Safety Verification and Refutation by k-Invariants and k-Induction

Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel(✉)

University of Oxford, Oxford, UK
{martin.brain,saurabh.joshi,daniel.kroening,
peter.schrammel}@cs.ox.ac.uk

Abstract. Most software verification tools can be classified into one of a number of established families, each of which has their own focus and strengths. For example, concrete counterexample generation in model checking, invariant inference in abstract interpretation and completeness via annotation for deductive verification. This creates a significant and fundamental usability problem as users may have to learn and use one technique to find potential problems but then need an entirely different one to show that they have been fixed. This paper presents a single, unified algorithm kIkI, which strictly generalises abstract interpretation, bounded model checking and k-induction. This not only combines the strengths of these techniques but allows them to interact and reinforce each other, giving a 'single-tool' approach to verification.

1 Introduction

The software verification literature contains a wide range of techniques which can be used to prove or disprove safety properties. These include:

Bounded Model Checking. Given sufficient time and resource, BMC will give counterexamples for all false properties, which are often of significant value for understanding the fault. However only a small proportion of true properties can be proven by BMC.

***k*-Induction.** Generalising Hoare logic's ideas of loop invariants, k-induction can prove true properties, and, in some cases provide counterexamples to false ones. However it requires inductive invariants, which can be expensive (in terms of user time, expertise and maintenance).

Abstract Interpretation. The use of over-approximations makes it easy to compute invariants which allow many true propositions to be proven. However false properties and true-but-not-provable properties may be indistinguishable. Tools may have limited support for a more complete analysis.

This research was supported by the ARTEMIS Joint Undertaking under grant agreement number 295311 (VeTeSS), the Toyota Motor Corporation and ERC project 280053 (CPROVER).

S. Blazy and T. Jensen (Eds.): SAS 2015, LNCS 9291, pp. 145–161, 2015.
DOI: 10.1007/978-3-662-48288-9_9

The range and variety of tools and techniques available is a sign of a healthy and vibrant research community but presents challenges for non-expert users. *The choice of which tools to use and where to expend effort depends on whether the properties are true or not – which is exactly what they want to find out.*

To build a robust and usable software verification system it is necessary to combine a variety of techniques. One option would be to run a series of independent tools, in parallel (as a portfolio, for example) or in some sequential order. However this limits the information that can be exchanged between the algorithms – what is needed is a genuine compound rather than a simple mixture. Another option would be to use monolithic algorithms such as CEGAR [5], IMPACT [20] or IC3/PDR [2,17] which combine some of the *ideas* of simpler systems. These are difficult to implement well as their components interact in complex and subtle ways. Also they require advanced solver features such as interpolant generation that are not widely available for all theories (bit-vectors, arrays, floating-point, etc.). In this paper, we argue for a compound with simple components and well-understood interaction.

This paper draws together a range of well-known techniques and combines them in a novel way so that they strengthen and reinforce each other. k-induction [26] uses syntactically restricted or simple invariants (such as those generated by abstract interpretation) to prove safety. Bounded model checking [1] allows us to test k-induction failures to see if they are real counter-examples or, if not, to build up a set of assumptions about system behaviour. Template-based abstract interpretation is used for invariant inference [15,23,24] with unrolling producing progressively stronger invariants. Using a solver and templates to generate invariants allows the assumptions to be used without the need for backwards propagators and 'closes the loop' allowing the techniques to strengthen each other. Specifically, the paper makes the following contributions:

1. A new, unified, simple and elegant algorithm, kIkI, for integrated invariant inference and counterexample generation is presented in Sect. 2. Incremental bounded model checking, k-induction and classical over-approximating abstract interpretation are shown to be restrictions of kIkI.
2. The techniques required to efficiently implement kIkI are given in Sect. 3 and an implementation, 2LS, is described in Sect. 4.
3. A series of experiments are given in Sect. 5. We show that kIkI *verified more programs and is faster* than a portfolio approach using incremental BMC, k-induction and abstract interpretation, showing genuine synergy between components.

2 Algorithm Concepts

This section reviews the key concepts behind kIkI. Basic familiarity with transition systems and first and second order logic will be assumed. As we intend to use kIkI to verify software using bit-vectors, we will focus on finite state systems.

2.1 Program Verification as Second Order Logic

To ease formalisation we view programs as symbolic transition systems. The state of a program is described by a logical interpretation with logical variables corresponding to each program variable, including the program counter. Formulae can be used to describe sets of states – the states in the set are the models of the formulae. Given $\boldsymbol{x}$, a vector of variables, $Start(\boldsymbol{x})$ is the predicate describing the start states. A *transition relation*, $Trans(\boldsymbol{x}, \boldsymbol{x}')$ is formula describing a relation between pairs of such interpretations which describes the (potentially non-deterministic) progression relations between states. From these we can derive the set of reachable states as the least fixed-point of the transition relation starting from the states described by $Start(\boldsymbol{x})$. Although this set is easily defined, computing a predicate that describes it (from *Start* and *Trans*) is often difficult and we will focus on the case when it is not practical. Instead *inductive invariant* are used; *Inv* is an inductive invariant if it has the following property:

$$\forall \boldsymbol{x}_0, \boldsymbol{x}_1 \,.\, (Inv(\boldsymbol{x}_0) \wedge Trans(\boldsymbol{x}_0, \boldsymbol{x}_1) \Rightarrow Inv(\boldsymbol{x}_1)) \tag{1}$$

Each inductive invariant is a description of *a* fixed-point of the transition relation but is not necessarily guaranteed to be *the least* one, nor is it guaranteed to include $Start(\boldsymbol{x})$ although many of the inductive invariants we use will do. For example, the predicate *true* is an inductive invariant for all systems as it describes the complete state space. From an inductive invariant we can find loop invariants and function and thread summaries by projecting on to a subset of variables $\boldsymbol{x}$.

