Inferring Definite Counterexamples Through Under-Approximation

Jörg Brauer[1,⋆] and Axel Simon[2,⋆⋆]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Informatik 2, Technical University Munich, Germany

Abstract. Abstract interpretation for proving safety properties summarizes concrete traces into abstract states, thereby trading the ability to distinguish traces for tractability. Given a violation of a safety property, it is thus unclear which trace led to the violation. Moreover, since part of the abstract state is over-approximate, such a trace may not exist at all. We propose a novel backward analysis that is based on abduction of propositional Boolean logic and that only generates legitimate traces that reveal actual defects. The key to tractability lies in modifying an existing projection algorithm to stop prematurely with an under-approximation and by combining various algorithmic techniques to handle loops finitely.

1 Introduction

Model checking has the attractive property that, once a specification cannot be verified, a trace illustrating a counterexample is returned which can be inspected by the user. These traces have been highlighted as invaluable for fixing the defect [9]. In contrast, abstract interpretation for asserting safety properties typically summarizes traces into abstract states, thereby trading the ability to distinguish traces for computational tractability. Upon encountering a violation of the specification, it is then unclear which trace led to the violation. Moreover, since the abstract state is an over-approximation of the set of actually reachable states, a trace leading to an erroneous abstract state may not exist at all.

Given a safety property that cannot be proved correct, a trace to the beginning of the program would be similarly instructive to the user as in model checking. However, obtaining such a trace is hard as this trace needs to be constructed by going backwards step-by-step, starting at the property violation. One approach is to apply the abstract transfer functions that were used in the forward analysis in reverse [28]. However, these transfer functions over-approximate. Thus, a counterexample computed using this approach may therefore be spurious, too. However, spurious warnings are the major hinderance of many static analyses, except those crafted for a specific application domain [11]. It has even been noted that unsound static analyses might be preferable over sound ones because the

⋆ Supported partly by DFG GRK 1298 and DFG EXC 89.
⋆⋆ Supported by DFG Emmy Noether SI 1579/1.

number of false positives can be traded off against missed bugs, thereby delivering tools that find defects rather than prove their absence [3].

Rather than giving up on soundness, we propose a practical technique to find legitimate traces that reveal actual defects, thereby turning sound static analyses into practical bug-finding tools. We use the results of an approximate forward analysis to guide a backward analysis that builds up a trace from the violation of the property to the beginning of the program. At its core, it uses a novel SAT-based projection algorithm that has been adapted to deliver an under-approximation of the transition relation in case the exact solution would be too expensive to compute. Furthermore, assuming that the projection is exact, if the intersection between a backward propagated state and the states of the forward analysis is empty on all paths, the analysis has identified a warning as spurious. Hence our analysis has the ability to both, find true counterexamples and to identify warnings as spurious. To our knowledge, our work is the first to remove spurious warnings without refining or enhancing the abstract domain.

One challenge to the inference of backward traces is the judicious treatment of loops. Given a state s' after a loop, it is non-trivial to infer a state s that is valid prior to entering the loop. In particular, it is necessary to assess how often the loop body needs to be executed to reach the exit state s'. This problem is exacerbated whenever analyzing several loops that are nested or appear in sequence. Our solution to this issue is to summarize multiple loop iterations in a closed Boolean formula and to use iterative deepening in the number of loop executions across all loops until a feasible path between s to s' is found.
The practicality of our approach is based on the following technical contributions:

– We use an over-approximating affine analysis between the backward propagated state s' after the loop and the precondition s of the loop inferred by the forward analysis to estimate the number of loop iterations. If an affine relationship exists, we derive a minimum number of loop iterations that the state s' has to be transformed by the loop.
– We synthesize a relational Boolean loop transformer f^{2^i}, which expresses 2^i executions of a loop, given $f^{2^{i-1}}$. These loop transformers are then used to construct f^n for arbitrary n, thereby providing the transfer function to calculate an input state from the given output state of the loop in $\log_2(n)$ steps for n iterations. This approach can also be applied to nested loops.
– We provide a summarization technique, which describes $0, \ldots, 2^n$ iterations of a loop as one input/output relation. This method combines the Boolean transfer functions f^{2^n} with a SAT-based existential elimination algorithm. The force of this combination is that we can modify the elimination algorithm to generate under-approximate state descriptions — any approximated result thus still describes states which are possible in a concrete execution.

The remainder of the paper is structured as follows: After the next section details our overall analysis strategy, Sect. 3 illustrates the three contributions in turn. Section 4 details the modifications to the projection algorithm to allow under-approximations which Sect. 5 evaluates in our implementation. Section 6 presents related work before Sect. 7 discusses possible future work and concludes.

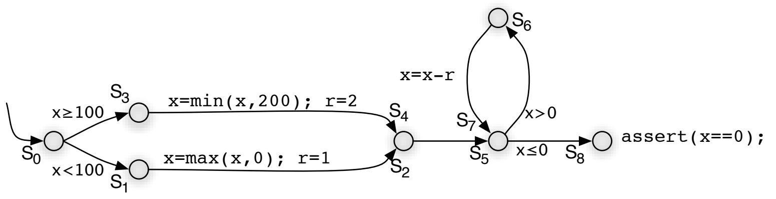

Fig. 1. Backward propagation past a loop

i		0	1	2	3	4	5	6	7	8
$S_i =$	x	any	≤ 99	$[0, 99]$	≥ 100	$[100, 200]$	$[0, 200]$	$[1, 200]$	$[-1, 199]$	$[-1, 0]$
	r	any	any	$[1, 1]$	any	$[2, 2]$	$[1, 2]$	$[1, 2]$	$[1, 2]$	$[1, 2]$
$S'_i =$	x			$\perp$		$\perp$	1	$[0, 1]$	-1	-1
	r						2	$[1, 2]$	$[1, 2]$	$[1, 2]$
$S''_i =$	x	$[101, 125]$		$\perp$	$[101, 125]$	$[101, 125]$	$[1, 125]$			
	r	2			2	2	2			

Table 1. Abstract states in the analyzer, presented as ranges for conciseness

2 Backward Analysis using Under-Approximation

The various SAT-based algorithms that constitute our backwards analysis are orchestrated by a strategy that tries to find a path to the beginning of the program with minimal effort. Specifically, the idea is to perform iterative deepening when unrolling loops until either a counterexample is found or a proof that the alarm was spurious. We illustrate this strategy using Fig. 1 which shows a program that limits some signed input variable x to $0 \leq x \leq 200$ and then iteratively decreases x by one if the original input was $x < 100$ and by two otherwise. The abstract states $S_0, \ldots S_8$ inferred by a forward analysis (here based on intervals) are stored for each tip of an edge where an edge represents either a guard or some assignments. The resulting states of the forward analysis are listed in Table 1. Since S_8 violates the assertion $x = 0$, we propagate the negated assertion $x \leq -1 \vee x \geq 1$ backwards as an assumption. As the forward analysis constitutes a sound over-approximation of the reachable states, we may intersect the assumption with S_8, leading to the refined assumption S'_8 in Table 1. We follow the flow backwards by applying the guard $x \leq 0$ which has no effect on S'_8.

