Loop Refinement Using Octagons and Satisfiability

Jörg Brauer, Volker Kamin,
Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
lastname@embedded.rwth-aachen.de

Thomas Noll
Software Modelling and Verification Group
RWTH Aachen University
noll@cs.rwth-aachen.de

Abstract

This paper presents a technique for refining the control structure of loops in programs operating over finite bit-vectors. This technique is based on abstract interpretation using octagons and affine equalities in order to identify infeasible sequences of loop iterations. Our approach naturally integrates wrap-around arithmetic during the generation of abstractions. Abstract interpreters operating on a refined control structure then typically derive strengthened program invariants without having to rely on complicated domain constructions such as disjunctive completions.

1 Introduction

Microcontroller programs typically consist of an infinite while-loop that incorporates several tasks such as sensing inputs or processing data. These loops exhibit a complex control structure, based on conditional branching, which depends on certain bits of the status register. Hence, when verifying microcontroller programs, it is required to reason about programs at the granularity of bits [8, 4]. This poses one problem to verification efforts based on abstract interpretation [13].

Further, the analysis of such loops often requires the application of widenings [15] to guarantee or accelerate termination, which often induces a significant loss of precision. To alleviate this problem, we propose to refine the control flow graph (CFG) of the program, following the ideas described by Balakrishnan et al. [5]. Our method is based on deriving systems of transfer functions for sequences of instructions that constitute paths through a loop [7]. The transfer functions, which are derived using SAT solving, consist of pairs of guards on the inputs and updates on the outputs to incorporate the effects of register overflows. Based on the application of the transfer functions, infeasible sequences of loop iterations are detected to guide the refinement of the CFG.

1.1 Illustrative Example

To illustrate the application of this technique, consider the (stripped) program given in Fig. 1. This code fragment implements a state machine typical to microcontroller code, taken from a program that controls a light switch with three operation modes (on, off, and dimmed). The task of the program is to change the internal state of the program in case a button is pushed. A typical property to be verified for this program is that when $\mathtt{m} = 0$ holds, then $\mathtt{m} = 1$ has to hold before $\mathtt{m} = 2$. This means that, when the light is turned off, it starts as dimmed before it is fully turned on. Typical path-insensitive analyses that operate on the original CFG fail to verify this property because the abstract values are merged at the end of the `switch` statement. Overall, there are five possible paths through the loop:

$$\pi_1 = \langle 1, 2, 3, 4, 10 \rangle \qquad \pi_2 = \langle 1, 2, 3, 5, 6, 8, 10 \rangle$$
$$\pi_3 = \langle 1, 2, 3, 5, 7, 8, 10 \rangle \qquad \pi_4 = \langle 1, 2, 3, 9, 10 \rangle$$
$$\pi_5 = \langle 1, 2, 10 \rangle$$

The infeasibility of the sequence $\langle \pi_1, \pi_4 \rangle$ corresponds to the property to be verified. Our approach first identifies the postcondition of π_1 and the precondition of π_4:

$$\mathsf{post}(\pi_1) \;=\; (\mathtt{m} = 1) \qquad \mathsf{pre}(\pi_4) \;=\; (\mathtt{m} = 2)$$

Based on $\mathsf{post}(\pi_1) \sqcap \mathsf{pre}(\pi_4) = \emptyset$, the CFG is refined to resemble the infeasibility of this sequence. Similarly, the infeasibility of $\langle \pi_4, \pi_2 \rangle$ and $\langle \pi_4, \pi_3 \rangle$ can be derived. Finally, this leads to the refined CFG depicted in Fig. 2. The edges for π_5, which can be executed from any state, are omitted to make the presentation accessible. The desired property can then be verified using a straightforward analysis on the refined CFG, dismissing the application of more sophisticated domain constructions such as disjunctive completion [29].

```
while (1) {
  if (isButtonPressed()) {
    switch (m) {
      case 0: m = 1; break;
      case 1:
        if (TIFR & (1 << OCF1A)) m = 0;
        else m = 2;
       break;
      case 2: m = 0;
  }
}
```

Figure 1: Loop from a light-switch controller

1.2 Abstract Interpretation of Sequences

Our tool for binary code verification, called [MC]SQUARE, is based on abstract interpretation [30, 31]. Given a concrete domain C, the key idea of abstract interpretation is to simulate the execution of a concrete operation $f : C \to C$ using an abstract counterpart $g : D \to D$, where D is an abstract description of C, written as $g \propto f$. Suppose that a program fragment consists of a sequence of n concrete operations $f_1, f_2, \ldots, f_n$, where each concrete operation $f_i : C \to C$ has its own abstract counterpart $g_i : D \to D$, called its transfer function. Then, the effect of applying the n concrete operations to some input is described by applying the composition of the n transfer functions, that is, $(g_n \circ \cdots \circ g_1) \propto (f_n \circ \cdots \circ f_1)$.