Many verification tasks can be reduced to showing that the reachable states do not intersect with a set of error states, denoted by the predicate $Err(\boldsymbol{x})$. Techniques for proving systems safe can be seen as computing an inductive invariant that is disjoint from the error set. Using existential second order quantification (denoted $\exists_2$) we can formalise this as:

$$\begin{aligned} \exists_2 Inv \,.\, \forall \boldsymbol{x}_0, \boldsymbol{x}_1 \,.\, & (Start(\boldsymbol{x}_0) \Rightarrow Inv(\boldsymbol{x}_0)) \wedge \\ & (Inv(\boldsymbol{x}_0) \wedge Trans(\boldsymbol{x}_0, \boldsymbol{x}_1) \Rightarrow Inv(\boldsymbol{x}_1)) \wedge \\ & (Inv(\boldsymbol{x}_0) \Rightarrow \neg Err(\boldsymbol{x}_0)) \end{aligned} \tag{2}$$

Alternatively, if the system is not safe, then there is a reachable error state. One way of showing this is to find a concrete, n-step counterexample[1]:

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_n \,.\, Start(\boldsymbol{x}_0) \wedge \bigwedge_{i \in [0, n-1]} Trans(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \wedge Err(\boldsymbol{x}_n) \tag{3}$$

2.2 Existing Techniques

Viewing program verification as existential second-order logic allows a range of existing tools to be characterised in a common framework and thus compared

[1] If the state space is finite and the system is not safe there is necessarily a finite, concrete counterexample. For infinite state spaces there are additional issues such as errors only reachable via infinite counterexamples and which fixed-points can be described by a finite formulae.

and contrasted. This section reviews some of the more widely used approaches. The following abbreviations, corresponding to k steps of the transition system and the first k states being error free, will be used:

$$T[k] = \bigwedge_{i \in [0,k-1]} Trans(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \qquad P[k] = \bigwedge_{i \in [0,k-1]} \neg Err(\boldsymbol{x}_i)$$

Bounded Model Checking (BMC). [1] focuses on refutation by picking a *unwinding limit* k and solving:

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k \,.\, Start(\boldsymbol{x}_0) \wedge T[k] \wedge \neg P[k+1] \tag{4}$$

Models of this formula correspond to concrete counterexamples of some length $n \leqslant k$. The unwinding limit gives an *under-approximation* of the set of reachable states and thus can fail to find counterexamples that take a large number of transition steps. In practice BMC works well as the formula is existentially quantified and thus is in a fragment handled well by SAT and SMT solvers. There are also various simplifications that can reduce the number of variables (see Sect. 3.1).

Incremental BMC (IBMC) (e.g. [9]) uses repeated BMC (often optimised by using the solver incrementally) checks with increasing bounds to avoid the need for a fixed bound. If the bound starts at 0 (i.e. checking $\exists x_0 \,.\, Start(\boldsymbol{x}_0) \wedge Err(\boldsymbol{x}_0)$) and is increased linearly (this is the common use-case), then it can be assumed that there are no errors at previous states, giving a simpler test:

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k \,.\, Start(\boldsymbol{x}_0) \wedge T[k] \wedge P[k] \wedge Err(\boldsymbol{x}_k) \tag{5}$$

K-Induction [26] can be viewed as an extension of IBMC that can show system safety as well as produce counterexamples. It makes use of *k-inductive invariants*, which are predicates that have the following property:

$$\forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k \,.\, I[k] \wedge T[k] \Rightarrow KInv(\boldsymbol{x}_k) \tag{6}$$

where

$$I[k] = \bigwedge_{i \in [0,k-1]} KInv(\boldsymbol{x}_i)$$

k-inductive invariants have the following useful properties:

- Any inductive invariant is a 1-inductive invariant and vice versa.
- Any k-inductive invariant is a $(k+1)$-inductive invariant.
- A (finite) system is safe if and only if there is a k-inductive invariant $KInv$ which satisfies:

$$\begin{aligned} \forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k \,.\, & (Start(\boldsymbol{x}_0) \wedge T[k] \Rightarrow I[k]) \wedge \\ & (I[k] \wedge T[k] \Rightarrow KInv(\boldsymbol{x}_k)) \wedge \\ & (KInv(\boldsymbol{x}_k) \Rightarrow \neg Err(\boldsymbol{x}_k)) \end{aligned} \tag{7}$$

Showing that a k-inductive invariant exists is sufficient to show that an inductive invariant exists *but it does not imply that the k-inductive invariant is an inductive invariant.* Often the corresponding inductive invariant is significantly more complex. Thus k-induction can be seen as a trade-off between invariant *generation* and *checking* as it is a means to benefit as much as possible from simpler invariants by using a more complex property check.

Finding a candidate k-inductive invariant is hard so implementations often use $\neg Err(\boldsymbol{x})$. Similarly to IBMC, linearly increasing k can be used to simplify the expression by assuming there are no errors at previous states:

$$\begin{aligned}\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k \,.\, &(Start(\boldsymbol{x}_0) \wedge T[k] \wedge P[k] \wedge Err(\boldsymbol{x}_k)) \vee \\ &(T[k] \wedge P[k] \wedge Err(\boldsymbol{x}_k))\end{aligned} \tag{8}$$

A model of the first part of the disjunct is a concrete counterexample (k-induction subsumes IBMC) and if the whole formula has no models, then $\neg Err(\boldsymbol{x})$ is a k-inductive invariant and the system is safe.

