Property Directed Invariant Refinement for Program Verification

Tobias Welp[1] Andreas Kuehlmann[1,2]

[1] University of California at Berkeley, CA, USA
[2] Coverity, Inc., San Francisco, CA, USA

Abstract

We present a novel, sound, and complete algorithm for deciding safety properties in programs with static memory allocation. The new algorithm extends the program verification paradigm using loop invariants presented in [1] with a counterexample guided abstraction refinement (CEGAR) loop [2] where the refinement is achieved by strengthening loop invariants using the QF_BV generalization of Property Directed Reachability (PDR) discussed in [3, 4]. We compare the algorithm with other approaches to program verification and report experimental results.

1 Introduction

Embedded software systems are ubiquitous in today's life. Practically every electric device sold nowadays contains microprocessors. This inclusion of embedded software systems has come with enormous savings in development and production costs and allowed for many more features previously infeasible to provide. However, the development also increased the risk associated with software failures. In particular in safety-critical applications, bugs can have grave, possibly fatal consequences. This understanding and the insight that testing is not sufficient to guarantee absence of bugs has fueled renewed interest in software verification.

In general, software verification is undecidable. Strategies to cope with this challenge include restricting the problem such that it becomes decidable, devising semi-algorithms without termination guarantee, or a mixture thereof.

As the main contribution of this paper, we present a novel, sound, and complete algorithm for intraprocedural verification of safety properties in programs with static memory allocation. This restriction is relevant as verification problems of this kind are frequent in embedded systems where safety is often critical.

The presented algorithm is based on PDR [5, 6], an efficient algorithm for solving model checking problems (MCP). We leverage PDR in two ways: First, we use a QF_BV generalization of PDR as backend model checker. Second, the design of the presented algorithm incorporates fundamental principles which motivated the design of PDR itself [7]. Most importantly, the presented algorithm is also property directed and attempts to decide the overall verification instance by proving a large number of small lemmata.

The remainder of this paper is structured as follows: In the following section, we discuss previous work we leveraged. Next, in Section 3, we present the proposed program verification algorithm followed by a discussion on how it relates to other verification algorithms. In Section 5, we describe details of our implementation of the presented algorithm and report experimental results. Finally, we draw conclusions and describe future work in Section 6.

2 Applied Previous Work

The presented verification algorithm leverages various ideas from previously published work. In the following subsections, we discuss the main ideas relevant to this work and give references to the respective publications.

2.1 Property Directed Reachability

PDR was originally introduced in [5] as an algorithm for solving hardware MCPs. Given an MCP composed of a set of initial states I, a transition relation T, and a set of bad states B, the objective of PDR is to check whether or not a bad state is reachable from an initial state using only valid transitions. In case a bad state is reachable, PDR returns a counterexample (cex) sequence from an initial state to a bad state. Otherwise, PDR returns a formula that encodes a set of Boolean states that is implied by the initial states, from which no states outside this set can be reached, and which is disjoint from the bad set, in other words, an inductive invariant proving that the bad set is not reachable.

PDR is currently considered the most efficient algorithm for hardware verification [6]. The convincing performance of PDR can be attributed to several of its design principles such as divide-and-conquer (the algorithm attempts to construct the overall proof that bad states are not reachable by solving a number of small lemmata), property directedness (any lemma to be proved has a clear objective towards constructing the overall proof), and the ability to take advantage of parallel computing.

In our program verification framework, we use the generalization of the Boolean PDR algorithm to the theory QF_BV presented in [3] with the extension [4] to refine the model of the program under verification (PUV).

2.2 Program Verification with Loop Invariants

One paradigm to program verification is to transform the control flow graph (CFG) into an SMT formula that is unsatisfiable only if the program satisfies all specified properties. Conceptionally, the biggest challenge associated with this approach are loops that do not allow an immediate transformation into an SMT formula.

A popular strategy to cope with loops has been proposed in [1] where all backedges are cut and loop variables are havoc'ed (i.e. allowed to take arbitrary values). The resulting model overapproximates the program behavior substantially and properties of interest often cannot be proved successfully. To alleviate this constriction, one attempts to capture the semantics of the loop using loop invariants and applies them after the havoc'ing which in turn restricts the range of values a variable can take after the loop. Given strong loop invariants, the given approach has proven to be practically useful. Unfortunately, loop invariants are often not available and despite

978-3-9815370-2-4/DATE14/©2014 EDAA

arguments from the academic community in favor of program annotation (see e.g. [8]), developers are usually not willing or able to provide loop invariants to the verification system. This instigated interest for automatic inference of loop invariants.

Previous approaches to automatic inference of loop invariants include abstract interpretation [9] as well as static [10] and dynamic [11] template solutions. In our work, we infer loop invariants using PDR. In addition to leveraging the convincing practical performance of PDR, the appeal of this approach rests in another important advantage in contrast to the previous approaches: It is property directed. Whereas the other approaches attempt to find generically strong loop invariants, PDR infers loop invariants directed towards the aim to prove the verification condition.