At this point, we try to continue backwards without entering the loop. This strategy ensures that the simplest counterexample is found first. However, in this case S'_8 conjoined with S_5 yields an empty state, indicating that the chosen path cannot represent a counterexample. The only feasible trace is therefore one that passes through the loops that we have skipped so far. In the example, only one loop exists on the path, and we calculate the effect of this loop having executed $0, \ldots, 2^i$ times, beginning with $i = 0$. At the bit-level, the effect of executing a loop body backwards can be modelled as a Boolean function f which one can compose with itself to express the effect of executing the body twice: $f^2 = f \circ f$.

For the sake of presentation, let $f \vee f^2$ denote the effect of executing the loop once or twice. We then pre-compute f^{2^i} and express the semantics of $0, \ldots, 2^i$ iterations as $\varphi_{i+1} = \varphi_i \vee \varphi_i \circ f^{2^i}$ with φ_0 being defined as the identity. For each $i = 1, \ldots$, we unroll all loops that we have encountered so far until we manage to propagate the resulting state further backwards. For instance, in the example we unroll the loop once by propagating the state S'_8 backwards through the loop, yielding $S'_7 = S'_8$ and $S'_6 = \{x \mapsto r - 1 \in [0,1], r \mapsto [1,2]\}$. Applying the guard $x > 0$ yields a non-empty $S'_5 = \{x \mapsto [1,1], r \mapsto [2,2]\}$. However, $S'_5 \sqcap S_2 = \emptyset$ and $S'_5 \sqcap S_4 = \emptyset$ and hence the loop must be unrolled further. After five more iterations, we find $S''_5 = \varphi_5(S'_5) = \{x \mapsto 2n - 1 \wedge n \in [1,63], r \mapsto 2\}$ which has a non-empty intersection with S_4, leading to $S''_4 = \{x \mapsto 2n - 1 \wedge n \in [51,63], r \mapsto 2\} = S''_3 = S''_0$, thereby providing a counterexample that violates the assertion.

Interestingly, the above construction can also be used to identify a warning as false positive: If during the unrolling of a loop $\varphi_{i+1}(S) \models \varphi_i(S)$ then further unrolling does not add any new states. If propagating this fixpoint beyond a certain point p in the program is impossible (it drops to bottom) then the warning is spurious and the forward analysis lost precision between p and the assertion.

However, calculating φ_i can become very expensive and a fixpoint might be impossible to obtain. The source of the complexity is the elimination of existentially quantified variables that tie the input of a function to the output. For instance, the Boolean formula $(\boldsymbol{o} = f^2(\boldsymbol{i})) \equiv \exists \boldsymbol{t} : (\boldsymbol{o} = f(\boldsymbol{t}) \wedge \boldsymbol{t} = f(\boldsymbol{i}))$ introduces fresh variables $\boldsymbol{t}$ that must be removed in order to avoid exponential growth of the formula when calculating $f^{2n} = f^n \circ f^n$. Further intermediate variables are required in φ_i to express that the result is either $\boldsymbol{o}$ or $\boldsymbol{t}$.

In order to reduce the cost of the calculation, we employ a simple pre-analysis that infers a minimal number of loop iterations 2^m that are required to proceed past the initialization in the loop header. In case $m > 0$, we calculate the formula $f^m \circ \varphi_{i-m}$ that does not consider cases in which the loop exits in the first 2^m iterations and which is cheaper to calculate than φ_i for $i \geq m$. Moreover, rather than examining all φ_i with $i \geq m+1$ at once, we fix $i = 0$ in $f^m \circ \varphi_{i-m}$ for all loops in the program. If no trace can be found, we retry for each i. The two heuristics square with the observation that, usually, an error trace through a loop exists for small k, unless a loop iterates m times where m is constant, which is addressed by composing φ_i with f^m. If calculating $f^n \circ \varphi_{i-m}$ is still too expensive, we apply an algorithm that can under-approximate the elimination of the existentially quantified variables. Once under-approximation is used, traces may be missed and an inferred error cannot be shown to be a false positive. However, any trace found using under-approximation is still a valid counterexample.

In summary, if loops must be unrolled, our approach uses an iterative deepening approach where in each step the number of iterations that are considered is doubled. It also applies a heuristic that unrolls a loop by n iterations if it is clear that the loop cannot exit earlier. These techniques are applied in the next section to eliminate false positives. For complex loops and many iterations, we under-approximate the existential elimination in a well-motivated fashion. This approach is detailed in the context of inferring counterexamples in Sect. 4.

```
1   unsigned int log2(unsigned char c) {
2           unsigned char i = 0;
3           if (c==0) return 0; else c--;
4           while (c > 0) {
5                   i = i + 1;
6                   c = c >> 1;
7           }
8           assert(i <= 7);
9           return i;
10  }
```

Fig. 2. Computation of the log2 of an unsigned integer `c`; even though the code is correct, abstract interpreters based on domains such as convex polyhedra emit a warning

3 Eliminating False Positives

Consider the program in Fig. 2, which computes the logarithm to the base 2 of an unsigned character `c` (a bit-vector of length 8) and stores the result in `i`. Clearly, `i` should hold a value less than 8, which is formulated in terms of an assertion. The assertion is valid, yet most abstract interpreters emit a warning; typical domains fail to capture the relation between `i`, which is used in the assertion, and `c`, which specifies the termination condition. We build towards our technique, which proves the non-existence of a defective path, in three steps.

3.1 Concrete relational semantics in Boolean logic

To mark the warning as spurious, our analysis thus attempts to exclude all paths that lead to a state satisfying the invariant $\iota = (0 \leq \texttt{i} \leq 255 \wedge 0 \leq \texttt{c} \leq 0)$ produced by the forward analysis for line 8, and at the same time violates $0 \leq \texttt{i} \leq 7$. We express the concrete relational semantics of each block in the program in Boolean logic. The values of `i` on entry and exit of each basic block are represented using bit-vectors $\boldsymbol{i}$ and $\boldsymbol{i}'$, respectively. Likewise, use bit-vectors $\boldsymbol{c}$ and $\boldsymbol{c}'$ to represent `c`. In the following, let $\langle \boldsymbol{x} \rangle = \sum_{i=0}^{7} 2^i \cdot \boldsymbol{x}[i]$ denote the unsigned value of a bit-vector $\boldsymbol{x}$, and let $\boldsymbol{x}[j]$ denote the j^{th} bit of $\boldsymbol{x}$. Let the notation $[\![\cdot]\!]$ encode an arithmetic constraint as Boolean formula. Then, $f_I(\boldsymbol{V}, \boldsymbol{V}') := [\![\boldsymbol{i}' = 0, \boldsymbol{c} \neq 0, \boldsymbol{c}' = \boldsymbol{c} - 1]\!]$ encodes the initialization block of the function over inputs $\boldsymbol{V} = \{\boldsymbol{c}, \boldsymbol{i}\}$ and outputs $\boldsymbol{V}' = \{\boldsymbol{c}', \boldsymbol{i}'\}$. In a similar fashion, $f_L(\boldsymbol{V}, \boldsymbol{V}')$ encodes the loop body:

$$f_I(\boldsymbol{V}, \boldsymbol{V}') = \left\{ \bigwedge_{j=0}^{7} \neg \boldsymbol{i}'[j] \wedge \bigvee_{j=0}^{7} \boldsymbol{c}'[j] \wedge ((\bigwedge_{j=0}^{7} \boldsymbol{c}[j] \leftrightarrow (\boldsymbol{c}'[j] \oplus \bigwedge_{k=0}^{j-1} \boldsymbol{c}'[k]) \right.$$

$$f_L(\boldsymbol{V}, \boldsymbol{V}') = \left\{ \begin{array}{l} (\bigvee_{j=0}^{7} \boldsymbol{c}[j]) \wedge \neg \boldsymbol{c}'[7] \wedge (\bigwedge_{j=0}^{6} \boldsymbol{c}'[j] \leftrightarrow \boldsymbol{c}[j+1]) \wedge \\ (\bigwedge_{j=0}^{7} \boldsymbol{i}'[j] \leftrightarrow (\boldsymbol{i}[j] \oplus \bigwedge_{k=0}^{j-1} \boldsymbol{i}[k]) \end{array} \right.$$

In order to find a path to a state that satisfies ι and violates the assertion, encode ι in Boolean logic as $[\![\iota]\!] = \bigwedge_{j=0}^{7} \neg \boldsymbol{c}[j]$. Furthermore, let $[\![0 \leq \langle \boldsymbol{i} \rangle \leq 7]\!] = \bigwedge_{j=4}^{7} \neg \boldsymbol{i}[j]$ encode the assertion. The error state $g(\boldsymbol{V})$ after the loop is given as:

$$
\begin{aligned}
g(\boldsymbol{V}) &= [\![\iota]\!] \wedge \neg[\![0 \leq \langle \boldsymbol{i} \rangle \leq 7]\!] \\
&= [\![(0 \leq \langle \boldsymbol{i} \rangle \leq 255) \wedge (\langle \boldsymbol{c} \rangle = 0)]\!] \wedge [\![8 \leq \langle \boldsymbol{i} \rangle \leq 255]\!] \\
&= [\![(8 \leq \langle \boldsymbol{i} \rangle \leq 255) \wedge (\langle \boldsymbol{c} \rangle = 0)]\!] \\
&= \bigwedge_{j=0}^{7} \neg \boldsymbol{c}[j] \wedge \bigvee_{j=4}^{7} \boldsymbol{i}[j]
\end{aligned}
$$

We commence by testing the shortest trace to the erroneous state $g(\boldsymbol{V})$, i.e., the trace going through the initialization block followed directly by an assertion violation. This path is feasible if $f_I(\boldsymbol{V}, \boldsymbol{V}') \wedge g(\boldsymbol{V}')$ is satisfiable; since the formula is unsatisfiable, this path cannot be part of a counterexample. A valid trace thus traverses the loop $n \geq 1$ times. One way to discover n, and thus a trace to the loop-entry state $\omega = (\langle \boldsymbol{i} \rangle = 0) \wedge (0 \leq \langle \boldsymbol{c} \rangle \leq 255)$, is to iteratively unroll the loop. However, instead of composing n functions f_L (each representing one iteration), we infer an $m \leq n$ using affine abstraction and derive f_L^m in $\log_2(m)$ steps.

3.2 Lower bounds on the number of loop iterations

The first step of computing a lower bound on the number of loop iterations is to abstract $f_L(\boldsymbol{V}, \boldsymbol{V}')$ using a conjunction of affine equalities [16], which relate symbolic bounds $\boldsymbol{V}_{\ell,u} = \{\boldsymbol{c}_\ell, \boldsymbol{c}_u, \boldsymbol{i}_\ell, \boldsymbol{i}_u\}$ on entry of the block to symbolic bounds $\boldsymbol{V}'_{\ell,u} = \{\boldsymbol{c}'_\ell, \boldsymbol{c}'_u, \boldsymbol{i}'_\ell, \boldsymbol{i}'_u\}$ on exit. Here, $\boldsymbol{c}_\ell$, $\boldsymbol{c}_u$, $\boldsymbol{c}'_\ell$, and $\boldsymbol{c}'_u$ are bit-vectors representing the lower and upper bounds of `c`, respectively; similarly for `i`. Applying the abstraction scheme from [5, Sect. 3] yields the following system of affine equations:

$$
\mathcal{F} = \left\{ \begin{array}{llll} \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 2 \cdot (\langle \boldsymbol{c}'_u \rangle + 1) - 1 & \wedge \\ \langle \boldsymbol{i}_\ell \rangle = \langle \boldsymbol{i}'_\ell \rangle - 1 & \wedge & \langle \boldsymbol{i}_u \rangle = \langle \boldsymbol{i}'_u \rangle - 1 & \end{array} \right\}
$$

We transform $g(\boldsymbol{V})$ to express affine constraints on the outputs $\boldsymbol{V}'_{\ell,u}$ by automatically lifting the characterization over program variables to relations over range variables (see [5, Sect. 3.2] for further details of this operation):

$$
g_{\mathsf{aff}}(\boldsymbol{V}') = \left\{ \begin{array}{llll} \langle \boldsymbol{i}'_\ell \rangle = 8 & \wedge & \langle \boldsymbol{i}'_u \rangle = 255 & \wedge \\ \langle \boldsymbol{c}'_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}'_u \rangle = 0 & \end{array} \right\}
$$