Abstract Interpretation. [6] While BMC and IBMC compute under-approximations of the set of reachable states, the classical use of abstract interpretation is to compute inductive invariants that include $Start(\boldsymbol{x})$ and thus are over-approximations of the set of reachable states. Elements of an abstract domain can be understood as sets or conjuncts of formulae [8], so abstract interpretation can be seen as:

$$\begin{aligned}\exists_2 AInv \in \mathscr{A} \,.\, \forall \boldsymbol{x}, \boldsymbol{x}_1 \,.\, &(Start(\boldsymbol{x}) \Rightarrow AInv(\boldsymbol{x})) \wedge \\ &(AInv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}_1) \Rightarrow AInv(\boldsymbol{x}_1))\end{aligned} \tag{9}$$

where $\mathscr{A}$ is the set of formulae described by the chosen abstract domain. As a second step then one checks:

$$\forall \boldsymbol{x} \,.\, AInv(\boldsymbol{x}) \Rightarrow \neg Err(\boldsymbol{x}) \tag{10}$$

If this has no models then the system is safe, otherwise the safety cannot be determined without finding a more restrictive $AInv$ or increasing the set $\mathscr{A}$, i.e. choosing a more expressive abstract domain.

2.3 Our Algorithm: *k*I*k*I

The phases of the kIkI algorithm are presented as a flow chart in Fig. 1 with black arrows denoting transitions. Initially, $k = 1$ and $\mathscr{T}$ is a set of predicates that can be used as invariant with $\top \in \mathscr{T}$ (see Sect. 3 for details of how this is implemented).

After an initial test to see if any start states are errors[2], kIkI computes a k-inductive invariant that covers the initial state and includes the assumption

[2] If the transition system is derived from software and the errors are generated from assertions this will be impossible and the check can be skipped.

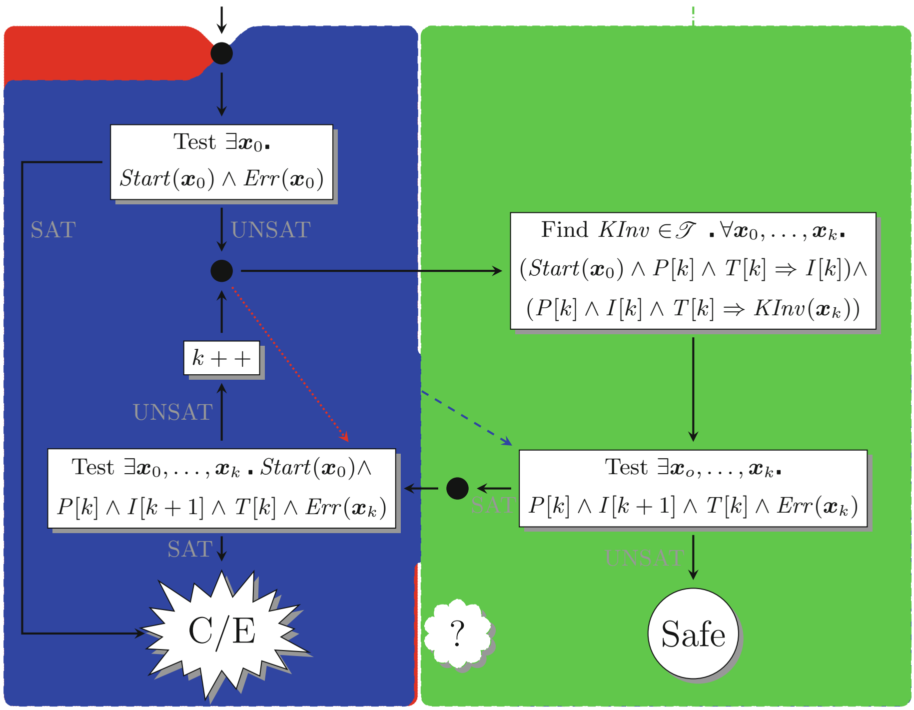


Fig. 1. The kIkI algorithm (colours in online version)

that there are no errors in earlier states. The invariant is then checked to see whether it is sufficient to show safety. If there are possible reachable error states then a second check is needed to see if the error is reachable in k steps (a genuine counterexample) or whether it is a potential artefact of a too weak invariant. In the latter case, k is incremented so that a stronger (k-)invariant can be found and the algorithm loops.

Also displayed in Fig. 1 are the steps of incremental BMC, k-induction and classical over-approximating abstract interpretation, given, respectively by the red dotted, blue dashed and green dashed/dotted boxes and arrows. kIkI can simulate k-induction by having $\mathscr{T} = \{\top\}$ and incremental BMC by over-approximating the first SAT check. Classical over-approximate abstract interpretation can be simulated by having $\mathscr{T} = \mathscr{A}$ and terminating with the result "unknown" if the first SAT check finds a model. These simulations give an intuition for the proof of the following results:

Theorem 1.