2.3 Counterexample Guided Abstraction Refinement

In our approach to program verification, we use a model of the PUV which overapproximates the behavior of the actual program. If we can prove all properties of interest with this model, we can conclude that the real program has the properties as well. Otherwise, we obtain a cex that we use to refine the overapproximation of the PUV. This CEGAR paradigm was first introduced in the context of functional hardware verification [2].

3 Program Verification using PDR

In the following, we assume that we are given a program in which safety properties of interest are encoded as assertions in the source code. We denote a program as safe if none of its assertions can be violated, regardless of the program input. Otherwise, we denote the program as not safe. Given an input program, the problem of interest is to decide whether or not the PUV is safe.

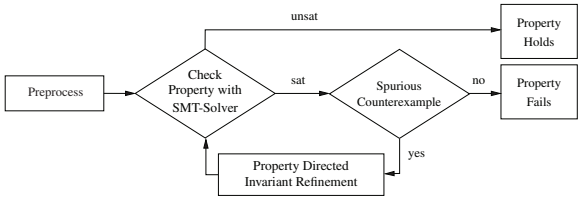


Figure 1: 10,000 Foot-View on Proposed Framework.

Figure 1 depicts a high-level view of the proposed algorithm. In a preprocessing stage, the program is transformed into a model that overapproximates the behavior of the PUV. Next, the model is iteratively refined in a CEGAR loop until we prove or disprove that the PUV safe.

In the following subsections, we discuss details for the individual parts of the framework. We begin with discussing the preprocessing in the next subsection, followed by a discussion of the CEGAR loop in Subsection 3.2, and finally present the loop refinement using PDR in Subsection 3.3. We illustrate each part using a concrete example.

3.1 Preprocessing

We assume that the PUV is represented as a control flow graph (CFG) with basic blocks as nodes containing assumptions, assignments, and assertions and directed edges that allow jumps between basic blocks. A basic block B_1 dominates another basic block B_2 if B_1 is on all paths from the entry basic block to B_2. For simplicity of exposition but irrelevant for the correctness of the algorithm, we also assume the PUV to be structured. Then, every loop l in the CFG can be uniquely identified by a pair (H,T) of basic blocks, where H is the loop head and T is the loop tail. A basic block T is called a loop tail if it is dominated by the loop head H and there is a back edge (T,H) in the CFG. For more detailed information on this notation, please refer to [12].

The loop body of a loop $l = (H,T)$ (denoted as body_l) is the set composed of all basic blocks that are on any path between H and T. We associate with l two sets of variables U_l and R_l where U_l is the set of variables which are updated in loop_l and R_l is the set of variables which are referenced in loop_l. Note that $U_l \subseteq R_l$.

Algorithm 1 `preprocess()`

1: **for** each loop $l = (H,T)$ in PUV **do**
2: remove back edge (T,H)
3: append $\text{LI}_l(U_l', R_l)$ to H; set $\text{LI}_l(U_l', R_l) = \mathit{true}$
4: `passify()`

The individual steps of the preprocessing are summarized in Algorithm 1. For each loop l, we cut the associated back edge. Note that after this step, the CFG becomes a directed acyclic graph that underapproximates the behavior of the PUV. Next, we append the loop invariant $\text{LI}_l(U_l', R_l)$ to the loop header H. The semantic of $\text{LI}_l(\cdot)$ is that the variables in U_l' may take any value after the loop under the condition that together with the values of the variables in R_l before the loop, the loop invariant evaluates to true. Initially, we set all loop invariants to *true*. Intuitively, this means that after the loop invariant, all loop variables in U_l' may take arbitrary values and that the overall model overapproximates the behavior of the PUV.

In the main loop of our verification algorithm, we will check whether or not the program is safe using an SMT-solver. To this end, we construct an SMT formula F that is unsatisfiable only if the program is safe. We encode the PUV using three sets of constraints.

The first set of constraints assures that a possible solution to the SMT formula corresponds to a feasible flow through the program. Therefore, we associate with each basic block B_i in the CFG a Boolean variable b_i and add a clause

$$b_i \Rightarrow \bigvee_{B_j \in \text{Predecessors}(B_i)} b_j$$

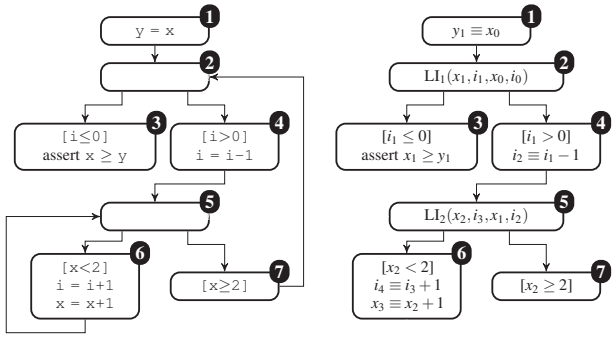
for each B_i with the exception of the entry basic block. The constraints assure that a B_i can only be visited if at least one of the predecessors is visited, too.