However, a more precise description of $f_n \circ \cdots \circ f_1$ can be found by deriving a single transfer function g such that $g \propto (f_n \circ \cdots \circ f_1)$. This approach is particularly valuable in the context of bit-twiddling programs [20, 21, 7] by allowing to capture relational information among intermediate instructions whose semantics exceeds the expressiveness of the respective abstract domain.

1.3 Refining Loops

Our approach builds upon the work presented in [7, 21], where transfer functions for numeric abstract domains such as octagons [25], bit-level congruences [21], and affine relations [19] are automatically derived using SAT solving. Given a set of program variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, our method derives preconditions for whole paths through a loop in terms of octagons, which are relationships of the form $\pm v_i \pm v_j \leq c$ with $c \in \mathbb{Z}$. These preconditions are induced through guards (such as the conditions on the value of m in the introductory example).

Further, we use the technique from [7] to derive affine transfer functions of the form $\sum_{i=1}^{n} \lambda_i \cdot v_i = d$ ($\lambda_i, d \in \mathbb{Z}$) for paths. Observe that instructions may cause overflows, whereas the expressiveness of affine relations does not suffice for representing modular arithmetic. Hence, our approach derives additional octagons that over-approximate combinations of inputs leading to overflows, underflows, or normal operation. This results in several pairs of octagons and affine updates to describe the effects of a path. Each of these pairs covers a single wrap-around mode.

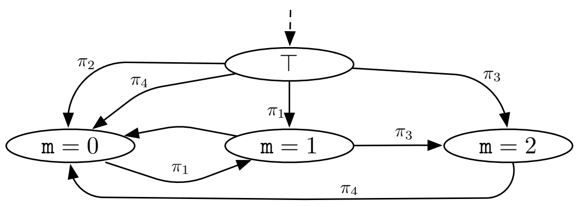

Figure 2: Refined CFG of light-switch controller

Example To illustrate, consider j = i+1. If the operation is applied to $\mathtt{i} = 127$ (in two's complement), the operation results in -128. Hence, this operation is non-affine over finite bit-vectors. The operation, however, can be described by two guarded affine relations:

$$\begin{array}{lllll}(-128 \leq \mathtt{i} & \wedge & \mathtt{i} \leq 126) & \Rightarrow & (\mathtt{j} = \mathtt{i} + 1) \\ (127 \leq \mathtt{i} & \wedge & \mathtt{i} \leq 127) & \Rightarrow & (\mathtt{j} = -128)\end{array}$$

Note that a disequality on a single variable, say, $x \leq c$ can equivalently be expressed as an octagonal constraint $x + x \leq 2 \cdot c$.

Approach Given two paths π_i and π_j through a loop, a precondition $\mathsf{pre}(\pi_i)$ is derived and $\mathsf{pre}(\pi_i)$ is transformed into a postcondition $\mathsf{post}(\pi_i)$ using the affine update. This step is based on linear programming. If $\mathsf{post}(\pi_i) \sqcap \mathsf{pre}(\pi_j) \neq \emptyset$, an execution of π_i could be followed by π_j, and the control flow graph of the program is extended in order to resemble the feasibility of this sequence. This idea is fully developed in the remainder.

1.4 Contributions

In this paper, we make the following contributions:

- We describe how to derive pre- and postconditions for sequences of operations over bounded integers, which are expressed in terms of octagons.
- We show how to refine the CFG of a program based on feasible sequences of loop iterations.
- We detail several refinement strategies and discuss the issue of convergence.
- We discuss two approaches for analyzing nested loops, namely bottom-up and top-down refinement.

2 A Primer on SAT-Based Abstraction

Encoding the semantics of instructions using propositional logic, often colloquially referred to as *bit-blasting*, has become a standard technique in program analysis and model checking [7, 8, 11, 12, 20, 22, 21, 36]. Our approach for deriving transfer functions of sequences is based on the approach of Brauer and King [7], who extended the techniques put forward by Monniaux [26] to derive abstractions of Boolean formulae for linear template constraint domains, most notably octagons [25].

2.1 Bit-Blasting Blocks

To illustrate the process of deriving octagonal abstractions, let $\mathcal{V} = \{v_1, \ldots, v_n\}$ denote the set of program variables accessed in a block. Given a word-length w, bit-vectors $\boldsymbol{v} = \langle \boldsymbol{v}[0], \ldots, \boldsymbol{v}[w-1] \rangle$ and $\boldsymbol{v}' = \langle \boldsymbol{v}'[0], \ldots, \boldsymbol{v}'[w-1] \rangle$ for the inputs and outputs of the block are introduced for each $v \in \mathcal{V}$. Further, let $\boldsymbol{V} = \{\boldsymbol{v} \mid v \in \mathcal{V}\}$ and $\boldsymbol{V}' = \{\boldsymbol{v}' \mid v \in \mathcal{V}\}$ such that $\boldsymbol{V} \cap \boldsymbol{V}' = \emptyset$. Then, the semantics of the block (or path) π can be described propositionally using a Boolean formula $\varphi_\pi(\boldsymbol{V}, \boldsymbol{V}')$ over $\boldsymbol{V} \cup \boldsymbol{V}'$ that relates the inputs $\boldsymbol{V}$ to the outputs $\boldsymbol{V}'$.