– *When kIkI terminates it gives either a k-inductive invariant sufficient to show safety or a length k counterexample.*

```
void main()
{                         guard#0 == TRUE
  unsigned x = 0;         x#0 == 0u

                          guard#1 == guard#0
  while (x<10)            x#phi1 == (guard#ls0 ? x#lb1 : x#0)
  {                       guard#2 == (x#phi1 < 10) && guard#1
    ++x;                  x#2 == 1u + x#phi1
  }
                          guard#3 == !(x#phi1 < 10) && guard#1
  assert(x==10);          x#phi1 == 10u || !guard#3
}
```

(a) The program (b) The annotated SSA

Fig. 2. Conversation from program to SSA

- *If IBMC or k-induction terminate with a length k counterexample, then kIkI will terminate with a length k counterexample.*
- *If k-induction terminates with a k-inductive invariant sufficient to show safety, then kIkI will terminate with a k-inductive invariant sufficient to show safety.*
- *If an (over-approximating) abstract interpreter returns an inductive invariant AInv that is sufficient to show safety and $\mathscr{A} \subseteq \mathscr{T}$, then kIkI will terminate with $k = 1$ and an inductive invariant sufficient to show safety.*

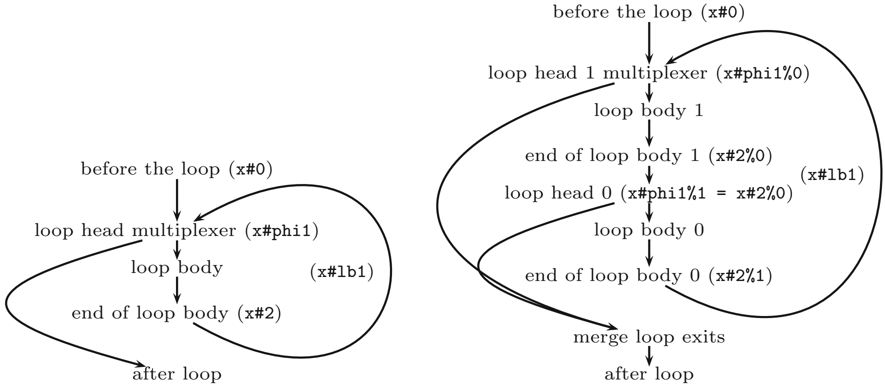
Hence kIkI strictly generalises its components by exploiting the following synergies between them: unrolling k times helps abstract interpretation to generate stronger invariants, namely k-invariants, which are further strengthened by the additional facts known from not having found a counterexample for $k-1$ iterations; stronger invariants help k-induction to successfully prove properties more often; and constraining the state space by invariants ultimately accelerates the countermodel search in BMC. We will observe these synergies also experimentally in Sect. 5.

3 Algorithm Details

Section 2 introduced kIkI but omitted a number of details which are important for implementing the algorithm efficiently. Key amongst these are the encoding from program to transition system and the generation of k-inductive invariants.

3.1 SSA Encoding

The presentation of kIkI used transition systems and it is possible to implement this directly. However the symbolic transition systems generated by software have structural properties that can be exploited. In most states the value of the program counter uniquely identifies its next value (i.e. most instructions do not branch) and most transitions update a single variable. Thus states in



(a) The SSA form of a loop. (b) The SSA loop unwinding

Fig. 3. Illustrations of various SSA encodings

the transition can be merged by substituting in the symbolic values of updated variables, so reducing the size of the formulae generated.

Rather than building the transition system and then reducing it, it is equivalent and more efficient to convert the program to *single static assignment form* (SSA). For acyclic code, the SSA is a formula that exactly represents the strongest post condition of running the code and generation of this is a standard technique found in most software BMC and Symbolic Execution tools. We extend this with an over-approximate conversion of loops so that the SSA allows us to reason about abstractions of a program with a solver.

Figure 2 gives an example of the conversion. The SSA has been made acyclic by cutting loops at the end of the loop body: the variable[3] `x#2` at the end of the loop body ("poststate") corresponds to `x#lb1`, which is fed back into the loop head ("prestate"). A non-deterministic choice (using the free Boolean variable `guard#ls0`) is introduced at the loop head in order to join the values coming from before the loop and from the end of the loop body. Figure 3a illustrates how the SSA statements express control flow.

It is easy to see that this representation "havocs" loops because `x#lb1` is a free variable – this is why its models are an over-approximation of actual program traces. Precision can be improved by constraining the feedback variable `x#lb1` by means of a *loop invariant* which we are going to infer. Any property that holds at loop entry (`x#0`) and at the end of the body (`x#2`) can then be assumed to hold on the feedback variable `x#lb1`.

Loop unwinding is performed in the usual fashion; the conversion to SSA simply repeats the conversion of the body of the loop. Figure 3b illustrates an example of this. The top-most loop head multiplexer is kept and its feedback variable is constrained with the bottom-most loop unwinding. The only subtlety

[3] Variable name suffixes are use to denote the multiple *logical* variables that correspond to a single *program* variable at different points in the execution.

is that the value of variables from different loop exits must be merged. This can be achieved by use of the `guard` variables which track the reachability of various program points for a given set of values. The unwinding that we perform is incremental, in the sense that the construction of the formula is monotonic. Assumptions have to be used to deal with the end of loop merges as there always has to be a case for "value is merged from an unwinding that has not been added yet" and this has to be assumed false.

A more significant example is given in the extended version [3].

3.2 Invariant Inference via Templates

A key phase of kIkI is the generation of *KInv*, a k-inductive invariant. Perhaps the most obvious approach is to use an off-the-shelf abstract interpreter. This works but will fail to exploit the real power of kIkI. Each iteration, kIkI unrolls loops one more step (which can improve the invariant given by an abstract interpreter) and adds assumptions that previous unwindings do not give errors. Without backwards propagation it is difficult for an abstract interpreter to make significant use of these assumptions. For example, an abstract interpretation with intervals would need backwards propagation to make use of `assume(x + y < 10)`. Thus we use a solver-based approach to computing *KInv* as it can elegantly exploit the assumptions that are added without needing to (directly) implement transformers.