The second set of constraints aims at modeling the data-flow of the program. Therefore, we translate the PUV into a passive program using dynamic single-assignment [13]. The principal idea of this technique is that each time a variable is updated, the new value is assigned to a new variant of the variable. In our notation, we indicate the i^{th} variant of variable x with x_i. As an example, a statement such as $x = x + 1$ in the PUV would be represented as $x_{i+1} \equiv x_i + 1$. To represent a loop invariant $\text{LI}_l(U_l', R_l)$ in the passive program, we use the current variant for each variable that is referenced as current variable x and the next variant for each variable that is referenced as next variable x' in the loop invariant. Then, we increment the variant counter for each variable that is in U_l. For each passive formula f encoding a statement, assumption, or loop invariant in a basic block B_i, we add a clause of the form

$$b_i \Rightarrow f$$

Intuitively, the clause encodes that if the control flow visits B_i, then f must be true.

Lastly, in the third set of constraints, we constrain that the model of the PUV is not safe. For an assertion A in basic block $B_{\pi(A)}$ with



(a) Control Flow Graph (b) Passive Program

Figure 2: Nested Loops Example

passive formula a as condition to be violated, we must visit block $B_{\pi(A)}$ and a must not hold. The model is not safe if at least one assertion is violated. Hence, we have formally

$$\bigvee_{A \in \text{Assertions}} b_{\pi(A)} \wedge \neg a$$

If the so composed SMT-formula F is unsatisfiable, then the program is safe.

3.1.1 Example: Nested Loops

Consider the PUV given as CFG in Figure 2a. The numbers in the upper right corner of basic blocks indicate their numbering that we will use in the following for reference, e.g. the entry basic block will be denoted with B_1.

The PUV contains two loops $l_1 = (B_2, B_7)$ and $l_2 = (B_5, B_6)$. We remove the two back edges and append $\mathrm{LI}_1(\cdot)$ and $\mathrm{LI}_2(\cdot)$ to the loop heads B_2 and B_5, respectively. After passification, this yields the DAG in Figure 2b.

To check whether the obtained model of the PUV is safe, we construct an SMT-formula F as follows. We assure that every satisfying assignment corresponds to a valid control flow by adding the following constraints to F where, for instance, the first constraint assures that basic block B_2 cannot be visited unless B_1 is also visited.

$$\begin{array}{ccccc} (b_2 \Rightarrow b_1) & \wedge & (b_4 \Rightarrow b_2) & \wedge & (b_6 \Rightarrow b_5) \\ (b_3 \Rightarrow b_2) & \wedge & (b_5 \Rightarrow b_4) & \wedge & (b_7 \Rightarrow b_5) \end{array}$$

Next, we add the following data-flow constraints to F where e.g. the first constraint requires that if basic block B_1 is visited, then condition $y_1 \equiv x_0$ must hold.

$$\begin{array}{lcllcl} (b_1 & \Rightarrow & (y_1 \equiv x_0)) & \wedge & (b_2 & \Rightarrow & \mathrm{LI}_1(x_1,i_1,x_0,i_0)) \\ (b_3 & \Rightarrow & (i_1 \leq 0)) & \wedge & (b_4 & \Rightarrow & (i_1 > 0)) \\ (b_4 & \Rightarrow & (i_2 \equiv i_1 - 1)) & \wedge & (b_5 & \Rightarrow & \mathrm{LI}_2(x_2,i_3,x_1,i_2)) \\ (b_6 & \Rightarrow & (x_2 < 2)) & \wedge & (b_6 & \Rightarrow & (i_4 \equiv i_3 + 1)) \\ (b_6 & \Rightarrow & (x_3 \equiv x_2 + 1)) & \wedge & (b_7 & \Rightarrow & (x_2 \geq 2)) \end{array}$$

Lastly, we constrain that the assertion in B_3 must be violated. We enforce this by asserting that

$$b_3 \wedge (x_1 < y_1)$$

3.2 Iterative Invariant Refinement Loop

Algorithm 2 summarizes our proposed algorithm for program verification. After having preprocessed the program and having constructed the SMT formula F as discussed in the previous subsection, we check if F is satisfiable. If this is not the case, then we return that the program is safe. Otherwise, we obtain a satisfying assignment to all variables in F, cex c, and check whether c corresponds to a run that violates an assertion. To this end, we interpret the PUV with the arguments encoded in c. If the interpretation run violates an assertion, we report the program as not safe. Otherwise, the observed discrepancy between PUV and its model encoded as SMT formula is due to the weakness of one or more loop invariants. We scan through the program path admitting c in our abstract model and search for a loop l where LI_l admits the cex but its body_l does not. As with determining if a cex is real or spurious, we use interpretation to this end. Once we found such a loop l, we strengthen its invariant using `refine()` such that it remains an overapproximation of the loop but no longer admits c.