2.2 Deriving Octagon Abstractions

To explain the derivation of an octagonal relation $x+y \leq c$ between two bit-vectors $\boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{V}$, let $\boldsymbol{V} = \{\boldsymbol{x}, \boldsymbol{y}\}$ and $\boldsymbol{V}' = \{\boldsymbol{x}', \boldsymbol{y}'\}$ for simplicity. Further, introduce an additional bit-vector $\boldsymbol{c}$ for the upper bound. Here, we assume that operands are signed, and therefore, let $\langle\!\langle \boldsymbol{x} \rangle\!\rangle = \sum_{i=0}^{w-2} 2^i \boldsymbol{x}[i] - 2^{w-1}\boldsymbol{x}[w-1]$. Then put:

$$\theta = \forall \boldsymbol{x} : \forall \boldsymbol{y} : (\varphi_\pi(\boldsymbol{V}, \boldsymbol{V}') \Rightarrow (\langle\!\langle \boldsymbol{x} \rangle\!\rangle + \langle\!\langle \boldsymbol{y} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c} \rangle\!\rangle))$$

Clearly, θ characterizes sound values of $\boldsymbol{c}$ for the constraint $\langle\!\langle \boldsymbol{x} \rangle\!\rangle + \langle\!\langle \boldsymbol{y} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{c} \rangle\!\rangle$. However, for an 8-bit architecture, the value $\langle\!\langle \boldsymbol{c} \rangle\!\rangle = 510$ satisfies θ because $\langle\!\langle \boldsymbol{x} \rangle\!\rangle \leq 255$ and $\langle\!\langle \boldsymbol{y} \rangle\!\rangle \leq 255$. Optimality can be specified using an additional formula ψ that operates over disjoint bit-vectors $\hat{\boldsymbol{V}}$ and $\hat{\boldsymbol{V}}'$, respectively, in order to avoid accidental coupling. Then, optimality is imposed using the following characterization:

$$\begin{array}{rcl} \psi & = & \forall \hat{\boldsymbol{x}} : \forall \hat{\boldsymbol{y}} : \forall \hat{\boldsymbol{c}} : \\ & & ((\varphi_\pi(\hat{\boldsymbol{V}}, \hat{\boldsymbol{V}}') \Rightarrow \langle\!\langle \hat{\boldsymbol{x}} \rangle\!\rangle + \langle\!\langle \hat{\boldsymbol{y}} \rangle\!\rangle \leq \langle\!\langle \hat{\boldsymbol{c}} \rangle\!\rangle) \Rightarrow \\ & & (\langle\!\langle \boldsymbol{c} \rangle\!\rangle \leq \langle\!\langle \hat{\boldsymbol{c}} \rangle\!\rangle)) \end{array}$$

The formula expresses that, whenever there exists another bound $\hat{\boldsymbol{c}}$ for the octagonal constraint, then $\boldsymbol{c} \leq \hat{\boldsymbol{c}}$ has to hold. Hence, while θ characterizes sound valuations of $\boldsymbol{c}$, the formula ψ induces optimality. Consequently, $\theta \wedge \psi$ characterizes safe and optimal solutions of the constant $\boldsymbol{c}$. The force of this approach is that by putting θ and ψ into conjunctive normal form, universal quantifier elimination becomes trivial: Eliminating a variable simply amounts to removing all corresponding literals [22].

Finally, a quantifier-free version of $\theta \wedge \psi$ is passed to a SAT solver and the value of $\boldsymbol{c}$ is extracted from the satisfying assignment. A generalization of this idea for *all* octagonal constraints is found in [7]. We abstain from presenting further details and refer to this operation as $\alpha_{\mathsf{Oct}}(\pi)$.

2.3 Deriving Affine Abstractions

Based on the same propositional encoding $\varphi_\pi(\boldsymbol{V}, \boldsymbol{V}')$, affine relations among program variables can be derived using successive calls to a SAT solver. The algorithm described in [7] computes the affine closure of $\varphi_\pi(\boldsymbol{V}, \boldsymbol{V}')$ for *range constraints*. In essence, the method used corresponds to a symbolic implementation of the best transformer as described by Reps et al. [28].

Our technique used here is simpler in that it discards range constraints in the propositional encoding, and only computes direct affine relations between input and output variables. This is a straightforward application of the closure computation described in [7, Sect. 3.2]. In the following, we refer to this operation, which yields a conjunction of affine equalities connecting input and output variables, as $\alpha_{\mathsf{Aff}}(\pi)$.