Directly using a solver we would need to handle (the existential fragment of) second-order logic. As these are not currently available, we reduce to a problem that can be solved by iterative application of a first-order solver. We restrict ourselves to finding invariants *KInv* of the form $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression, a so-called *template*, over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$ (see Sect. 3.3). This restriction is analogous to choosing an abstract domain in an abstract interpreter and has similar effect – $\mathscr{T}$ only contains a the formulae that can be described by the template. Fixing a template reduces the second-order search for an invariant to the first-order search for template *parameters*:

$$\exists \boldsymbol{\delta}.\, \forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k.\, (\mathit{Start}(\boldsymbol{x}_0) \wedge T[k] \Rightarrow \mathcal{T}[k](\boldsymbol{\delta})) \wedge \\ (\mathcal{T}[k](\boldsymbol{\delta}) \wedge T[k] \Rightarrow \mathcal{T}(\boldsymbol{x_k}, \boldsymbol{\delta})) \tag{11}$$

with $\mathcal{T}[k](\boldsymbol{\delta}) = \bigwedge_{i \in [0,k-1]} \mathcal{T}(\boldsymbol{x_i}, \boldsymbol{\delta})$. Although the problem is now expressible in first-order logic, it contains quantifier alternation which poses a problem for current SMT solvers. However, we can solve this problem by iteratively checking the negated formula (to turn $\forall$ into $\exists$) for different choices of constants $\boldsymbol{d}$ for the parameters $\boldsymbol{\delta}$; as for the second conjunct in (11):

$$\exists \boldsymbol{x}_0 \ldots \boldsymbol{x}_k \,.\, \neg\big(\mathcal{T}[k](\boldsymbol{d}) \wedge T[k] \Rightarrow \mathcal{T}(\boldsymbol{x_k}, \boldsymbol{d})\big) \tag{12}$$

The resulting formula can be expressed in quantifier-free logics and efficiently solved by SMT solvers. Using this as a building block, one can solve this $\exists\forall$ problem (see Sect. 3.4).

3.3 Guarded Template Domains

As discussed in the previous section, we use templates and repeated calls (with quantifier-free formulae) to a first-order solver to compute k-inductive invariants.

An abstract value $\boldsymbol{d}$ represents, i.e. *concretises* to, the set of all $\boldsymbol{x}$ that satisfy the formula $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$. We require an abstract value $\perp$ denoting the empty set $\mathcal{T}(\boldsymbol{x}, \perp) \equiv \mathit{false}$, and $\top$ for the whole domain of $\boldsymbol{x}$: $\mathcal{T}(\boldsymbol{x}, \top) \equiv \mathit{true}$.

Template Polyhedra. We use template polyhedra [24], a class of templates for numerical variables which have the form $\mathcal{T} = (\mathbf{A}\boldsymbol{x} \le \boldsymbol{\delta})$ where $\mathbf{A}$ is a matrix with fixed coefficients. Subclasses of such templates include *Intervals*, which require constraints $\begin{pmatrix} 1 \\ -1 \end{pmatrix} x_i \le \begin{pmatrix} \delta_{i1} \\ \delta_{i2} \end{pmatrix}$ for each variable x_i, *Zones* (differences), and *Octagons* [21]. The r^{th} *row* of the template are the constraint generated by the r^{th} row of matrix $\mathbf{A}$.

In our template expressions, variables $\boldsymbol{x}$ are *bit-vectors* representing signed or unsigned integers. These variables can be mixed in template constraints. Type promotion rules are applied such that the bit-width of the types of the expressions are extended in order to avoid arithmetic under- and overflows in the template expressions. $\top$ corresponds to the respective maximum values in the promoted type, whereas $\perp$ must be encoded as a special symbol.

Guarded Templates. Since we use SSA form rather than control flow graphs, we cannot use numerical templates directly. Instead we use *guarded templates*. In a guarded template each row r is of the form $G_r \Rightarrow \widehat{\mathcal{T}}_r$ for the r^{th} row $\widehat{\mathcal{T}}_r$ of the base template domain (e.g. template polyhedra). G_r is the conjunction of the SSA guards g_i associated with the definition of variables x_i occurring in $\widehat{\mathcal{T}}_r$. G_r denotes the guard associated to variables $\boldsymbol{x}$ appearing at the loop head, and G'_r the guard associated to the variables $\boldsymbol{x}'$ at the end of the respective loop body. Hence, template rows for different loops have different guards.

A guarded template in terms of the variables at the loop head is hence of the form $\mathcal{T}(\boldsymbol{x}_0, \boldsymbol{\delta}) = \bigwedge_r G_r(\boldsymbol{x}_0) \Rightarrow \widehat{\mathcal{T}}_r(\boldsymbol{x}_0, \boldsymbol{\delta})$. Replacing parameters $\boldsymbol{\delta}$ by the values $\boldsymbol{d}$ we get the invariants $\mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ at the loop heads.