Algorithm 2 `verifyProgram()`

```
1: preprocess()
2: while counterexample c exists do
3:     if c real cex then return "Program not safe"
4:     let l be a loop whose LI_l(·) spuriously admits c
5:     refine(l, c)
6: return "Program safe"
```

Theorem 1. *Algorithm 2 is a sound and complete algorithm for intraprocedural verification of structured programs with static memory allocation.*

Proof. We sketch the proof of Theorem 1. We start by showing partial correctness. First note that if `verifyProgram()` returns "Program not safe", then it has found a real cex. To see why the program is safe if the algorithm returns "Program safe", note that at any time, F corresponds to a model that overapproximates the behavior of the PUV. Hence, safeness of the model implies safeness of the PUV. It remains to show that the algorithm terminates. As we assume static memory allocation, there is only a finite number of cex. As in each iteration of the main loop, at least one cex is invalidated, the algorithm must eventually terminate. □

3.2.1 Example: Nested Loops (cont'd)

With the loop invariants $\mathrm{LI}_1(\cdot)$ and $\mathrm{LI}_2(\cdot)$ being initially *true*, the PUV cannot be proved safe. Assume that the SMT-solver returned the cex

$$c = \{y_1 = 0, x_0 = 0, i_0 = 0, x_1 = -1, i_1 = 0, \dots\} \tag{1}$$

An interpretation run quickly shows that c is spurious. The cause is that $\mathrm{LI}_1(\cdot)$ is too weak. We call the procedure `refine()` to strengthen the loop invariant. The details of `refine()` are topic of the next subsection. For now, assume that the result of `refine()` is the new invariant

$$\mathrm{LI}_1(\cdot) = (x_0 < 0) \vee (x_1 > -1)$$

In the next iteration of the while-loop, we can find another spurious cex such as

$$c = \{y_1 = 1, x_0 = 1, i_0 = 0, x_1 = 0, i_1 = 0, \dots\} \tag{2}$$

Again, we refine the invariant for loop 1. This time, we obtain

$$\mathrm{LI}_1(\cdot) = (x_1 \geq x_0)$$

as new, strengthened loop invariant. With this invariant for loop 1, the SMT formula F is no longer satisfiable. This proves that the program is safe.

It is instructive to note that the derived loop invariant is not the strongest loop invariant that can be inferred for loop 1 but strong enough to prove the desired property. In this sense, our algorithm is also property directed. Also note the achieved abstraction from the cex. For instance, the inferred loop invariant is independent from variable i.

3.3 Property Directed Invariant Refinement

Algorithm 3 gives an overview of how LI_l is refined given a spurious cex c. After generalizing c, we invoke the QF_BV PDR algorithm with the intent to infer a strengthened loop invariant for l that contradicts c. If the attempt is successful, we update the SMT formula F to reflect the new loop invariant returned by the backend model checker. Note that this can be implemented by simply adding $b_H \Rightarrow \mathrm{LI}_l(\cdot)$ to the constraint base because any new loop invariant implies all previous loop invariants which allows for effective use of incremental solving.

In the presence of nested loops with weak loop invariants, it is also possible that PDR returns a spurious cex sequence s. In this case, it remains to determine an individual spurious transition c' in s and to find a nested loop l' whose invariant admits c' while $\mathrm{body}_{l'}$ does not. These tasks can be performed using interpretation similar as in Algorithm 2. Once a culprit loop l' is determined, `refine()` is called recursively to strengthen $\mathrm{LI}_{l'}$. This recursive strengthening is repeated until all loop invariants of nested loops are strong enough such that PDR succeeds in disproving c.

In the following subsection, we discuss details of the cex generalization and in Subsection 3.3.3 we describe the composition of the MCPs which are fed to PDR for the actual invariant refinement.

3.3.1 Generalization of Counterexamples

The motivation of the generalization of spurious cex at the beginning of the `refine()` procedure is to promote that the PDR model checker is finding loop invariants that do not only exclude the current cex but also a large number of related spurious cex. As such, it is a pure optimization that is not required for the correctness of the algorithm. To some extend, cex generalization mirrors simulation-based expansion of proof obligations in the PDR algorithm originally presented in [6].