Example Consider an assignment such as $z = y + z$. Whereas the semantics of this assignment is trivial over unbounded integers, this is not so in the bounded case, as the $+$ operation can over- or underflow. The SAT-based method derives three octagonal guards that distinguish overflow, underflow, and normal operation modes:

$$\begin{array}{rcl} g_O & = & (1 \leq x \wedge 1 \leq y \wedge 128 \leq x + y) \\ g_U & = & (x \leq -1 \wedge y \leq -1 \wedge x + y \leq -129) \\ g_E & = & (-128 \leq x + y \wedge x + y \leq 127) \end{array}$$

Now, consider the overflow case only. The wrap-around leads to a result z that is the sum of x and y shifted by 256. Computing the affine closure of the corresponding Boolean formula gives:

$$\varphi_O = (x + y - 256 = z)$$

Accordingly, the transfer functions for the other two cases are synthesized, which yields:

$$\begin{array}{l} \varphi_U = (x + y + 256 = z) \\ \varphi_E = (x + y = z) \end{array}$$

This method can be applied to arbitrary sequences of operations over bounded integers (or bit-vectors, equivalently), as we will see in the following section.

```
1: while (x < 0) {
2:   y = y+ abs(x);
3:   if (y > 64) {
4:     x = abs(y + 1);
5:   }
6: }
7: return x;
```

Figure 3: Termination bug due to wraps

3 Detecting Termination Bugs

To illustrate our method, consider the example program given in Fig. 3 for an 8-bit architecture. As long as x is smaller than 0, the loop is executed, and first, the absolute value of x is added to y. If $\mathtt{y} > 64$ holds, then x is assigned the absolute value of y incremented by 1, and the loop terminates.

3.1 Termination Bug Explained

This program, however, contains a termination bug that is introduced through the limited bit-width of machine arithmetic. Suppose that the loop is entered with $\mathtt{x} = -1$ and $\mathtt{y} = 126$. Then, in line 3, the variable y is assigned the value 127. Clearly, the condition $\mathtt{y} > 64$ is satisfied, and thus, x is assigned the value of `abs(y + 1)`, where `y+1` wraps. However, the absolute value of -128 is -128 (since the domain is bounded by $[-128, 127]$ in two's complement), and consequently, x is still negative. Then, in the second iteration, y is assigned the value -1. In the third iteration, y is assigned 127 due to an underflow, the condition $\mathtt{y} > 64$ is satisfied, and x is set to -128 again. The loop keeps following these steps and never terminates.

3.2 Deriving Transfer Functions

Here, we discuss the derivation of the transfer functions step-by-step. Consider the operation `y = y + abs(x)`. For the function `abs(x)`, we obtain three different guarded updates:

$$\begin{array}{lcl}(-128 \leq x \leq -128) & \Rightarrow & (\mathtt{abs}(x) = -128)\\ (-127 \leq x \leq -1) & \Rightarrow & (\mathtt{abs}(x) = -x)\\ (0 \leq x \leq 127) & \Rightarrow & (\mathtt{abs}(x) = x)\end{array}$$

When considering the assignment `y = y + abs(x)`, we obtain:

$$\begin{array}{lcl}(-128 \leq x \leq -128) & \Rightarrow & (y' = y -_{\mathcal{BV}} 128)\\ (-127 \leq x \leq -1) & \Rightarrow & (y' = y -_{\mathcal{BV}} x)\\ (0 \leq x \leq 127) & \Rightarrow & (y' = y +_{\mathcal{BV}} x)\end{array}$$

Here, the output variable is primed. It is important to note that the $+_{\mathcal{BV}}$ and $-_{\mathcal{BV}}$ in these equations are operations over bit-vectors, and thus, require a case distinction when expressed as unbounded operations since they can wrap. Now, consider the first equation only. For $-128 \leq x \leq -128$, the assignment $y' = y - 128$ cannot overflow, thus we have two remaining equations:

$$\begin{array}{l}(-128 \leq x \leq -128 \wedge 0 \leq y \leq 127)\\ \qquad \Rightarrow (y' = y - 128)\\ (-128 \leq x \leq -128 \wedge -128 \leq y \leq -1)\\ \qquad \Rightarrow (y' = 128 + y)\end{array}$$

By splitting $-_{\mathcal{BV}}$ into different cases, the difference cannot wrap anymore. When we additionally include the guard $y' \leq 64$ in the derivation process (the branching condition is violated), it turns out that the method yields stronger preconditions:

$$\begin{array}{l}(-128 \leq x \leq -128 \wedge 64 \leq y \leq 127)\\ \qquad \Rightarrow (y' = y - 128)\\ (-128 \leq x \leq -128 \wedge -128 \leq y \leq -64)\\ \qquad \Rightarrow (y' = 128 + y)\end{array}$$

If this process is applied to all possible combinations of paths through the loop, with the understanding that wraps are handled using case distinction, we end up with 13 different transfer functions.