Then, applying $\mathcal{F}$ to $g_{\mathsf{aff}}(\boldsymbol{V}')$ yields

$$
\mathcal{F}(g_{\mathsf{aff}}(\boldsymbol{V}')) = \left\{ \begin{array}{llll} \langle \boldsymbol{i}_\ell \rangle = 7 & \wedge & \langle \boldsymbol{i}_u \rangle = 254 & \wedge \\ \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 1) & \end{array} \right\}
$$

which, in turn, gives $(7 \leq \langle \boldsymbol{i} \rangle \leq 254) \wedge 0 \leq \langle \boldsymbol{c} \rangle \leq 1)$. The intersection with the precondition $[\![\omega]\!]$ yields $\bot$; thus a single loop iteration does not suffice. In the next iteration, we summarize two executions of the loop using relational composition $\circ_{\mathsf{Lin}}$ of two affine systems $\mathcal{F}_1$ and $\mathcal{F}_2$. This amounts to renaming the outputs of $\mathcal{F}_1$ and the inputs of $\mathcal{F}_2$ to the same temporary variables, and eliminating these from the conjunction of both systems using projection [19, 27]. The projection, in turn, has a straightforward implementation using Gauss elimination. In the example, two iterations of the loop are characterized by:

$$
\mathcal{F}^2 = \mathcal{F} \circ_{\mathsf{Lin}} \mathcal{F} = \left\{ \begin{array}{llll} \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 4 \cdot (\langle \boldsymbol{c}'_u \rangle + 1) - 1 & \wedge \\ \langle \boldsymbol{i}_\ell \rangle = \langle \boldsymbol{i}'_\ell \rangle - 2 & \wedge & \langle \boldsymbol{i}_u \rangle = \langle \boldsymbol{i}'_u \rangle - 2 & \end{array} \right\}
$$

Again, we get $F^2(g_{\mathsf{aff}}(\boldsymbol{V})) \sqcap [\![\omega]\!] = \bot$. Likewise, compute:

$$F^4 = F^2 \circ_{\mathsf{Lin}} F^2 = \left\{ \begin{array}{llll} \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 16 \cdot (\langle \boldsymbol{c}'_u \rangle + 1) - 1 & \wedge \\ \langle \boldsymbol{i}_\ell \rangle = \langle \boldsymbol{i}'_\ell \rangle - 4 & \wedge & \langle \boldsymbol{i}_u \rangle = \langle \boldsymbol{i}'_u \rangle - 4 & \end{array} \right\}$$

$$F^8 = F^4 \circ_{\mathsf{Lin}} F^4 = \left\{ \begin{array}{llll} \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 256 \cdot (\langle \boldsymbol{c}'_u \rangle + 1) - 1 & \wedge \\ \langle \boldsymbol{i}_\ell \rangle = \langle \boldsymbol{i}'_\ell \rangle - 8 & \wedge & \langle \boldsymbol{i}_u \rangle = \langle \boldsymbol{i}'_u \rangle - 8 & \end{array} \right\}$$

Observe that

$$F^8(g_{\mathsf{aff}}(\boldsymbol{V}')) = \left\{ \begin{array}{llll} \langle \boldsymbol{c}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{c}_u \rangle = 255 & \wedge \\ \langle \boldsymbol{i}_\ell \rangle = 0 & \wedge & \langle \boldsymbol{i}_u \rangle = 247 & \end{array} \right\}$$

describes states that satisfy the invariant $[\![\omega]\!]$ prior to the loop. Thus, the minimum number of loop iterations is $5 \leq m \leq 8$. Using binary search, we determine which F^m is the first to satisfy $[\![\omega]\!]$. This gives $m = 8$. Observe that, due to abstraction, this bound is not necessarily exact (though it is in this example) in that any counterexample trace must traverse the at least loop eight times.

3.3 Summarizing a Number of Iterations

We now face the task of efficiently calculating input-output behavior of eight loop iterations as a Boolean formula f_L^8 which is later used to compute the pre-image of f_L^8 subject to $g(\boldsymbol{V})$. Analogous to the construction of composing affine transformers, we incrementally double the number of iterations summarized in a single formula, thus finessing the need to unroll the loop. Specifically, put:

$$\begin{array}{ll} f_L^0(\boldsymbol{V}, \boldsymbol{V}') = \mathsf{id}(\boldsymbol{V}, \boldsymbol{V}') & f_L^2(\boldsymbol{V}, \boldsymbol{V}') = \exists \boldsymbol{V}'' : f^1(\boldsymbol{V}, \boldsymbol{V}'') \wedge f^1(\boldsymbol{V}'', \boldsymbol{V}') \\ f_L^1(\boldsymbol{V}, \boldsymbol{V}') = f_L(\boldsymbol{V}, \boldsymbol{V}') & f_L^4(\boldsymbol{V}, \boldsymbol{V}') = \exists \boldsymbol{V}'' : f^2(\boldsymbol{V}, \boldsymbol{V}'') \wedge f^2(\boldsymbol{V}'', \boldsymbol{V}') \end{array}$$

Each f_L^i describes an input-output relation for exactly i applications of f_L. To eliminate $\boldsymbol{V}''$ from the formulae, we apply a SAT-based projection algorithm [6]. This construction suffices to test $[\![\omega]\!] \wedge f_L^8(\boldsymbol{V}, \boldsymbol{V}') \wedge g(\boldsymbol{V}')$ for satisfiability, i.e., to check if the erroneous state $g(\boldsymbol{V}')$ can be reached with exactly eight iterations starting in $[\![\omega]\!]$. If unsatisfiable, it is necessary to unroll the loop further. Again, we construct a summary φ_i of $m + 2^i$ iterations, where m is the lower bound on the number of iterations. Then, φ_i describes all states reachable after $m, \ldots, m + 2^i$ iterations whereas $f_L^{m+2^i}$ describes exactly $m + 2^i$ iterations.

3.4 Summarizing a Range of Iterations

Formally, let $\varphi_i(\boldsymbol{V}) = \exists \boldsymbol{V}' : \exists \boldsymbol{V}'' : (\bigvee_{j=0}^{2^i} f_L^j(\boldsymbol{V}, \boldsymbol{V}')) \wedge f_L^m(\boldsymbol{V}', \boldsymbol{V}'') \wedge g(\boldsymbol{V}'')$, that is, erroneous states expressed over $\boldsymbol{V}''$ being backpropagated m times around the loop giving $\boldsymbol{V}'$, which are, in turn, $j = 0, \ldots, 2^i$ times transformed into constraints over $\boldsymbol{V}$. Rather than recalculating each $\varphi_i(\boldsymbol{V})$ from scratch, we compute $\varphi_i(\boldsymbol{V})$ based on the following inductive definition, allowing us to reuse $\varphi_{i-1}(\boldsymbol{V})$ to compute $\varphi_i(\boldsymbol{V})$ and requiring only i instead of 2^i steps:

$$\varphi_i(\boldsymbol{V}) = \left\{ \begin{array}{ll} \exists \boldsymbol{V}' : f_L^m(\boldsymbol{V}, \boldsymbol{V}') \wedge g(\boldsymbol{V}') & : i = 0 \\ \varphi_{i-1}(\boldsymbol{V}) \vee (\exists \boldsymbol{V}' : f^{2^i}(\boldsymbol{V}, \boldsymbol{V}') \wedge \varphi_{i-1}(\boldsymbol{V}')) & : \text{otherwise} \end{array} \right.$$

```
1   unsigned int hamDist(int x, int y) {
2           unsigned int d = 0;
3           unsigned int v = x ^ y;
4           while (v != 0) {
5                   d = d + 1;
6                   v = v & (v - 1);
7           }
8           assert(d < 32);
9           return d;
10  }
```