We propose to generalize cex using abstract interpretation [9] with the abstract domains of integer intervals and ternary vectors. The choice for a concrete abstract domain is furnished by the kind of operation that calculates an abstract value. We use integer intervals if a result is calculated by an operation that interprets bitvectors as integers (e.g. addition). For all other operations (e.g. conjunctions), we use ternary vectors.

We start abstract interpretation runs at a loop header l and continue with the interpretation of the loop until a fixpoint is found. The final value sets at loop header l represent overapproximations of the reachable values of the variables for the given initial values.

Algorithm 4 gives details of how we use the results of abstract interpretation runs for the actual generalization. Note that we use

Algorithm 3 `refine(`l`, `c`)`

```
1: generalizeCex(l, c)
2: while PDR(I,T,c) does not hold do
3:     let s be the counterexample sequence
4:     let c' be a spurious transition in s
5:     let l' be a loop which LI_l' admits c' but body_l does not
6:     refine(l', c)
7: update F to reflect strengthened LI_l(·)
```

Algorithm 4 `generalizeCex(`l`, `$c = \{c_b, c_e\}$`)`

```
1: r = abstractInterpret(l, c_b)
2: for all x ∈ U_l do
3:     if disjoint(r[x], c_e[x]) then
4:         expand c_e[x] max. preserving disjointedness from r[x]
5:         for all y ≠ x ∈ U_l do c_e[y] = ⊤
6: for all x ∈ R_l do
7:     Substitute c_b[x] with largest value set v ⊇ c_b[x] s.t.
8:         ∃y ∈ U_l.abstractInterpret(l, c_b/v)[y] ∩ c_e[y] = ∅
```

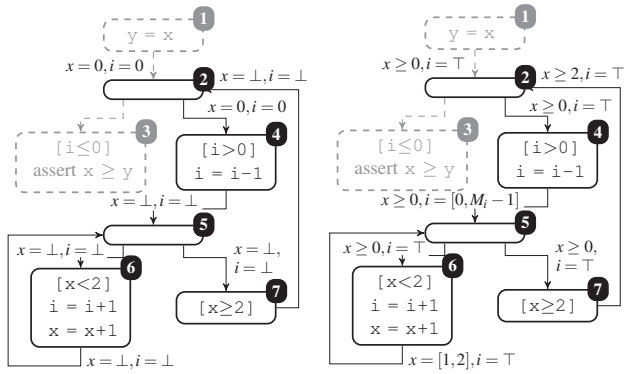
c_b (c_e) to denote the portion of c that corresponds to the values of the variables before (at the end) of the loop. Lines 1-5 of Algorithm 4 aim at generalizing c_e: The program is interpreted with the initial values given in c_b to obtain an overapproximation r of the reachable value sets for the variables at the loop header. If, for any variable in $x \in U_l$, $r[x]$ is disjoint from the corresponding value set in c_e, the value set $c_e[x]$ is expanded maximally while preserving the disjointedness from $r[x]$. This procedure guarantees that the resulting cex remains spurious regardless of the other variables. Hence, we can enlarge all other value sets in c_e to their maxima $\top$.

Lines 6-8 aim at generalizing the value sets for the c_b-portion of the cex: For each $x \in R_l$, we expand $c_b[x]$ maximally such that for at least one $y \in U_l$, the overapproximation of its reachable value set remains disjoint from the corresponding value set in c_e.

3.3.2 Example: Nested Loops (cont'd)

We consider the situation when `generalizeCex()` is called on loop 1 with the cex in equation (1). Mapping variants to variables, we have

$$c = \{c_b = \{x = 0, i = 0\}, c_e = \{x = -1, i = 0\}\}$$



(a) Fixpoint Abstract Interpretation 1 (b) Fixpoint Abstract Interpretation 2

Figure 3: Fixpoints during Cex Generalization in Example

An abstract interpretation run with the initial values c_b yields the fixpoint in Figure 3a. The reachable value sets at the loop header remain $r = \{x = 0, i = 0\}$. $c_e[x]$ is disjoint from $r[x]$, hence we can generalize c_e as defined in lines 4-5 of Algorithm 4 to obtain

$$c = \{c_b = \{x = 0, i = 0\}, c_e = \{x \leq -1, i = \top\}\}$$

Next, we attempt to expand the value sets for the initial values c_b. For x, decreasing the lower bound causes that the reachable value sets returned by the abstract interpretation ceases to be disjoint from the post-values in c_e. However, increasing the upper bound of x and expanding the value set for i arbitrarily preserve this condition (see fixpoint in Figure 3b where we denote with M_i the maximum value i can take). We arrive at the generalized cex

$$c = \{c_b = \{x \geq 0, i = \top\}, c_e = \{x \leq -1, i = \top\}\} \qquad (3)$$

