The latter value is represented by a newly introduced unconstrained *loop-back* variable x^{lb}. An example of this conversion is given in Figure 2.

The loop has been cut at the end of its body: instead of passing the version of x from the end of the loop body (x_5) back to the Φ node in the loop head, a free

```
1 unsigned x = 0;
2
3 while (x < 10)
4 {
5     ++x;
6 }
```

(a) A loop in C



(b) Encoding of the loop in the SSA form

Figure 2: Conversion of loops in the SSA form used in 2LS

"loop-back" variable x_6^{lb} is passed. The choice of the value of x in the Φ node is made non-deterministically using the free Boolean "loop-select" variable g_6^{ls}. This way, the SSA form remains acyclic.

Since x_6^{lb} and g_6^{ls} are free variables, this representation is an over-approximation of the actual program traces. The precision can be improved by constraining the value of x_6^{lb} by means of a *loop invariant*, which is then inferred during the analysis. A loop invariant for the variable x describes a property that holds for x at the end of the loop body, after each iteration of the loop. In the program in Figure 2, it describes a property that holds for x_5 and hence can be assumed to hold for x_6^{lb} as well.

For example, a common property that is computed when dealing with numerical variables are the lower and the upper bound of their value. For the given example, such invariant for x_6^{lb} could be:

$$x_6^{lb} \geq 1 \wedge x_6^{lb} \leq 10. \tag{13}$$

4.2.3 Function Inputs and Outputs

When dealing with real-world programs, it is usually essential to perform inter-procedural analysis (i.e. to analyse each function of the program separately). To facilitate such analysis, 2LS introduces special sets of variables in the SSA form of each function f. These are in particular:

- $\boldsymbol{x}_f^{pi}$ denoting the set of *input parameters* of the function. The set includes all

variables entering f from the caller function (i.e. formal parameters of the function and global variables[1]). When analysing f separately, their value is unconstrained.

- $\boldsymbol{x}_f^{po}$ denoting the set of *output parameters* of the function. The set includes all SSA variables that are changed within f and that can be used by the caller function (i.e. the return value and the changed global variables).

4.2.4 Over-Approximation of Function Calls

To handle inter-procedural analysis correctly, function calls are over-approximated in the SSA form used in 2LS. An invocation of a function f in a program location i is replaced by a *function placeholder* predicate $f_i(\boldsymbol{x}_i^{ai}, \boldsymbol{x}_i^{ao})$ where $\boldsymbol{x}_i^{ai}$ and $\boldsymbol{x}_i^{ao}$ are sets of input and output arguments of the call, respectively. These are analogous to $\boldsymbol{x}_f^{pi}$ and $\boldsymbol{x}_f^{po}$ described in the previous section and describe same variables from the view of the caller function:

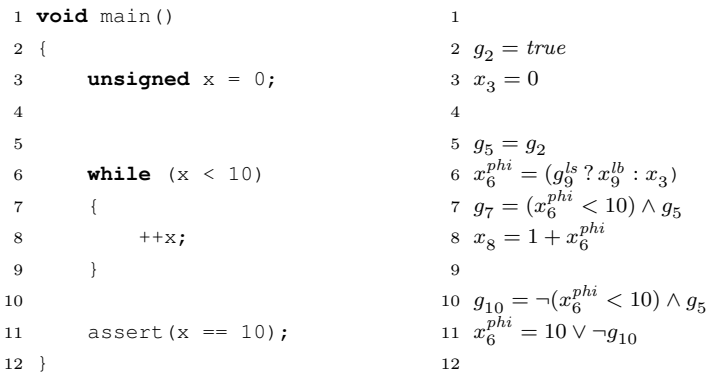
- $\boldsymbol{x}_i^{ai}$ denotes the set of *input arguments* of the call. The set includes all SSA variables that are passed to the function (i.e. the actual arguments of the call and the SSA instances of global variables in the program location i).