Fig. 3. Erroneous hamming distance calculation; the assertion in line 8 does not hold

Note that, due to monotonicity, there exists an $i \geq 0$ with $\varphi_i(\boldsymbol{V}) \models \varphi_{i-1}(\boldsymbol{V})$. In the example, since $\varphi_4(\boldsymbol{V}) \models \varphi_3(\boldsymbol{V})$ and $\varphi_3(\boldsymbol{V}) \wedge [\![\omega]\!]$ is unsatisfiable, we deduce that no trace from $[\![\omega]\!]$ to the erroneous state $g(\boldsymbol{V})$ exists that iterates more than eight times. Hence, the warning emitted by the forward analysis is spurious. In certain cases, calculating φ_i can become too costly, which is addressed next.

4 Finding Counterexamples

Although the iterative deepening heuristic reduces the complexity of the generated formulae, exact state spaces cannot always be computed since calculating $\exists \boldsymbol{V'} : f^{2^i}(\boldsymbol{V}, \boldsymbol{V'}) \wedge \varphi_{i-1}(\boldsymbol{V'})$ may result in an exponentially sized formula. However, if the aim is to only find a counterexample rather than eliminating false positives, an under-approximation of the projection $\exists \boldsymbol{V'} : \psi$ suffices. In order to illustrate the idea, consider Fig. 3 which presents a function to calculate the Hamming distance of two integers `x` and `y`. Once more, we bit-blast the concrete semantics of both, loop body and loop pre-condition. Here, $\oplus$ denotes the Boolean exclusive-or and $\boldsymbol{u}$ is an auxiliary bit-vector that captures the intermediate value of `v-1`:

$$f_I(\boldsymbol{V}, \boldsymbol{V'}) = \left\{ \bigwedge_{j=0}^{31} \neg \boldsymbol{d'}[j] \wedge \bigwedge_{j=0}^{31} \boldsymbol{v'}[i] \leftrightarrow \boldsymbol{x}[j] \oplus \boldsymbol{y}[j] \right.$$

$$f_L(\boldsymbol{V}, \boldsymbol{V'}) = \left\{ \begin{array}{l} (\bigwedge_{j=0}^{31} \boldsymbol{d'}[j] \leftrightarrow (\boldsymbol{d}[j] \oplus \bigwedge_{k=0}^{j-1} \boldsymbol{d}[k])) \wedge (\bigvee_{j=0}^{31} \boldsymbol{v}[j]) \wedge \\ (\bigwedge_{j=0}^{31} \boldsymbol{v}[j] \leftrightarrow (\boldsymbol{u}[j] \oplus \bigwedge_{k=0}^{j-1} \boldsymbol{u}[k])) \wedge (\bigwedge_{j=0}^{31} \boldsymbol{v'}[j] \leftrightarrow (\boldsymbol{v}[j] \wedge \boldsymbol{u}[j])) \end{array} \right.$$

As before, let $\iota = (\langle \boldsymbol{v} \rangle = 0 \wedge \langle \boldsymbol{d} \rangle = \top)$ describe the invariant derived at the assertion which was inferred during the forward analysis and let $\omega = (\langle \boldsymbol{d} \rangle = 0 \wedge \langle \boldsymbol{v} \rangle = \langle \boldsymbol{x} \rangle \oplus \langle \boldsymbol{y} \rangle)$ represent the state at loop entry. The erroneous state after the loop is thus characterized as $g(\boldsymbol{V}) = [\![\iota]\!] \wedge \bigvee_{j=5}^{31} \boldsymbol{d}[j]$ in Boolean logic.

4.1 Lower Bounds on the Number of Loop Iterations

Again, to compute a lower bound on the number of loop iterations required to reach the erroneous state $g(\boldsymbol{V})$ from the pre-condition $\boldsymbol{V'}$ defined by $f_I(\boldsymbol{V}, \boldsymbol{V'})$, we derive an abstraction of the loop transfer function $f_L(\boldsymbol{V}, \boldsymbol{V}')$ in terms of a conjunction of affine equalities. This operation gives:

$$\mathcal{F} = \left\{ \langle \boldsymbol{d}_\ell \rangle = \langle \boldsymbol{d}'_\ell \rangle - 1 \quad \wedge \quad \langle \boldsymbol{d}_u \rangle = \langle \boldsymbol{d}'_u \rangle - 1 \right\}$$

Note that `v = v & (v - 1)` is non-affine, hence the lack of an affine constraint over `v`. We transform $g(\boldsymbol{V})$ to express affine constraints on the outputs as per [5]:

$$g_{\mathsf{aff}}(\boldsymbol{V}') = \left\{ \langle \boldsymbol{d}'_\ell \rangle = 32 \quad \wedge \quad \langle \boldsymbol{d}'_u \rangle = 2^{32} - 1 \quad \right\}$$

Applying $\mathcal{F}$ to $g_{\mathsf{aff}}(\boldsymbol{V}')$ yields $\mathcal{F}(g_{\mathsf{aff}}(\boldsymbol{V}')) = \left\{ \langle \boldsymbol{d}_\ell \rangle = 31 \wedge \langle \boldsymbol{d}_u \rangle = 2^{32} - 2 \right\}$ which, in turn, gives $31 \leq \langle \boldsymbol{d} \rangle \leq 2^{32} - 2$. The intersection with the state $\omega = (\langle \boldsymbol{d} \rangle = 0)$ that the forward analysis inferred for the loop entry yields $[\![31 \leq \langle \boldsymbol{d} \rangle \leq 2^{32} - 2]\!] \sqcap [\![\omega]\!] = \bot$; thus a single loop iteration does not suffice. Following the strategy discussed in Sect. 3.2, we compute $\mathcal{F}^2(g_{\mathsf{aff}}(\boldsymbol{V}'))$, $\mathcal{F}^4(g_{\mathsf{aff}}(\boldsymbol{V}'))$, $\mathcal{F}^8(g_{\mathsf{aff}}(\boldsymbol{V}'))$, $\mathcal{F}^{16}(g_{\mathsf{aff}}(\boldsymbol{V}'))$, and $\mathcal{F}^{32}(g_{\mathsf{aff}}(\boldsymbol{V}'))$. It is only $\mathcal{F}^{32}(g_{\mathsf{aff}}(\boldsymbol{V}'))$ that satisfies $\mathcal{F}^{32}(g_{\mathsf{aff}}(\boldsymbol{V}')) \sqcap [\![\omega]\!] \neq \bot$. Consequently, the minimum number of loop iterations is $17 \leq m \leq 32$. Using binary search, we determine which $\mathcal{F}^m$ is the first to satisfy $[\![\omega]\!]$. This gives $m = 32$ as the minimum number of iterations.