3.3.3 Invariant Refinement with PDR

Given a loop $l = (H,T)$ and a cex c, the input MCP (I,T,B) of the PDR-algorithm for loop l is constructed as follows. For each variable $x \in U_l$, denote with x_b and x_e, respectively, the corresponding variants at the beginning and at the end of loop l. For each variable $x \in R_l \setminus U_l$, denote with x_b the corresponding variant. As initial condition I, we require that for each $x \in U_l$, we have that $x_e \equiv x_b$. Intuitively, this means that the values of a variable before and after the loop are equal before any iterations. The transition relation T is composed of four parts. First, for each $x \in R_l$, we constrain that the initial value does not change: $x'_b \equiv x_b$. Second, for each $x \in U_l$, we constrain $x'_e \equiv x_T$ where x_T is the last instantiated variant of x in the loop tail T. Third, we add the SMT formula corresponding to the part of the CFG that represents loop_l. Nested loops are represented by their loop invariants but their bodies are excluded. Fourth, we constrain b_T, i.e. that each transition requires that the loop tail is reached. B are the states that are contained in cex c.

This construction guarantees that the size of the MCP is bound by the size of the loop. Unrolling or "inlining" of nested loops are purposely avoided.

We preserve the trace constructed for loop l by the PDR algorithm between calls. This avoids that runtime required to prove that a certain subspace is unreachable in a certain number of steps is spent more than once and promotes finding stronger loop invariants. This memoization does not compromise the correctness of PDR: Consider any two successive invocations $\texttt{PDR}(I_1,T_1,B_1)$ and $\texttt{PDR}(I_2,T_2,B_2)$ for refinement of the same loop. The trace constructed during the first call may be reused in the second call, because we always call PDR with the same initial condition ($I_1 = I_2$) and any refinement of a nested loop invariant leads to a strengthening of the transition relation ($T_2 \Rightarrow T_1$). Hence, if a state was determined unreachable by PDR in the first call, it is also unreachable in the second call.

3.3.4 Example: Nested Loops (cont'd)

We use PDR to strengthen the loop invariant of loop 1 with the generalized cex in equation (3). We have

$$\begin{aligned} I &= (x_1 \equiv x_0) \wedge (i_1 \equiv i_0) \\ T &= (x'_0 \equiv x_0) \wedge (i'_0 \equiv i_0) \wedge (x'_1 \equiv x_2) \wedge (i'_1 \equiv i_3) \wedge (b_5 \Rightarrow b_4) \\ &\quad \wedge (b_7 \Rightarrow b_5) \wedge (b_4 \Rightarrow (i_1 > 0)) \wedge (b_4 \Rightarrow (i_2 \equiv i_1 - 1)) \\ &\quad \wedge (b_5 \Rightarrow \mathrm{LI}_2(\cdot)) \wedge (b_7 \Rightarrow (x_2 \geq 2)) \wedge b_7 \\ B &= (x_0 \geq 0) \wedge (x_1 \leq -1) \end{aligned}$$

Note that the current version of $\mathrm{LI}_2(\cdot)$ is used within the transition relation. As this loop invariant is currently *true*, this practically allows any update in one iteration. Assume that PDR returns the following cex sequence

$$i_0 = 0, x_0 = 0, i_1 = 0, x_1 = 0 \rightarrow i'_0 = 0, x'_0 = 0, i'_1 = 0, x'_1 = -1$$

As the sequence has only one transition, it is clear that this one must be spurious. Similarly, as there is only one nested loop within loop 1, it is clear that the current invariant $\mathrm{LI}_2(\cdot)$ causes the discrepancy between model and program. We extract the following cex for loop 2 by projecting the variants appropriately:

$$c' = \{x_1 = 0, i_2 = 0, x_2 = -1, i_3 = 0\}$$

and call `refine()` recursively on loop 2 and c'. The procedure returns after having refined the invariant of loop 2 to

$$\mathrm{LI}_2(\cdot) = (x_1 < 0) \vee (x_2 > -1)$$

In the next iteration of the while-loop, we call the PDR-solver with the same MCP as above but with the updated $\mathrm{LI}_2(\cdot)$ within the transition relation. With this strengthening, the cex can be disproved and PDR returns

$$\mathrm{LI}_1(\cdot) = (x_0 < 0) \vee (x_1 > -1)$$

When `refine()` is called the second time with the cex in equation (2), the process of strengthening $\mathrm{LI}_1(\cdot)$ is similar with one significant difference: We start PDR with the trace constructed in the previous invocation. Therefore, the subspace previously determined as unreachable is available and we will obtain

$$\mathrm{LI}_1(\cdot) = (x_1 \geq x_0)$$

which is strong enough to prove the safety of the program. The specifics of how the inequality is inferred in the backend QF_BV PDR model checker is beyond the scope of this paper. Please refer to [3] for details.