- $\boldsymbol{x}_i^{ao}$ denotes the set of *output arguments* of the call. The set includes fresh SSA variables that represent values changed by the called function. Their value is initially unconstrained and thus the function placeholder represents an over-approximation of the actual function effect. Its value can be later constrained with means of an invariant which is usually denoted as a *function summary*.

4.3 Example of a Program Transformation into the SSA Form

To better understand conversion of a C program, we give an example in Figure 3. Line 2 is the entry location of the program. It is always reachable, therefore the guard g_2 is set to *true*. The definition of x is done at line 3. The head of the loop contains a Φ node (line 6) and since it is directly reachable from the beginning of the `main` function, its guard g_5 is the same as the guard of the entry point (g_2). The guard g_7 at line 7 expresses that the loop body is only reachable if the loop head is reachable (g_5) and if the loop condition is true ($x_6^{phi} < 10$). Line 8 sets the new value of x. The guard g_{10} at line 10 captures the fact that the location after the loop is reachable when the loop condition is false. Finally, line 11 requires x to be equal to 10 once the assertion is reachable (i.e. g_{10} is true).

[1]Currently, 2LS does not support dealing with dynamically allocated memory, which is why we do not consider objects on the heap.



(a) The C program (b) The corresponding SSA

Figure 3: Conversion from a C program to SSA

5 Template-Based Program Verification

The most important phase of the presented kIkI algorithm is inference of a k-inductive invariant *KInv* using the abstract interpretation approach. This problem, which can be expressed in (existential fragment) of second-order logic, is reduced to a problem expressible in quantifier-free first-order logic using so-called *templates*. This reduction enables 2LS to use an SMT solver for automated inference of *loop invariants* and *function summaries*. These are then used to check various properties of the analysed program. The whole concept is focused on finite state systems since 2LS uses bit-vectors to analyse software [3].

5.1 Invariant Inference via Templates

In order to exploit the power of the kIkI algorithm, 2LS uses a solver-based approach to computing inductive invariants. Formally, search for a 1-inductive invariant is expressed by Formula 3. This is extended to a k-inductive invariant in Formula 12. For simplification purposes, the following explanation will refer to 1-inductive invariants, however, the presented concepts can be easily applied to k-inductive invariants as well, by using $T[k]$ instead of *Trans* and $K[k]$ instead of *Inv*.

To directly handle Formula 3 by a solver, 2LS would need to handle second-order logic quantification. Since a suitably general and efficient second order solver is not currently available, the problem is reduced to one that can be solved by an iterative application of a first-order solver. This reduction is done by restricting the form of the inductive invariant *Inv* to $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression (a so-called *template*) over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. This restriction corresponds to the

choice of an abstract domain in abstract interpretation—a template only captures the properties of the program state space that are relevant for the analysis. This reduces the second-order search for an invariant to a first-order search for the template parameters:

$$\begin{aligned}\exists \boldsymbol{\delta}.\, \forall \boldsymbol{x}, \boldsymbol{x}'.\, (\mathit{Init}(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \wedge \\ (\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathcal{T}(\boldsymbol{x}', \boldsymbol{\delta})).\end{aligned} \tag{14}$$

Although the problem is now expressible in first-order logic, the formula contains quantifier alternation, which poses a problem for current SMT solvers. This is solved by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\boldsymbol{d}$ as candidates for template parameters $\boldsymbol{\delta}$. For a value $\boldsymbol{d}$, the template formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ is an invariant if and only if Formula 15 is unsatisfiable.

$$\begin{aligned}\exists \boldsymbol{x}, \boldsymbol{x}'.\, \neg(\mathit{Init}(\boldsymbol{x}) \implies \mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})) \vee \\ \neg(\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \implies \mathcal{T}(\boldsymbol{x}', \boldsymbol{d}))\end{aligned} \tag{15}$$

From the abstract interpretation point of view, $\boldsymbol{d}$ is an abstract value, i.e. it represents (*concretises to*) the set of all program states $\boldsymbol{x}$ that satisfy the formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\boldsymbol{x}, \bot) \equiv \mathit{false}$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv \mathit{true}$ [3].