4.2 Under-Approximating a Range of Iterations

As in Sect. 3.3, we face the task of summarizing the execution of 32 consecutive loop iterations. To find a backward trace, compute $f_L^0(\boldsymbol{V}, \boldsymbol{V}')$ and $f_L^1(\boldsymbol{V}, \boldsymbol{V}')$ as before. Rather than computing $f_L^i(\boldsymbol{V}, \boldsymbol{V}')$ exactly by enumerating all of the projection space, we preempt the computation of $f_L^2(\boldsymbol{V}, \boldsymbol{V}') = \exists \boldsymbol{V}'' : f^1(\boldsymbol{V}, \boldsymbol{V}'') \wedge f^1(\boldsymbol{V}'', \boldsymbol{V}')$ prematurely after, say, 100 models have been enumerated (though in our implementation, we have used a heuristic based on the structure of the erroneous goal state $g(\boldsymbol{V})$ rather than one that is based solely on the number of models, see Sect. 5). This tactic yields a formula $h_L^2(\boldsymbol{V}, \boldsymbol{V}')$ in CNF that entails $f_L^2(\boldsymbol{V}, \boldsymbol{V}')$. In other words, every of model $h_L^2(\boldsymbol{V}, \boldsymbol{V}')$ is also a model of $f_L^2(\boldsymbol{V}, \boldsymbol{V}')$, i.e., the formula is easier to compute and under-approximates $f_L^2(\boldsymbol{V}, \boldsymbol{V}')$. Based on $h_L^2(\boldsymbol{V}, \boldsymbol{V}')$, we compute $h^4(\boldsymbol{V}, \boldsymbol{V}'') = \exists \boldsymbol{V}'' : h_L^2(\boldsymbol{V}, \boldsymbol{V}'') \wedge h_L^2(\boldsymbol{V}'', \boldsymbol{V}')$. Likewise compute $h_L^8(\boldsymbol{V}, \boldsymbol{V}')$, $h_L^{16}(\boldsymbol{V}, \boldsymbol{V}')$ and $h_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$. This under-approximating strategy may decrease the size of the formulae exponentially.

4.3 Failing to Derive a Counterexample Trace

Suppose now that the summary of states $\varphi_0(\boldsymbol{V}) = \exists \boldsymbol{V}' : h_L^{32}(\boldsymbol{V}, \boldsymbol{V}') \wedge g(\boldsymbol{V}')$ yields a state description such that $\varphi_0(\boldsymbol{V}) \wedge [\![\omega]\!]$ is unsatisfiable and that $\varphi_i(\boldsymbol{V}) \models \varphi_0(\boldsymbol{V})$ for any $i \geq 1$. Then the under-approximated transfer function $h_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$ is insufficient to reach a loop-entry state. Hence, it is necessary to compute a greater under-approximation $\hat{h}_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$ such that $h_L^{32}(\boldsymbol{V}, \boldsymbol{V}') \models \hat{h}_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$. Doing so necessitates computing $\hat{h}_L^2(\boldsymbol{V}, \boldsymbol{V}')$ such that $h_L^2(\boldsymbol{V}, \boldsymbol{V}') \models \hat{h}_L^2(\boldsymbol{V}, \boldsymbol{V}')$; likewise for $\hat{h}_L^4(\boldsymbol{V}, \boldsymbol{V}')$, $\hat{h}_L^8(\boldsymbol{V}, \boldsymbol{V}')$, $\hat{h}_L^{16}(\boldsymbol{V}, \boldsymbol{V}')$, and $\hat{h}_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$. Based on the enlarged under-approximation $\hat{h}_L^{32}(\boldsymbol{V}, \boldsymbol{V}')$, we compute $\hat{\varphi}_0(\boldsymbol{V}) = \exists \boldsymbol{V}' : \hat{h}_L^{32}(\boldsymbol{V}, \boldsymbol{V}') \wedge g(\boldsymbol{V}')$, which by monotonicity satisfies $\varphi_0(\boldsymbol{V}) \models \hat{\varphi}_0(\boldsymbol{V})$. Suppose that $\hat{\varphi}_0(\boldsymbol{V}) \wedge [\![\omega]\!]$ is satisfiable, producing a model $\mathbf{m} \models \hat{\varphi}_0(\boldsymbol{V}) \wedge [\![\omega]\!]$ defined as follows:

$$\mathbf{m} = \left\{ \begin{array}{l} \boldsymbol{d}[0] \mapsto 0,\, \boldsymbol{d}[1] \mapsto 0,\, \boldsymbol{d}[2] \mapsto 0,\, \boldsymbol{d}[3] \mapsto 0,\, \ldots,\, \boldsymbol{d}[31] \mapsto 0 \\ \boldsymbol{x}[0] \mapsto 0,\, \boldsymbol{x}[1] \mapsto 1,\, \boldsymbol{x}[2] \mapsto 0,\, \boldsymbol{x}[3] \mapsto 1,\, \ldots,\, \boldsymbol{x}[31] \mapsto 1 \\ \boldsymbol{y}[0] \mapsto 1,\, \boldsymbol{y}[1] \mapsto 0,\, \boldsymbol{y}[2] \mapsto 1,\, \boldsymbol{y}[3] \mapsto 0,\, \ldots,\, \boldsymbol{y}[31] \mapsto 0 \end{array} \right\}$$

This model entails that we successfully applied an under-approximate loop transformer to find a trace that executes 32 iterations. Bit-vectors $\boldsymbol{x} = \langle 0101\ldots01\rangle$ and $\boldsymbol{y} = \langle 1010\ldots10\rangle$ then indicate values that give a Hamming distance of 32 and therefore violate the assertion. We have thus computed a definite counterexample.

5 Experiments

We have integrated the techniques described in this paper into the [MC]SQUARE framework, which is written in JAVA. Several programs have been analyzed that contain at least one loop each. The benchmarks shown in Tab. 2 include Wegner's bit-counting `bit-cnt`, the algorithm in Fig. 3 `ham-dist`, consecutive loops that shift and add `inc-lshift`, the algorithm in Fig. 2 `log`, parity calculation `parity`, `parity_mit`, bit-reversal `randerson`, swapping of bytes `swap` and two interdependent, nested `loops`. The running times were obtained on a 2.4 GHz MACBOOK PRO equipped with 4 GB of RAM. The programs are written in Instruction List, a language used in Programmable Logic Controllers. The semantics of these programs are translated into bit-vector relations, similarly to the examples in Sect. 3 and Sect. 4. This translation and the calculation of the affine loop transformers is written in JAVA using SAT4J. In none of the benchmark do these calculations take more than 0.1s of the runtimes. The Boolean summarization of loop iterations and the counterexample generation are implemented in C++ using MINISAT and CUDD. MINISAT frequently outperforms SAT4J by a factor of 5-10 and was thus chosen for the more demanding transfer function synthesis.