4 Related Work

The work most similar with respect to the high-level strategy for verifying assertions is InvGen [14]. InvGen also attempts to infer invariants and uses them to construct a combinational proof. In the backend, however, InvGen uses an entirely different strategy to infer the invariants. While our approach uses PDR, InvGen first applies a couple of dynamic strategies and then uses the template method from [10] as main reasoning method to find the invariants. This hybrid approach yields a relatively robust algorithm but has three main limitations: First, InvGen commits to conjunctions of linear inequalities as loop invariants. If the invariant required to prove a program safe is of any other form, the algorithm fails. Second, InvGen models programs using the quantifier free theory of linear arithmetic (QF_LA). Consequently, InvGen does not support programs with bit-level operations and the verification results are not guaranteed to be correct, e.g. in the presence of possible overflows. Third, InvGen is not able to generate cex. If the verification fails, it is not clear if the program is unsafe or if InvGen failed.

As our algorithm, TreeIC3 [15] uses PDR for program verification. TreeIC3 represents an extension of IMPACT [16] that uses interpolants to represent reachability information for nodes in an unrolling of the CFG. In this setup, TreeIC3 applies PDR as a complementary technique to infer the reachability information. This hybrid solution outperforms the original version with exclusive use of interpolation on a large fraction of benchmarks, often substantially. There are two main differences between our algorithm and TreeIC3: First, our algorithm calculates invariants for real locations in the PUV while TreeIC3 calculates invariants for nodes in an unrolling of the CFG. Second, as TreeIC3 calculates reachability information partially using interpolation, its success rests on the availability of an efficient interpolation algorithm. Such an interpolation procedure is not known for QF_BV. To avoid bit-blasting and the application of a Boolean interpolation algorithm, TreeIC3 contains an interpolation algorithm for the quantifier free theory of linear rational arithmetic (QF_LRA) [17]. This choice gives a very efficient algorithm, but does not allow for accurate modeling of any bitvector arithmetic.

5 Implementation and Experimentation

We implemented the presented algorithm in C++ and use functionality of the LLVM-framework [18] for parsing, representation, and

Benchmark	LOC	Prop. Directed Prog. Verif.			InvGen	TreeIC3	Symb.
		default	-gen.	-mem.	[14]	[15]	encod.
	(#)	Runtime (s)					
bind_exp_var	59	1.38	3.28	1.61	0.47	0.04	timeout
bound	47	0.71	2.3	3.94	0.96	fal. neg.	287.27
disj_simple	25	0.52	2.05	timeout	failed	fal. neg.	timeout
gulwani_cegar2	42	8.55	timeout	timeout	failed	0.06	timeout
heapsort1	58	38.47	timeout	timeout	1.05	0.12	timeout
id_build	34	0.18	2.64	1.65	0.41	fal. neg.	168.81
id_trans	43	0.48	1.75	0.5	0.38	0.09	timeout
mergesort	72	0.02	0.01	0.01	timeout	0.41	timeout
nested1	40	0.82	0.22	2.56	0.38	0.05	timeout
nested2	39	0.79	0.23	2.33	0.37	0.05	timeout
nest-if1	28	0.39	0.20	4.51	0.49	0.06	timeout
nest-len	30	0.68	1.16	17.81	0.74	0.31	timeout
NetBSD_loop	43	0.32	0.52	0.34	fal. pos.	fal. pos.	310.48
NetBSD_loop_int	47	2.49	0.84	3.39	fal. pos.	fal. pos.	timeout
sendmail_close	104	5.54	26.10	11.12	1.90	fal. neg.	timeout
sendmail_mime	89	6.73	2.94	timeout	failed	0.40	timeout
simple	38	0.54	1.04	527.05	0.19	fal. neg.	timeout
simple_if	26	0.55	0.44	0.66	failed	0.03	14.95
simple_nest	27	18.91	23.80	18.95	0.32	0.18	timeout
up_nested	28	0.01	0.01	0.01	0.65	0.05	1.06
drevil_2	21	1.51	2.85	2.15	failed	fal. neg.	timeout
jain_1	17	1.80	1.53	1.56	failed	0.02	timeout
jain_2	19	3.50	3.40	7.72	failed	0.03	timeout
for_bounded_lp	28	0.01	0.01	0.01	0.16	0.20	8.44
trex01_safe	50	0.51	1.02	0.98	failed	0.05	timeout
trex01_unsafe	50	0.01	0.01	0.01	failed	0.01	timeout
trex04	44	0.14	0.14	0.13	failed	0.03	timeout

Table 1: Performance of Verification Algorithms

analysis of the PUV. Working with the LLVM intermediate representation instead of a custom frontend for a specific language has the important advantage that we can verify programs of any language for which an LLVM-frontend exists. In our experimentation, we selected a number of C-language examples from the regression test suite of InvGen and from the bitvector and loop subsets of the SV-COMP software competition benchmarks [19].