Formally, the concretisation function γ is same for each abstract domain:

$$\gamma(\boldsymbol{d}) = \{\boldsymbol{x} \mid \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{16}$$

As for the abstraction function, it is essential to find the most precise abstract value representing a concrete program state. Thus:

$$\alpha(\boldsymbol{x}) = \min(\boldsymbol{d}) \text{ such that } \mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \equiv \mathit{true}. \tag{17}$$

Since the abstract domain forms a partially ordered set with ordering $\sqsubseteq$ and $\mathcal{T}(\boldsymbol{x}, \top) \equiv \mathit{true}$, existence of such a minimal value $\boldsymbol{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\boldsymbol{d} = \bot$ and iteratively solves the below quantifier-free formula (corresponding to the second disjunct in Formula 15) using an SMT solver:

$$\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg(\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})). \tag{18}$$

If the formula is unsatisfiable, then an invariant has been found, otherwise the model of satisfiability is returned by the solver. The model represents a counterexample to the current instance of the template being an invariant. The value of the template parameter $\boldsymbol{d}$ is though refined by joining with the obtained model of satisfiability using the domain-specific join operator [3].

Incremental Solving Similarly to the kIkI algorithm itself, invariant inference makes use of the incremental solving technique, described in Section 3.1. Here, since $\mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}')$ does not change, it is sufficient to provide current instances of the template formula (i.e. $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \neg\mathcal{T}(\boldsymbol{x}', \boldsymbol{d})$) in each iteration of solving of Formula 18.

5.2 Guarded Templates

Traditionally, analyses based on abstract interpretation use forms of control-flow graphs (CFG) to represent the analysed programs. In a CFG, a computed invariant can be directly bound to certain program state in which it is computed and for which it holds. Since 2LS uses logical formulae generated from the SSA form, invariants cannot be used directly. Instead, so-called *guarded templates* are used.

A guarded template has a form $G \Rightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ where G is a conjunction of SSA guards that are associated with the definition of variables whose values the invariant constrains. This guarantees that the invariant can be applied for some program run if and only if the variables that it describes are defined in the given run.

In case an invariant describes multiple different variables defined in different program locations, it can be split into multiple parts where each part has its own guard defined.

5.3 Loop Invariants

Loop invariants are used to constrain values of loop-back variables defined in Section 4.2.2. These variables represent abstractions of values of program variables returning from the end of the loop body to the loop head. Hence, a loop invariant must describe a property of a variable that holds at the end of the loop body, after each iteration of the loop.

Formally, let L be the set of all loops in a program and let $\boldsymbol{x}_l$ be the set of all loop-back pointers of some loop $l \in L$. A loop invariant Inv^l is a projection of an inductive invariant Inv (describing the set of all states reachable in a program) onto a set of variables $\boldsymbol{x}^l$. The loop invariant is expressed in the form of a guarded template. A guarded template for Inv^l has the form:

$$(g_{lh} \wedge g^{ls}_{lh}) \Rightarrow \mathcal{T}(\boldsymbol{x}_l, \boldsymbol{\delta}) \tag{19}$$

where lh is the program location of the head of the loop l. The guard g_{lh} expresses that l is reachable from the beginning of the program and the guard g^{ls}_{lh} is a free loop-select variable driving the choice between values of variables coming from the loop head and from the end of the loop body (for details see Section 4.2.2). If $(g_{lh} \wedge g^{ls}_{lh})$ is equal to *true*, the loop l is reached, and the loop-back variables of l are defined and hence the loop invariant Inv^l constraining their value can be used.

Example We now illustrate the procedure of computing a loop invariant of the program given in Figure 3. Here, the set of all loop-back variables $\boldsymbol{x}_l = \{x^{lb}_9\}$. We use the

template polyhedra domain [10], particularly its subclass for approximating the *interval abstract domain* [6]. Using this domain, we compute for each variable x an interval in which all possible values of x lie. Hence, the template has the form:

$$\mathcal{T}(\{x_9^{lb}\},(d_1,d_2)) \equiv x_9^{lb} \geq d_1 \wedge x_9^{lb} \leq d_2 \tag{20}$$

where d_1 and d_2 are template parameters whose values are to be inferred during the analysis. The template form expresses the fact that all values of x_9^{lb} lie in the interval $[d_1, d_2]$.