Then, the refinement process, starting with $\top$, is executed. Applying the transfer function above yields the postcondition $-128 \leq x \leq -128 \wedge 0 \leq y \leq 64$ from the precondition $\top \sqcap (-128 \leq x \leq -128 \wedge -128 \leq y \leq -64)$. Overall, the process exhibits that there exists a run that reaches a strongly connected component without any connection to the exit node. This graph structure exhibits the termination bug that was described in the beginning of this section.

4 Refining Loops

This section formally describes the details of the refinement algorithm, which operates on the set of *all* paths through a given loop. We abstain from formally defining loops and paths here and refer the interested reader to [5]. Informally speaking, we assume that a loop consists of an input node, a set of paths (sequences of instructions that have a single successor) through the loop, and a distinguished exit path.

4.1 Refinement

For refining a loop L, the following preprocessing steps are performed first:

1. Enumerate the set of all paths π_i through the loop L, denoted Π.

```
proc refine input:  $T$ = $\{\langle \mathsf{pre}(\pi_{i,j}), \varphi_{i,j}\rangle\}$
            output: directed graph $G$
begin
  $G$ = $(\mathsf{initial}, \emptyset)$
  worklist = $\{(\mathsf{initial}, f) \mid f \in T\}$;
  while (not worklist.isEmpty()) do
    $\langle \mathsf{inv}, \langle \mathsf{pre}(\pi_{i,j}), \varphi_{i,j}\rangle\rangle$ = worklist.pop()
    if ($\mathsf{inv} \sqcap g \neq \emptyset$) then
      $\mathsf{post}$ = $\varphi(\mathsf{inv} \sqcap g)$
      if (not $G$.contains($\mathsf{post}$)) then
        $G$.addVertex($\mathsf{post}$)
        worklist.add($\{\langle \mathsf{post}, f\rangle \mid f \in T\}$)
      fi
      $G$.addEdge($\mathsf{inv}$, $\mathsf{post}$, $\langle i, j\rangle$)
    fi
  od
  return $G$
end proc
```

Figure 4: Refinement algorithm

2. Derive transfer functions for each $\pi_i \in \Pi$. The set of transfer functions for each π_i consists of k_i pairs $\langle \mathsf{pre}(\pi_{i,j}), \varphi_{i,j}\rangle$, where the $\mathsf{pre}(\pi_{i,j}) = \alpha_{\mathsf{Oct}}(\pi_{i,j})$ are octagonal guards and $\varphi_{i,j} = \alpha_{\mathsf{Aff}}(\pi_{i,j})$ are affine updates (with $1 \leq j \leq k_i$).

The refinement process generates a labeled directed (and most likely cyclic) graph $G = \langle V, E, \mu\rangle$, where nodes represent program invariants[1] and edges correspond to executions of paths. Additionally, $\mu : E \to \Pi$ is an edge-labeling function. Starting from a root node that corresponds to the initial invariant, the algorithm (cp. Fig. 4) proceeds as follows:

1. Build a worklist, consisting of pairs of invariants and transfer functions to be executed. In the beginning, the worklist contains all pairs of the initial program invariant and transfer functions $\langle \mathsf{pre}(\pi_{i,j}), \varphi_{i,j}\rangle$.

2. If the worklist is empty, terminate. Otherwise, remove a pair $\langle \mathsf{inv}, \langle \mathsf{pre}(\pi_{i,j}), \varphi_{i,j}\rangle\rangle$ from the worklist and compute $\mathsf{inv} \sqcap \mathsf{pre}(\pi_{i,j})$.

3. If $\mathsf{inv} \sqcap \mathsf{pre}(\pi_{i,j}) = \emptyset$, proceed with step 2. In this case, the sequence is infeasible.

4. Apply the transfer function $\varphi_{i,j}$ to $\mathsf{inv} \sqcap \mathsf{pre}(\pi_{i,j})$, and denote the outcome by $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$. The result is an octagonal invariant.

5. If there exists a node $v \in V$ representing the invariant $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$, add an edge labeled with $\langle i, j\rangle$ from the current vertex inv to v.

6. Otherwise, insert a fresh node into G and add an edge from inv to $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$. In this case, add pairs $\langle \mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j})), \langle \mathsf{pre}(\pi_{i',j'}), \varphi_{i',j'}\rangle\rangle$ into the worklist for all transfer functions $\langle \mathsf{pre}(\pi_{i',j'}), \varphi_{i',j'}\rangle$ and proceed with step 2.