For the example program in Sect. 3.1, we have the following guarded interval template:

$$\mathcal{T}(\texttt{x\#lb1}, (\delta_1, \delta_2)) = \begin{cases} \texttt{guard\#1} \wedge \texttt{guard\#ls0} \Rightarrow \texttt{x\#lb1} \le \delta_1 \\ \texttt{guard\#1} \wedge \texttt{guard\#ls0} \Rightarrow -\texttt{x\#lb1} \le \delta_2 \end{cases}$$

We denote $\mathcal{T}'(\boldsymbol{x}_1, \boldsymbol{\delta}) = \bigwedge_r G'_r(\boldsymbol{x}_1) \Rightarrow \widehat{\mathcal{T}}_r(\boldsymbol{x}_1, \boldsymbol{\delta})$ the guarded template expressed in terms of the variables at the end of the loop body. Here, we have to express the join of the initial value at the loop head (like `x#0`) and the values that are fed back into the loop head (like `x#2`). For the example above, the corresponding guarded template is as follows:

$$\mathcal{T}'(\texttt{x\#2}, (\delta_1, \delta_2)) = \begin{cases} (pg \Leftrightarrow \texttt{guard\#2}) \;\wedge\; (ig \Leftrightarrow \texttt{guard\#1} \wedge \neg\texttt{guard\#ls0}) \wedge \\ ((ig \Rightarrow x' = \texttt{x\#0}) \wedge (pg \wedge \neg ig \Rightarrow x' = \texttt{x\#2})) \wedge \\ (pg \vee ig \Rightarrow x' \le \delta_1) \wedge (pg \vee ig \Rightarrow -x' \le \delta_2) \end{cases}$$

3.4 Accelerated Solving of the ∃∀ Problem

As discussed in Sect. 3.2, it is necessary to solve an ∃∀ problem to find values for template parameters $\boldsymbol{\delta}$ to infer invariants.

Model Enumeration. The well-known method [4,23] for solving this problem in formula (12) using SMT solvers repeatedly checks satisfiability of the formula for an abstract value $\boldsymbol{d}$ (starting with $\boldsymbol{d} = \bot$):

$$\mathcal{T}[k](\boldsymbol{d}) \wedge T[k] \wedge \neg\mathcal{T}'(\boldsymbol{x_k}, \boldsymbol{d}) \tag{13}$$

If it is unsatisfiable, then we have found an invariant; otherwise we join the model returned by the solver with the previous abstract value $\boldsymbol{d}$.

However, this method corresponds to performing a classical Kleene iteration on the abstract lattice up to convergence. Convergence is guaranteed because our abstract domains are finite. Though, the height of the lattice is enormous and even for a one loop program incrementing an unconstrained 64-bit integer variable the naïve algorithm will not terminate within human life time. Hence, we are not going to use this method.

Optimisation. What we need is a convergence acceleration that makes the computational effort *independent* from the number of states and loop iterations. To this end, we use a technique that is inspired by an encoding used by max-*strategy iteration* methods [11,12,22]. These methods state the invariant inference problem over template polyhedra as a disjunctive linear optimisation problem, which is solved iteratively by an upward iteration in the lattice of template polyhedra: using SMT solving, a conjunctive subsystem ("strategy") whose solution extends the current invariant candidate is selected. This subsystem is then solved by an LP solver; the procedure terminates as soon as an inductive invariant is found.

This method can only be used if the domain is convex and the parameter values are ordered and monotonic w.r.t. concretisation, which holds true, for example, for template polyhedra $\mathbf{A}\boldsymbol{x} \le \boldsymbol{d}$ where $\boldsymbol{d}$ is a parameter, but not for those where $\mathbf{A}$ is a parameter. If the operations in the transition relation satisfy certain properties such as monotonicity of condition predicates, then the obtained result is the least fixed point, i.e. the *same* result as the one returned by the naïve model enumeration above, but much faster on average.

Our Algorithm. We adapt this method to our setting with bit-vector variables and guarded templates. Since we deal with finite domains (bit-vectors) we can use *binary search* as optimisation method instead of an LP solver.

The algorithm proceeds as follows: We start by checking whether the current abstract value $\boldsymbol{d}$ (starting from $\boldsymbol{d} = \bot$) is inductive (Eq. (13)). If so, we have found an invariant; otherwise there are template rows R whose values are not inductive yet. We construct the system

$$\bigwedge_{i\in[0,k-1]} \left\{ \begin{array}{c} \bigwedge_{r\notin R} G_r(\boldsymbol{x}_i) \Rightarrow (e_r(\boldsymbol{x}_i) \le d_r) \\ \wedge \bigwedge_{r\in R} G_r(\boldsymbol{x}_i) \Rightarrow (e_r(\boldsymbol{x}_i) \le \delta_r) \end{array} \right\} \wedge T[k] \wedge \bigwedge_{r\in R} G'_r(\boldsymbol{x}_k) \wedge (\delta_r \le e_r(\boldsymbol{x}_k)) \tag{14}$$

where e_r is the left-hand side of the inequality corresponding to the r^{th} row of the template. Then we start the binary search for the optimal value of $\sum_{r \in R} \delta_r$ over this system. The initial bounds for $\sum_{r \in R} \delta_r$ are as follows:

- The lower bound ℓ is $\sum_{r \in R} d'_r$ where d'_r is the value of $e_r(\boldsymbol{x}_k)$ in the model of the inductivity check (13) above;
- The upper bound u is $\sum_{r \in R} \mathit{max\_value}(r)$ where $\mathit{max\_value}$ returns the maximum value that $e_r(\boldsymbol{x}_k)$ may have (dependent on variable type).

The binary search is performed by iteratively checking (14) for satisfiability under the assumption $\sum_{r \in R} \delta_r \geq m$ where $m = \mathit{median}(\ell, u)$. If satisfiable, set $\ell := m$, otherwise set $u := m$ and repeat until $\ell = u$. The values of δ_r in the last satisfiable query are assigned to d_r to obtain the new abstract value. The procedure is then repeated by testing whether $\boldsymbol{d}$ is inductive (13). Note that this algorithm uses a similar encoding for bound optimisation as strategy iteration, but potentially requires a higher number of iterations than strategy iteration. This choice has been made deliberately in order to keep the size of the generated SMT formulas small, at the cost of a potentially increased number of iterations.