We now show how iterative solving of Formula 18 allows 2LS to infer values of d_1 and of d_2. The transition relation $Trans(\boldsymbol{x}, \boldsymbol{x'})$ is expressed by the SSA form given in Figure 3. Formula 18 is repeatedly solved until it is unsatisfiable. We assume that $Trans(\boldsymbol{x}, \boldsymbol{x'})$ is already proven by the solver to be satisfiable (since it describes an acyclic program) and that in every iteration we only solve current instances of the invariant (since an incremental solver is used).

In Formula 18, two instances of a template are used. The instance $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ describes a loop invariant and hence we define its guarded form to be:

$$(g_5 \wedge g_9^{ls}) \Rightarrow \mathcal{T}(\{x_9^{lb}\},(d_1,d_2)) \tag{21}$$

where g_5 guards the reachability of the loop and g_9^{ls} is the loop-select variable corresponding to x_9^{lb}.

The second instance of the template $\mathcal{T}(\boldsymbol{x'}, \boldsymbol{d})$ describes the same loop invariant for the program state $\boldsymbol{x'}$ obtained after execution of the transition relation from the state $\boldsymbol{x}$. For the variable x in the analysed loop, this corresponds to the SSA variable x_8 (i.e. the state of x at the end of the loop body). Its guarded form is hence:

$$(g_5 \wedge g_7) \Rightarrow \mathcal{T}(\{x_8\},(d_1,d_2)) \tag{22}$$

where g_7 guards the reachability of the definition of x_8.

We now describe particular iterations of solving of Formula 18.

1. As stated in Section 5.1, the initial value of the template parameter $\boldsymbol{d} = \bot$ and $\mathcal{T}(\boldsymbol{x}, \bot) \equiv \mathit{false}$. The formula to solve is hence:

   $$(g_5 \wedge g_9^{ls}) \Rightarrow \mathit{false} \wedge \neg((g_5 \wedge g_7) \Rightarrow \mathit{false}). \tag{23}$$

   The only possibility to satisfy the formula is that $(g_5 \wedge g_9^{ls})$ is evaluated to false. Since $g_5 = \mathit{true}$, g_9^{ls} must be equal to false.

   In such case, $x_3 = 0$ is chosen as the value of x_6^{phi} and subsequently $x_8 = 1$. This value represents the value of x at the end of the loop body and thus it is used to refine the current invariant. Both d_1 and d_2 are updated to 1 and the current invariant is:

   $$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 1. \tag{24}$$

2. In the second iteration, we use the previously computed invariant in Formula 18. The formula to solve is:

$$\begin{aligned}(g_5 \wedge g_9^{ls}) \Rightarrow (x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 1) \wedge \\ \neg((g_5 \wedge g_7) \Rightarrow (x_8 \geq 1 \wedge x_8 \leq 1)).\end{aligned} \tag{25}$$

   In order to satisfy this formula, the solver must choose 1 as the value of x_9^{lb} and hence the value of $x_8 = 2$. Using this value to refine the template invariant causes the template parameter d_2 to be updated to 2. The current instance of the invariant is:

$$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 2. \tag{26}$$

3. Analogously to the previous step, values $3, 4, 5, \ldots$ are consecutively found as models of satisfiability and they are used to update the value of d_2. This continues until the guard g_7 is *false* and the second conjunct of the solved formula cannot be evaluated to *true* (and Formula 18 is unsatisfiable).

   Since $g_7 = (x_6^{phi} < 10) \wedge g_5$, such situation occurs for the first time when $x_9^{lb} = 10$. Hence, this is last iteration and the final computed invariant is:

$$x_9^{lb} \geq 1 \wedge x_9^{lb} \leq 10. \tag{27}$$

   The invariant can be then used to prove that the assertion $x_6^{phi} = 10 \vee \neg g_{10}$ always holds and that the analysed program is correct.