Table 2 presents the timings for different analysis strategies. In the simplest strategy, the post-condition state that violates the assertion g is propagated through φ_n, the fixpoint of the input/output behavior of a loop. The times to calculate the loop transfer function φ_n is given in column "Runtime (Full) / TF" whereas propagating the state g through φ_n is given in column "Runtime (Full) / CE". Note that these two phases are interleaved and that the table presents the accumulated times spent in each phase. The next sections discuss the impact of pre-computing a minimal number of unrollings of a loop using affine abstractions and of restricting the post-condition g to find a counterexample quicker.

5.1 Affine Estimation of Iterations

Inferring a lower bound on the number of loop iterations follows the algorithm presented in [4, Sect. 3.2]. The affine relationships on the bounds of the variables are inferred by asking for an initial solution to the loop transfer function f, yielding an assignment for the input and output variables. These assignments form a linear equation system. By using cheap incremental SAT solving, different assignments are queried and joined into the equation system by calculating the affine hull which, in turn, reduces to Gauss elimination. This process will terminate after at most $n+1$ queries to the SAT solver for n input/output variables. Each query is rather trivial by current standards. If an affine relationship exists, a minimal number of loop iterations can be calculated by composing the affine transfer function repeatedly with itself using the $\circ_{\mathsf{Lin}}$ operation which, again, reduces to cheap Gauss elimination. Indeed, these steps contribute less than 0.1s for each benchmark and we therefore omitted this phase from the table. All our examples contain at least one variable that increases with each loop iteration such that the estimated minimal number of iterations is exactly the number of iterations it takes to exit the loop. This minimum number of iterations n allows us to reduce the size of the formula by those conjuncts that model that the loop may be exited after $i < n$ unrollings, thereby alleviating the SAT solver from proving this fact at the binary level. The speedup due to unrolling is minor (and thus omitted from the table). Still, it shows that proving the exit condition in MiniSat is more costly than Gauss elimination in Java and querying Sat4J.

5.2 Focussing the Search for Counterexamples

According to Table 2, the dominant part of the backwards analysis is the phase of calculating and composing loop transformers, which hinges on the performance of projection. In our experiments, we used model enumeration [6] and combined it with BDDs so as to derive a quantifier-free CNF formula [22]. Cudd v2.4.2 was used since it offers direct support for enumerating a compact CNF formula. Although this combination of data structures for representing Boolean formulae during projection is the best we could find, there naturally exist problems that result in large (intermediate) formulae. Indeed, McMillan [26], amongst others, has observed that no Boolean structure (such as BDDs or CNF) exists which is suitably small for all kinds of inputs; indeed, some problems exist where the CNF is exponential in the size of the respective BDD, and vice versa. However, the projection algorithm of [6] enumerates *prime implicants*, i.e., a Boolean formula that contains a minimum number of literals, thereby covering as many models as possible. This observation is relevant for the task of inferring counterexamples (rather than eliminating false positives where the state space has to be enumerated exhaustively): The algorithm enumerates prime implicants, which entailed that stopping the projection early means that a maximum number of models of the Boolean formula is propagated backwards. In principle, this means that the largest number of states, which also has the simplest representation, is tried first when resorting to under-approximation. Unfortunately, not every model of the formula has the same probability to constitute a counterexample trace. Consider the erroneous target $g(\boldsymbol{V}) = [\![\iota]\!] \wedge \bigvee_{j=5}^{31} \boldsymbol{d}[j]$ from Sect. 4. The prime implicant that captures the maximum number of states is $\boldsymbol{d}[31]$, i.e., the formula stating that the most significant bit of d is set. This choice is in contrast to the intuition that many errors are off-by-one errors and thus happened close to those numeric values $d \in [0, 31]$ that do not violate the assertion.

Benchmark	# Instr.	Runtime (Full)		Runtime (Simp.)	
		TF	CE	TF	CE
`bit-cnt`	26	4.1s	0.9s	0.4s	0.4s
`ham-dist`	19	4.8s	1.7s	0.8s	0.3s
`inc-lshift`	14	3.2s	2.7s	0.8s	0.6s
`log`	22	1.9s	1.3s	0.3s	0.3s
`parity`	28	8.3s	1.2s	1.2s	0.4s
`parity_mit`	17	6.2s	2.6s	1.5s	1.2s
`randerson`	23	8.0s	2.4s	4.2s	0.6s
`swap`	15	5.9s	1.8s	0.9s	0.5s
`loops`	207	43.6s	8.0s	13.1s	5.8s

Table 2. Experimental results for PLC benchmarks

Hence, we employ a heuristic that constrains $g(\boldsymbol{V})$ so that a sub-range of target values are considered that lie close to the feasible state, extending the sub-range to the next power of two iff the given under-approximation is insufficient for finding a counterexample. This is a straightforward extension considering the bit-level encodings of integer values. For the example in Sect. 4, this strategy is applied as follows: The goal-state requires $2^5 \leq \langle \boldsymbol{d} \rangle \leq 2^{32}-1$. In the first iteration, our strategy tries to find values that satisfy $2^5 \leq \langle \boldsymbol{d} \rangle \leq 2^6$. If no counterexample is found, we proceed with $2^5 \leq \langle \boldsymbol{d} \rangle \leq 2^7$, and so forth. This focusses model enumeration to regions that are more likely to contain an actual trace. These simpler models also reduce the runtime of computing projection. The difference is shown in the columns "Runtime (Simp.)", showing significant speed-ups to find counterexamples compared to the "Full" column where g is used without restrictions. Depending on the problems, counterexamples can be found up to 10 times faster by searching near states that do not violate the assertion.

5.3 Discussion

Using Boolean functions to represent a program state has obvious limits. However, when trading the ability to remove false positives for the aspiration of finding backwards traces, under-approximation can yield useful results, even on complex loops. Interestingly, each prime implicant and each sub-range can be tested for feasibility in parallel, which squares with the advent of multi-core processors and may allow the search for counterexamples on larger computer clusters.

6 Related Work

A sound static analysis, usually expressed using the abstract interpretation framework [10], is bound to calculate an over-approximate result to elude undecidability. Due to over-approximation, a safety property may not be verifiable even though it holds. In this case, the emitted warning is a so-called false positive [3] which cannot a priori be distinguished from an actual defect in the software. While an analysis with zero false positives is possible [11], it is crucial to understand the origin of each alarm in order to either refine the analysis or to fix the defect. Thus, analyzing warnings which are emitted poses two related questions: firstly, *is the warning legitimate?*, and if so, *how can the error state be reached in terms of a concrete execution?* The difficulty of answering the first has led to approaches that rank warnings based on the likelihood of being actual defects. Statistical classifications have been based on error correlation [20] or bayesian filtering [15]. Recent work [23] clusters defects, allowing to eliminate dependent defects if a master defect is shown to be spurious (defects can be proven legitimate, too).