Table 1 summarizes runtimes of the presented algorithm for the benchmarks. Column 1 gives the name of the benchmark and column 2 the lines of code as a crude measure of benchmark size. The next three columns contain runtime statistics of our algorithm once both with cex generalization and memoization of reachability information in the PDR trace (default), once without generalization, and once without memoization. As points of reference, columns 6 and 7 report runtimes when using InvGen and TreeIC3 on these benchmarks. We report failure if the program terminated without providing a result (either by giving up or due to a crash), timeout in case the program did not terminate within 600 seconds, false negative if the verification tool reported a safe PUV not to be safe, and false positive if an unsafe PUV is reported as safe.

Our verifier solves all verification tasks within reasonable runtime limits. Memory requirements are generally low (<100MB), which is explained by the avoidance of unrolling both in the frontend and in the backend model checker of the program verifier. Generalization of cex reduces the runtime often substantially. Without the use of memoization, the algorithm is less stable, documented by several timeouts. In comparison to our algorithm, InvGen terminates faster in the majority of cases. However, InvGen fails in roughly 40% of the benchmarks and returns a wrong verification result in two cases. TreeIC3 is even faster than InvGen but yields wrong verification results in 30% of the cases.

We also encoded the problems symbolically as QF_BV MCPs and used the model checker [4] directly without the invariant refinement proposed in this paper. The results are in the last column of Table 1; only six benchmarks can be solved within the time limit.

6 Conclusions and Future Work

We presented an algorithm for intraprocedural verification of safety properties in programs with static memory allocation. The algorithm leverages successfully applied techniques for verification tasks such as program verification with loop invariants and CEGAR. The core of our algorithm consists of the refinement of loop invariants using a QF_BV PDR model checker. We implemented the proposed algorithm in C++ using LLVM and presented results from experimentation with the software prototype that demonstrate overall good performance of the presented verification algorithm and its merits (correctness) and limitations (performance) in comparison with the closest related approaches.

As future work, we are interested in two research directions. First, we intend to investigate the potential to generalize our verification algorithm to allow interprocedural verification using function summaries. To this end, we conjecture that a similar property directed refinement strategy as applied to loop invariants could be used for function summaries. Second, we consider to complement our algorithm with lightweight but incomplete techniques such as DAIKON [11] to initialize loop invariants before starting property directed invariant refinement, potentially allowing runtime savings for the overall verification.

References

[1] M. Barnett and K. R. M. Leino, "Weakest-Precondition of Unstructured Programs," in *Proc. of the Workshop on Program Analysis for Software Tools and Engineering*, pp. 82–87, 2005.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[3] T. Welp and A. Kuehlmann, "QF_BV Model Checking with Property Directed Reachability," in *Proc. of the Int'l Conf. on Design, Automation and Test in Europe*, 2013.

[4] T. Welp and A. Kuehlmann, "QF_BV Property Directed Reachability with Mixed Type Atomic Reasoning Units," in *Proc. of Int'l Asia-South Pacific Design Automation Conference*, 2014.

[5] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Proc. of the 12th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, pp. 70–87, 2011.

[6] N. Eén, A. Mishchenko, and R. Brayton, "Efficient Implementation of Property Directed Reachability," in *Proc. of the Int'l Conf. on Formal Methods in Computer-Aided Design*, pp. 125–134, 2011.

[7] A. R. Bradley, "Understanding IC3," in *Proc. of the 15th Int'l Conf. Theory & Appl. Satisfiability Testing*, pp. 1–14, 2012.

[8] E. W. Dijkstra and W. H. Feijen, *A Method of Programming*. Addison-Wesley, 1988.

[9] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proc. of the Symp. on Principles of Programming Languages*, pp. 238–252, 1977.

[10] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear Invariant Generation Using Non-Linear Constraint Solving," in *Computer Aided Verification*, vol. 2725 of *LNCS*, pp. 420–432, 2003.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, pp. 213–224, 2001.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[13] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *Int'l Journal of Parallel Programming*, vol. 20, pp. 23–53, 1991.

[14] A. Gupta and A. Rybalchenko, "InvGen: An Efficient Invariant Generator," in *Computer Aided Verification*, vol. 5643 of *LNCS*, pp. 634–640, 2009.

[15] A. Cimatti and A. Griggio, "Software Model Checking via IC3," in *Computer Aided Verification*, vol. 7358 of *LNCS*, pp. 277–293, 2012.

[16] K. L. McMillan, "Lazy Abstraction with Interpolants," in *Computer Aided Verification*, vol. 4144 of *LNCS*, pp. 123–136, 2006.

[17] K. McMillan, "An Interpolating Theorem Prover," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988 of *LNCS*, pp. 16–30, 2004.

[18] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2004.

[19] D. Beyer, "Competition on Software Verification," in *Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 504–524, 2012.