Discussion of Intrinsics For all involved operations, algorithms are known from the literature. Our contribution here is a novel combination of these existing algorithms. The intersection $\sqcap$ of two octagons is computed by representing the octagons in terms of difference-bound matrices (DBMs) and computing the component-wise minimum [25, p. 10]. Whereas the guards derived for blocks a-priori are tightly closed – intuitively, this means that all hyperplanes defined through inequalities actually touch the enclosed volume – the greatest lower bound operator $\sqcap$ does not preserve this property [25, p. 28]. On the other hand, testing emptiness of octagons is based on investigating cycles in their potential graphs [1, 25], but the algorithm is only applicable to tightly closed octagons (in case of integral solutions). Hence, the tight closure of $\mathsf{inv} \sqcap g_{i,j}$, which can be computed in cubic time [2], has to be used as the input for the emptiness test. Essentially, the tight closure makes all implict constraints explicit.

As an alternative, one could use the two-phase Simplex algorithm for testing emptiness, since the first stage of the two-phase Simplex algorithm amounts to deciding feasibility of the system by solving the so-called auxiliary problem [10]. Transforming an invariant using an affine relation to obtain an octagonal postcondition is based on linear programming.

Finally, a new (larger) CFG for the loop is generated by simply expanding all edges in the resulting graph G. We discuss the details of the applied operations in the remainder of this section.

4.1.1 Transforming $\mathsf{inv} \sqcap \mathsf{pre}(\pi)$

The interesting step in our algorithm amounts to applying an affine transfer function φ to an octagonal invariant $\mathsf{inv} \sqcap \mathsf{pre}(\pi)$ in order to obtain $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi))$. While it is possible to directly transform the octagonal constraints by applying the affine transfer function, this would yield a convex polyhedron in general: While the dimensionality does not grow, the unit gradient of the octagonal hyperplanes might be scaled arbitrarily.

Numerous ways exist for deriving octagonal abstractions from convex polyhedra. The seminal paper on octagons [25, Sect. 4.3] described an exact abstraction for $\mathbb{Q}$ and $\mathbb{R}$, based on a *frame representation* of the convex polyhedron. A frame consists of a finite set of vertices $\mathcal{V} = \{v_1, \ldots, v_k\}$ and a finite set of rays

[1] Equivalently, one could call the nodes *states*. The term *invariant* expresses that the state represented by the node is an invariant whenever the program traverses a (possibly infinite) path through G to reach the respective node.

$\mathcal{R} = R_1, \ldots, R_l$ such that:

$$\left\{ \sum_{i=1}^{k} \lambda_i V_i + \sum_{i=1}^{l} \mu_i R_i \mid \lambda_i \geq 0, \mu_i \geq 0, \sum_{i=0}^{k} \lambda_i = 1 \right\}$$

Extracting octagonal constraints from the frame has the worst-case complexity $\mathcal{O}(n^2 \times (|V| + |R|))$, and optimality is guaranteed for solutions in $\mathbb{Q}$ or $\mathbb{R}$ only. To remedy this computational bottleneck, we use a different (non-optimal) method based on the Simplex algorithm [1, 22]. While this method has exponential worst-case complexity in theory, it is extremely efficient in practical applications (and has shown linear complexity in all our experiments).

Algorithm $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi))$ consists of a conjunction of (at most 8) octagonal constraints for every two $v_i, v_j \in \mathcal{V}$, each of which is of the form $\pm v_i' \pm v_j' \leq c'$. Additionally, we have a precondition that is a conjunction of inequalities, and an update that is an affine relation consisting of a conjunction of affine equalities. To obtain c', our approach uses linear programming applied separately to each constraint.

That is, it maximizes the target function $\pm v_i' \pm v_j'$ subject to the additional constraints $\mathsf{pre}(\pi)$ and φ, which yields c'. In case that $c' \notin \mathbb{N}$, we define the resulting constraint as $\pm v_i' \pm v_j' \leq \lfloor c' \rfloor$ if $c' \geq 0$ and $\pm v_i' \pm v_j' \leq \lceil c' \rceil$ otherwise, which is safe but non-optimal. This step is repeated for each octagonal output constraint, such that the method yields $\mathsf{post}(\pi)$.

Complexity The Simplex algorithm operates on $\mathbb{Q}$, while our domain consists of bounded integers. However, we only consider the relaxed (non-optimal) optimization problem for performance reasons. Further, if the constraints consist of two variables per inequality, the worst-case complexity is cubic [34].

Example As an example, consider an octagonal precondition $x + y \leq 96$ and the following affine system:

$$x' = 2x + y \qquad y' = 4y$$

To derive an upper bound for $\theta = x' + y'$, maximize θ subject to the following constraints:

$$\begin{array}{lcl} x' - 2x + y & = & 0 \qquad x + y \leq 96 \\ y' - 4y & = & 0 \end{array}$$

Simplex then computes the maximal solution $x' + y' = 85.\bar{3}$ in three iterations. Since $85.\bar{3} \notin \mathbb{N}$, we set $c' = \lfloor 85.\bar{3} \rfloor = 85$. This solution over-approximates the integral system because the maximum value is derived for $x = 10.\bar{6}$ and $y = 21.\bar{3}$, whereas the maximal value obtained through integer linear programming (ILP) is 80.