An exact answer to both questions is required in counterexample-guided abstraction refinement (CEGAR) in model checking [8]. However, deciding if a warning is legitimate is strictly easier in the context of CEGAR than in a general static analysis as the model checker produces an abstract counterexample. A concrete counterexample may then be inferable by replaying the trace in the concrete program [21]. If successful, the concrete trace can be used afterwards for, e.g., error localization [2]. If constructing the trace fails at a certain program point, a new predicate can be introduced to refine the abstract model [1]. In the context of numeric analysis, Gulavani and Rajamani [14] propose to refine a pre-analysis, based on a fixed point computation with widening, by introducing predicates using so-called hints. Later, they extended their technique to combine widening with interpolants between verification conditions and the inferred state [13]. Yet, neither work is concerned with computing the backward trace but assumes that it has been inferred by a theorem prover. Our approach can infer a set of traces that could provide additional hints as to what new predicates are needed, thus extending their work [13, 14]. Finitization, as performed by our techniques, also appears in bounded model checking [7]. There, the state space of the system is explored in a breadth-first fashion in forward direction, up to a given depth k. By way of comparison, our approach unrolls the program back-to-front, implementing strategies to minimize the unrolling depth k during the generation of a counterexample on an under-approximate description of each block.

For static analyses that operate on the semantics of the actual program, no model program exists in which the trace can be inferred, and backward reasoning from the warning to the program entry is required [28]. Backward reasoning, in turn, amounts to solving the following abduction problem: Given B and C, compute a non-empty A in $(A \wedge B) \Rightarrow C$. Here, A and C can be thought of as states before and after a guard B, respectively. When A, B and C are elements of an abstract domain then $A \neq \bot$ is called the pseudo-complement of B relative to C if it is the largest unique element with $(A \sqcap B) \models C$. A domain in which each pair of elements has a pseudo-complement is called a Heyting domain. Few classes of linear constraints allow abduction [24] and no single numeric domain commonly used in forward analyses is Heyting, nor is the combination of Heyting domains necessarily a Heyting domain [25]. As an example, consider the intervals $B = [0, 0]$, $C = [-5, 5]$ for which two incomparable A can be found, namely $A = [-5, -1]$ and $A = [1, 5]$. One way out of this dilemma is to lift a non-Heyting domain to its power-set domain [18], which yields a Boolean domain. A Boolean domain $\mathcal{B}$ is always Heyting since for each $b \in \mathcal{B}$ there exists a "full" complement $\bar{b} \in \mathcal{B}$ with $b \sqcup \bar{b} = \top$ and $b \sqcap \bar{b} = \bot$. Boolean functions naturally form a Boolean domain which motivates our choice for inferring backward traces. Given their expressiveness and the recent advances in SAT solving, it is sufficient to only use Boolean functions which also forestalls potential difficulties of combining this domain with other (Heyting) domains. Rival [28] sidesteps the abduction problem by calculating an A' with $A \models A'$ using the same domains as in forward analysis. To cap the over-approximation of the backward transformer, backward states are intersected with the forward invariants. Over-approximation makes it unlikely that an empty state is ever observed. Then, a warning cannot be identified as a false positive. Indeed, Rival's analysis merely informs a tool-users about inputs in which a counterexample might lie. In contrast, Erez [12] aims at reducing the number of false positives by performing a bounded search for backward traces using theorem proving. Further afield is the work of Kim et al. [17] who, after a fast but imprecise forward analysis, slice the program for a property violation before running a more expensive forward analysis based on SMT solving.

7 Conclusion

This paper advocates integrating under-approximate abduction using SAT into forward abstract interpretation frameworks. The motivation is to generate a definitive counterexample once a property violation has been detected or to identify a warning as spurious. Using Boolean formulae as abstract domain is theoretically motivated as many domains used in verification cannot express abduction. Moreover, the domain benefits from the progress in SAT solving, specifically the recent advances in computing under-approximate projections.

Acknowledgements The authors want to thank Andy King for interesting discussions and Stefan Kowalewski for his support in this line of scientific enquiry.

References

1. T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV*, volume 3114 of *LNCS*, pages 457–461. Springer, 2004.
2. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, pages 97–105. ACM, 2003.
3. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.
4. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.
5. J. Brauer and A. King. Transfer function synthesis without quantifier elimination. In *ESOP*, volume 6602 of *LNCS*, pages 97–115. Springer, 2011.
6. J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, LNCS. Springer, July 2011.

7. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, volume 1855 of *LNCS*. Springer, 2000.
9. E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 208–224. Springer, 2003.
10. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM, 1977.
11. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The Astrée analyser. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
12. G. Erez. Generating concrete counterexamples for sound abstract interpretation. Master's thesis, School of Computer Science, Tel-Aviv University, Israel, 2004.
13. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.
14. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, volume 3920 of *LNCS*, pages 474–488. Springer, 2006.
15. Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *SAS*, volume 3672 of *LNCS*, pages 203–217. Springer, 2005.
16. M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
17. Y. Kim, J. Lee, H. Han, and K.-M. Choe. Filtering false alarms of buffer overflow analysis using SMT solvers. *Inform. & Softw. Techn.*, 52(2):210–219, 2010.
18. A. King and L. Lu. Forward versus Backward Verification of Logic Programs. In *ICLP*, volume 2916 of *LNCS*, pages 315–330. Springer, 2003.
19. A. King and H. Søndergaard. Inferring Congruence Equations Using SAT. In *CAV*, volume 5123 of *LNCS*, pages 281–293. Springer, 2008.
20. T. Kremenek and D. R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *SAS*, volume 2694 of *LNCS*, pages 295–315. Springer, 2003.
21. D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via program execution. In *ICFEM*, volume 3308 of *LNCS*, pages 224–238. Springer, 2004.
22. S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In *CAV*, volume 2725 of *LNCS*, pages 141–153. Springer, 2003.
23. W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI*, 2012. to appear.
24. M. J. Maher. Abduction of linear arithmetic constraints. In *ICLP*, volume 3668 of *LNCS*, pages 174–188. Springer, 2005.
25. M. J. Maher and G. Huang. On computing constraint abduction answers. In *LPAR*, volume 5330 of *LNCS*, pages 421–435. Springer, 2008.
26. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.
27. M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.
28. X. Rival. Understanding the Origin of Alarms in Astrée. In *SAS*, volume 3672 of *LNCS*, pages 303–319. Springer, 2005.