5.4 Function Summaries

Even though loop invariants are sufficient for intra-procedural analysis, additional concepts must be used when analysing functions of the original program separately. In 2LS function placeholders and function summaries are used.

A *function placeholder* $f_i(\boldsymbol{x}^{ai}, \boldsymbol{x}^{ao})$, as described in Section 4.2.4, over-approximates the effect of a call to function f in a program location i (since 2LS does not handle recursive programs, we assume that i is in function other that f). It is a predicate parametrised by two sets of variables $\boldsymbol{x}^{ai}$ and $\boldsymbol{x}^{ao}$ denoting the input and the output arguments of the call, respectively. Values of output arguments are initially unconstrained and can be constrained with means of a function summary.

A *function summary* abstracts the behaviour of a function. In other words, it describes how a function f transforms its formal inputs ($\boldsymbol{x}_f^{pi}$) into outputs ($\boldsymbol{x}_f^{po}$). In 2LS, a function summary is described by an inductive invariant over the sets of variables $\boldsymbol{x}_f^{pi}$ and $\boldsymbol{x}_f^{po}$. Formally, given a computed inductive invariant *Inv* approximating the set of all states reachable by a function f, input and output variables $\boldsymbol{x}_f^{pi}$ and $\boldsymbol{x}_f^{po}$, and

a predicate $Init_f(\boldsymbol{x})$ describing the initial states of the function f, a summary of f is a predicate Sum_f such that:

$$\begin{aligned}\forall \boldsymbol{x}, \boldsymbol{x}' : (\boldsymbol{x}_f^{pi} \subseteq \boldsymbol{x} \wedge Init(\boldsymbol{x}) \wedge \\ Inv(\boldsymbol{x}') \wedge \boldsymbol{x}_f^{po} \subseteq \boldsymbol{x}') \Longrightarrow Sum_f(\boldsymbol{x}_f^{pi}, \boldsymbol{x}_f^{po}).\end{aligned} \tag{28}$$

The first line of the implication antecedent expresses the fact that the initial states of the function depend on the set of input parameters $\boldsymbol{x}_f^{pi}$. Using a computed invariant for the output variables (second line), we define a summary $Sum(\boldsymbol{x}^{pi}, \boldsymbol{x}^{po})$ of the function. The summary can be then used to constrain a function call placeholder $f_i(\boldsymbol{x}_i^{ai}, \boldsymbol{x}_i^{ao})$ by replacing formal input and output variables $\boldsymbol{x}^{pi}$ and $\boldsymbol{x}^{po}$ in $Sum(\boldsymbol{x}^{pi}, \boldsymbol{x}^{po})$ by actual values of inputs and outputs $\boldsymbol{x}_i^{ai}$ and $\boldsymbol{x}_i^{ao}$, respectively [4].

5.4.1 Example

To better illustrate inter-procedural analysis, we show an example of analysis of a program containing a function call. The source of the program and SSA forms of both present functions are given in Figure 4.

```
1  int inc(int x)
2  {
3    int y = x + 1;
4    return y;
5  }
6
7  void main()
8  {
9      int a = 0;
10     int b = inc(a);
11     assert(b == 1);
12 }
```

(a) The C program

```
1
2  g_2 = true
3  y_3 = x_1 + 1
4  rv_4 = y_3
5
```

(b) The SSA form of `inc`

```
7
8  g_8 = true
9  a_9 = 0
10 inc_8({a_9}, {b_10})
11 b_10 = 1
```

(c) The SSA form of `main`

Figure 4: Example of a program containing a function call

The SSA form of the function `inc` contains a variable x_1 that corresponds to the input value of the function parameter x. Also, a special variable rv_4 denoting the return value of the function is introduced. The sets of input and output variables of the function can be defined as follows:

$$\begin{aligned}\boldsymbol{x}_f^{pi} &= \{x_1\} \\ \boldsymbol{x}_f^{po} &= \{rv_4\}\end{aligned} \tag{29}$$

The SSA form of `main` contains a function call placeholder $inc_8(\{a_9\}, \{b_{10}\})$ that approximates the effect of the function call. During the analysis, we first compute summary of `inc` and then we use it to constrain the function placeholder and to prove that the assertion holds.