A worked example is given in the extended version [3].

4 Implementation

We implemented kIkI in 2LS,[4] a verification tool built on the CPROVER framework, using MiniSAT-2.2.0 as a back-end solver (although other SAT and SMT solvers with incremental solving support can also be used). 2LS currently inlines all functions when running kIkI. The techniques described in Sect. 3 enable a single solver instance to be used where constraints and unwindings are added incrementally. This is essential because kIkI makes thousands of solver calls for invariant inference and property checks.

Our implementation is generic w.r.t. matrix $\mathbf{A}$ of the template polyhedral domain. In our experiments, we observed that very simple matrices $\mathbf{A}$ generating interval invariants are sufficient to compete with other state-of-the-art tools.

The tool can handle unrestricted sequential C programs (with the exception of programs with irreducible control flow). However, currently, invariants are not inferred over array contents or dynamically allocated data structures.

5 Experiments

We performed a number of experiments to demonstrate the utility and applicability of kIkI. All experiments were performed on an Intel Xeon X5667 at 3 GHz running Fedora 20 with 64-bit binaries. Each individual run was limited to 13 GB

[4] Version 0.2. The source code of the tool and instructions for its usage can be found on http://www.cprover.org/wiki/doku.php?id=2ls_for_program_analysis. In the experiments we ran it with the option `--competition-mode`.

of memory and 900 seconds of CPU time, enforced by the operating system kernel. We took the *loops* meta-category (143 benchmarks) from the SV-COMP'15 benchmark set.[5]

5.1 kIkI Verifies More Programs Than the Algorithms It Simulates

Table 1 gives a comparison between 2LS running kIkI (column 6) and *the same system* running as an incremental bounded model checker (IBMC) (column 2), incremental k-induction (i.e. without invariant inference, column 3) and as an abstract interpreter (AI) (column 4). kIkI is more complete than each of the restricted modes. This is not self-evident since it could be much less efficient and, thus, fail to solve the problems within the given time or memory limits. k-induction can solve 60.8 % of the benchmarks, 13 more than IBMC. 32 % of the benchmarks can be solved by abstract interpretation (bugs are only exposed if they are reachable with 0 loop unwindings). kIkI solves 62.9 % of the benchmarks, proving 3 more properties than k-induction.

Table 1. Comparison between kIkI, the algorithms it subsumes, the portfolio, and CPAchecker. The rows false alarms and false proofs indicate soundness bugs of the tool implementations.

	IBMC	k-induction	AI	portfolio	kIkI	CPAchecker	ESBMC
Counterexamples	**38**	**38**	17	**38**	**38**	36	35
Proofs	36	49	30	51	52	59	**91**
False proofs	0	0	0	0	0	2	12
False alarms	2	2	0	2	2	2	0
Inconclusive	0	0	93	0	0	4	2
Timeout	65	53	3	50	51	38	2
Memory out	2	1	0	2	0	2	1
Total runtime	17.1 h	13.8 h	0.89 h	13.3 h	13.2 h	10.9 h	0.54 h

5.2 kIkI Is at Least as Good as Their Naïve Portfolio

To show that kIkI is more than a mixture of three techniques and that they strengthen each other, consider column 5 of Table 1. This gives the results of an ideal portfolio in which the three restricted techniques are run in parallel on and the portfolio terminates when the first returns a conclusive result. Thus the CPU time taken is three times the time taken by the fastest technique for each benchmark (in practice these could be run in parallel, giving a lower *wall clock* time). In our setup, kIkI had a disadvantage as each component of virtual portfolio had the same memory restriction as kIkI, thus effectively giving the portfolio three times as much memory.

[5] http://sv-comp.sosy-lab.org/2015/benchmarks.php.

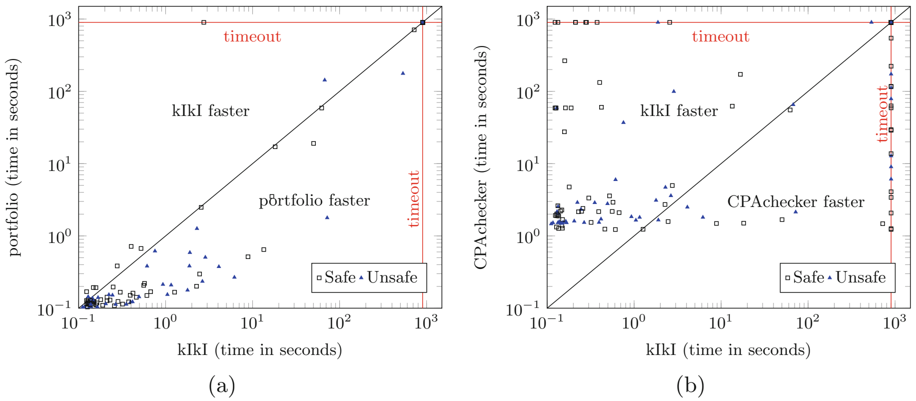


Fig. 4. Runtime comparison

Still, kIkI is slightly faster and more accurate than the portfolio as can be seen in Table 1. The scatter plot in Fig. 4a shows the results for each benchmark: one can observe that kIkI is up to one order of magnitude slower on many unsafe benchmarks, which is obviously due to the additional work of invariant inference that kIkI has to perform in contrast to IBMC. However, note that kIkI is faster than the portfolio on some safe and even one unsafe benchmarks. This suggests that kIkI is more than the sum of its parts.