4.2 Convergence

The refinement is guaranteed to terminate due to monotonicity of the intersection and the finiteness of the domain. Even though octagons do not satisfy the finite ascending chain condition when defined over unbounded integers, this is not so for the bounded case, and therefore, only a finite number of vertices can be inserted into the graph.

While the runtime requirements for the refinement did not cause any problems in our experiments, they might become an issue for larger and more complex loops. We suggest two strategies that enforce faster termination of the refinement process: The maximum depth of G can be bounded, or weak refinements can be ignored.

4.2.1 Bounded Refinement

Using this strategy, each node of G stores its minimal distance from the root node $\top$. This is done by assigning 0 to the root node and assigning ∞ to all newly created nodes. A node's distance is updated each time a new edge is added targetting it.

Step 6 of the algorithm then checks whether the current node's distance is already equal to a given upper bound. In this case, no fresh node is added, but instead an existing node is used as replacement. A valid candidate for a replacement must represent an invariant that is weaker than the current $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$. All such candidates preserve are sound, and $\top$ will always be a valid candidate. Using a breadth-first strategy in the worklist, it is guaranteed that all nodes are created before the first replacement is needed. Therefore, when choosing a replacement, all possible candidates are known and an optimization criterion can be used for selection.

Algorithm Let $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$ be the invariant that no fresh node may be created for and let $\mathsf{inv}_1, \ldots, \mathsf{inv}_n$ be all valid replacement candidates, that is, we have $\mathsf{inv}_i \sqsupseteq \mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi_{i,j}))$ for all $1 \leq i \leq n$. We suggest three different optimization criteria:

1. Select the candidate with the smallest volume.
2. Only consider variables that are directly used in conditional branches for calculating the volume. Then, select the candidate with the smallest volume.
3. For all candidates and all variables calculate the increase of range. Select the candidate with lowest maximum increase.

Using the first strategy, the overall weakening of the invariant is minimized. Using the second one, special consideration is given to variables that directly influence the control flow (i.e., possible paths). The last strategy inhibits drastic changes in one particular dimension.

```
 1: x=1;
 2: while (x < len) {
 3:   y = x + 1;
 4:   do {
 5:     if ((y & x) != 0) {
 6:       a[x] ^= a[y];
 7:       y++;
 8:     } else {
 9:       y += x;
10:     }
11:   } while (y <= len);
12:   x <<= 1;
13: }
```

Figure 5: Hamming code calculation

4.2.2 Norm-Based Refinement

The bounded refinement approach only searches for replacements to limit the depth of G. However, it might be hard to choose a suitable bound a-priori. Another way to accelerate converge is to inhibit refinements that do not lead to significantly strengthened invariants, compared to existing ones in G. Using the norm-based refinement strategy, each time a fresh node is about to be added to G, the algorithm compares all existing nodes using one of the optimization criteria from above. If the new node is too similar to an existing node (e.g., the volume is not sufficiently lower), that node is used as a replacement.

4.3 Finite Automata & Regular Languages

Observe that the refined CFG can equivalently be interpreted as a finite automaton, and thus, defines a regular language over the alphabet Π constraining the possible loop executions. This is the interpretation chosen in [5]. Naturally, this leads to the question if a more expressive class of languages or automata provides additional benefit. An extension towards context-free languages (or equivalently, pushdown automata) appears to be natural.

5 Nested loops

Until now, we have only considered simple loops without inner loops. Nested loops, however, are very common in practice. This section describes two strategies of extending the refinement algorithm to nested loops.

For both refinement strategies, the notion of paths is slightly altered by collapsing inner loops. In the example given in Fig. 5, the outer loop has only one path $\pi = \langle 1, L, 12, 13\rangle$ where L denotes the inner loop consisting of two paths. In general, a path π_i is devided into several path fragments $\pi^1_{i,j}, \ldots, \pi^n_{i,j}$ to separate inner loops. Further, for each $\pi^m_{i,j}$ the preprocessing step derives $k_{i,j,m}$ transfer functions $\langle \mathsf{pre}(\pi^m_{i,j}), \varphi^m_{i,j}\rangle$, one for each fragment, and the nodes representing invariants are augmented with a program counter p of the last statement in the sequence. From each node, only transfer functions are applicable that start in p. For the sake of simplicity, we assume in the following that an outer loop path consists of two path fragments separated by one inner loop.

5.1 Bottom-up Refinement

Here, the inner loop is refined first using the initial invariant yielding a graph G_L. For the refinement of the outer loop, let $\langle \mathsf{inv}, \langle \mathsf{pre}(\pi^k_{i,j}), \varphi^k_{i,j}\rangle\rangle$ be the element just removed from the work list in step 2. Assume that inv holds for program location p, and $\langle \mathsf{pre}(\pi^k_{i,j}), \varphi^k_{i,j}\rangle$ is a transfer function for a path through the inner loop.