Computing Summary We compute the summary predicate $Sum_{inc}(\{x_1\}, \{rv_4\})$ using the template-based synthesis of inductive invariants. Similarly to the previous example, we use the abstract template polyhedra domain [10] but now we use its subclass for approximating *zones abstract domain*. Here, instead of computing an interval for each individual variable, we compute for each pair of variables an interval in which their difference lies. In case of the function `inc`, important variables are x_1 and rv_4 and hence we specify the form of the template as follows:

$$\mathcal{T}(\{x_1, rv_4\}, (d_1, d_2)) \equiv (rv_4 - x_1) \geq d_1 \wedge (rv_4 - x_1) \leq d_2 \tag{30}$$

Using the algorithm for invariant inference, we compute $d_1 = d_2 = 1$. Hence, the summary $Sum_{inc}(\{x_1\}, \{rv_4\})$ is as follows:

$$Sum_{inc}(\{x_1\}, \{rv_4\}) = (rv_4 - x_1) \geq 1 \wedge (rv_4 - x_1) \leq 1 \tag{31}$$

Constraining Function Call Placeholder After the summary is computed, it can be used to constrain the value of the function call placeholder $inc_8(\{a_9\}, \{b_{10}\})$. This is done by replacing input and output parameters of the function by input and output arguments of the call. In the example, x_1 and rv_4 are replaced by a_9 and b_{10}, respectively:

$$inc_8(\{a_9\}, \{b_{10}\}) = (b_{10} - a_9) \geq 1 \wedge (b_{10} - a_9) \leq 1. \tag{32}$$

Using the constraint given in Formula 32, 2LS can prove that the assertion $b_{10} = 1$ always holds and that the program is thus correct.

6 Conclusion

In this work, we presented a novel approach to program verification. It is based on approximating the analysed program by logical formulae and on using an SMT solver to reason about program properties. Such general program representation allows to combine multiple analysis techniques together into an efficient, scalable algorithm called kIkI. We formally defined the most important concepts of the approach and demonstrated its capability to prove properties of simple programs.

References

[1] Alpern, B.; Wegman, M. N.; Zadeck, F. K.: Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 1988. pp. 1–11.

[2] Biere, A.; Cimatti, A.; Clarke, E.; et al.: Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer. 1999. pp. 193–207.

[3] Brain, M.; Joshi, S.; Kroening, D.; et al.: Safety Verification and Refutation by k-Invariants and k-Induction. In *Proceedings of the 22nd International Static Analysis Symposium*, *LNCS*, vol. 9291. Springer. 2015. pp. 145–161.

[4] Chen, H.; David, C.; Kroening, D.; et al.: Synthesising Interprocedural Bit-Precise Termination Proofs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2015. pp. 53–64.

[5] Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics: 10 Years Back, 10 Years Ahead*, edited by R. Wilhelm. Berlin, Heidelberg: Springer. 2001. pp. 138–156.

[6] Cousot, P.; Cousot, R.: Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France. 1976. pp. 106–130.

[7] Cousot, P.; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 1977. pp. 238–252.

[8] Günther, H.; Weissenbacher, G.: Incremental Bounded Software Model Checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. New York, NY, USA: ACM. 2014. pp. 40–47.

[9] Hooker, J. N.: Solving the incremental satisfiability problem. *Journal of Logic Programming*. vol. 15, no. 1&2. 1993: pp. 177–186.

[10] Sankaranarayanan, S.; Sipma, H. B.; Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer. 2005. pp. 25–41.

[11] Sheeran, M.; Singh, S.; Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the 3rd Formal Methods in Computer-Aided Design*. Berlin, Heidelberg: Springer. 2000. pp. 127–144.