5.3 *k*I*k*I Is Comparable with State-of-the-Art Approaches

We compared our implementation of kIkI with CPAchecker[6], and ESBMC[7], which uses k-induction. The results are shown in the last three columns in Table 1 and in the scatter plot in Fig. 4b. Additional results are given in the extended version [3]. In comparison to CPAchecker, the winner of SVCOMP'15, our prototype of kIkI is overall a bit slower and proves fewer properties (due to more timeouts), but as Fig. 4b shows, it significantly outperforms CPAchecker on most benchmarks. ESBMC exposes fewer bugs, but proves many more properties and is significantly faster. However, it has 6 times more soundness bugs than our implementation.[8] These results show that our prototype implementation of kIkI can keep up with state-of-the-art verification tools.

6 Related Work

Our work elucidates the connection between three well-studied techniques. Hence we can only give a brief overview of the vast amount of relevant literature.

[6] SVCOMP'15 version, http://cpachecker.sosy-lab.org/.
[7] SVCOMP'15 version, http://www.esbmc.org/.
[8] The two false alarms in our current implementation are due to limited support for dynamic memory allocation.

Since it was observed [26] that *k-induction* for finite state systems (e.g. hardware circuits) can be done by using an (incremental) SAT solver [9], it has become more and more popular also in the software community as a tool for safety proofs. Using SMT solvers, it has been applied to Lustre models [16] (monolithic transition relations) and C programs [7] (multiple and nested loops).

The idea of synthesising abstractions with the help of solvers can be traced back to predicate abstraction [13]; Reps et al. [23] proposed a method for symbolically computing best abstract transformers; these techniques were later refined [4,18,27] for application to various template domains. Using binary search for optimisation in this context was proposed by Gulwani et al. [15]. Similar techniques using LP solving for optimisation originate from strategy iteration [12]. Recently, SMT modulo optimisation [19,25] techniques were proposed that foster application to invariant generation by optimisation.

k-induction often requires additional invariants to succeed, which can be obtained by abstract interpretation. For example, Garoche et al. [10] use SMT solving to infer intermediate invariants over templates for the use in k-induction of Lustre models. As most of these approaches (except [4]), they consider (linear) arithmetic over rational numbers only, whereas our target are C programs with bit-vectors (representing machine integers, floating-point numbers, etc.). Moreover, they do not exploit the full power of the approach because they compute only 1-invariants instead of k-invariants. Another distinguishing feature of our algorithm is that it operates on a single logical representation and hence enables maximum information reuse by incremental SAT solving using a single solver.

Formalising program analysis problems such as invariant inference in second order logic and suggesting to solve these formulae with generic solvers has been considered by [14]. In this paper we provide an implementation that solves the second order formula describing the invariant inference problem by reduction to quantifier elimination of a first order formula. Our approach can also solve other problems stated in [14], e.g., termination, by considering different abstract domains, e.g., for ranking functions.

7 Conclusions

This paper presents kIkI and shows that it can simulate incremental BMC, k-induction and classical, over-approximating abstract interpretation. Experiments performed with an implementation, 2LS, show that it is not only "more" complete than each individual technique – but it also suggests that it is stronger than their naïve combination. In other words, the components of the algorithm synergistically interact and enhance each other. Moreover, our combination enables a clean, homogeneous, tightly integrated implementation rather than a loose, heterogeneous combination of isolated building blocks or a pipeline of techniques where each only strengthens the next.

There are many possible future directions for this work. Enhancing 2LS to support additional kinds of templates, possibly including disjunctive template and improving the optimisation techniques used for quantifier elimination is one area of interest. In another direction, kIkI could be enhance to support function modular, intraprocedural, thread modular and possibly multi-threaded analysis. Automatic refinement of the template domains is another tantalising possibility.

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)
2. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: Formal Methods in Computer-Aided Design, pp. 173–180. IEEE Computer Society (2007)
3. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction (extended version). Technical report (2015). arxiv.org/abs/1506.05671
4. Brauer, J., King, A., Kriener, J.: Existential quantification as incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
7. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011)
8. D'Silva, V., Kroening, D.: Abstraction of syntax. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 396–413. Springer, Heidelberg (2013)
9. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. ENTCS **89**(4), 543–560 (2003)
10. Garoche, P.-L., Kahsai, T., Tinelli, C.: Incremental invariant generation using logic-based automatic abstract transformers. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 139–154. Springer, Heidelberg (2013)
11. Gawlitza, T.M., Monniaux, D.: Improving strategies via SMT solving. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 236–255. Springer, Heidelberg (2011)
12. Gawlitza, T., Seidl, H.: Precise relational invariants through strategy iteration. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 23–40. Springer, Heidelberg (2007)
13. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
14. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416. ACM (2012)
15. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292. ACM (2008)
16. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with SMT-based techniques. In: FMCAD, pp. 1–9. IEEE Computer Society (2008)

17. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
18. Kahsai, T., Ge, Y., Tinelli, C.: Instantiation-based invariant discovery. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 192–206. Springer, Heidelberg (2011)
19. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: POPL, pp. 607–618. ACM (2014)
20. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
21. Miné, A.: The octagon abstract domain. In: Working Conference on Reverse Engineering, pp. 310–319. IEEE Computer Society (2001)
22. Monniaux, D., Schrammel, P.: Speeding up logico-numerical strategy iteration. In: Müller-Olm, M., Seidl, H. (eds.) SAS. LNCS, vol. 8723, pp. 253–267. Springer, Heidelberg (2014)
23. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
24. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
25. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 484–498. Springer, Heidelberg (2012)
26. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr, W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
27. Thakur, A., Reps, T.: A method for symbolic computation of abstract operations. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 174–192. Springer, Heidelberg (2012)