If $\mathsf{inv} \sqcap \mathsf{pre}(\pi^k_{i,j}) \neq \emptyset$, apply the fragment update $\varphi^k_{i,j}$ and denote the result by $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi^k_{i,j}))$. Next, select a replacement for $\mathsf{post}(\mathsf{inv}, \mathsf{pre}(\pi^k_{i,j}))$ from G_L, following one of the strategies described in Sect. 4.2.1. All terminal states in G_L that are reachable from the replacement node in G_L are suitable candidates. Then, proceed with the refinement from the chosen terminal node.

Hence, the bottom-up strategy refines each loop only once, but can only rely on weak initial invariants.

5.2 Top-down Refinement

In order to enhance precision, top-down refinement uses the same principle of path fragmentations, but instead of refining the inner loop with an initial invariant, top-down refinement does not choose replacement candidates among terminal nodes of an initially created graph for the inner loop. This approach avoids the pessimistic over-approximation that takes place during bottom-up refinement when selecting a suitable replacement node, and thus, derives the best results the refinement algorithm can compute (with respect to the given abstract domains).

5.3 Comparison

To illustrate the differences in expressiveness, consider the intricate program in Fig. 5. The code fragment calculates Hamming code control bits for a bit array `a` of length `len`. Note that indexing starts at 1 and all bits with an index that is a power of 2 are control bits. In case the last bit of the array is a control bit, the value would always be 0 and is, therefore, not calculated.

When the bottom-up strategy is applied, no upper bound for `y` in the inital invariants of inner loop can be derived. Therefore, an interval analyzer operating on the CFG refined using the bottom-up strategy will detect an out-of-bounds array access in `a[y]`. On the other hand, the validity of the access is determined when using a top-down refinement.

6 Experiments

We have implemented the ideas described in this paper in the [MC]SQUARE verification tool [30, 31], where SAT4J [23] is used as the SAT solving back-end.

The runtime requirements for deriving a transfer function for the example program in Fig. 1 are negligible. For the program in Fig. 3, deriving the six different transfer functions for the two paths required 0.8s including bit-blasting and CNF conversion [22, 27]. Since the [MC]SQUARE analyzes the compiled binary, the analyzed fragment consists of 23 instructions, and the `abs` function is implemented using bitwise operations to avoid conditional branching.

Observe that this is significantly less than the runtime for bit-wise linear congruences [8, 21], mainly due to two reasons: (1) The worst-case chain length for affine equalities is linear, whereas it is polynomial for congruences. (2) The join applied during transfer function synthesis is cubic in the number of variables, and thus, susceptible to the number of variables used. Handling words instead of bits significantly reduces the complexity.

Applying the refinement process then required approximately 0.1 seconds. When a non-relational interval analysis [9] is executed on the refined CFG, the runtime naturally increases (by a factor of 3) to 0.15 seconds for the program in Fig. 3. Whereas the interval analysis operating directly on the CFG from the Hamming code calculation (cp. Fig. 5) can only derive the invariant $\texttt{x} < \texttt{len}$, but no restriction on `y`, the invariant from the refined CFG also states that $\texttt{y} \leq \texttt{len}$, proving correctness of the array-access. Existing approaches not reasoning about programs at this granularity cannot verify this property. Deriving transfer functions required 0.9 seconds, and the refinement process was finished after 0.25 seconds, where most of the runtime is spent on detecting fixed points.

7 Related Work

The problem of analyzing path-specific program properties using abstract interpretation has led to the development of numerous approaches and techniques such as trace partitioning [24], Boolean partitioning [33], disjunctive completion [14, 16, 29], and path-sensitive analysis [6, 18]. On the one hand, techniques based on disjunctive domains typically do not scale to large programs, although recent work by Gurfinkel and Chaki [17] aims at alleviating this problem by representing disjunctive completions of intervals using BDD-like structures. On the other hand, partitioning techniques heavily rely on heuristics to choose split-points. Path-sensitive analyses are typically not well-suited for loops.

Instead of strengthening the power — and thus, the complexity — of abstract domains to integrate path-related information, we follow the approach proposed in [5]. The key idea of their technique is to refine the CFG of a program in order to increase the precision of abstract interpreters. However, the details of their abstract interpreter are not discussed and it is not known if their framework is applicable to bit-manipulating code.

In contrast, our work is tailored towards a specific combination of abstract domains, namely octagons [25, 2, 3] and affine equalities [19]. To derive invariants, we utilize transfer functions using Boolean logic [7]. This technique, which is essentially an adaptation of [26] to bounded integers, allows us to derive updates for blocks of non-affine operations. Through the application of *guarded updates*, this method handles wrap-around arithmetic using domains whose expressiveness does not suffice for representing modular arithmetic on their own.

The refinement itself heavily depends on linear programming and satisfiability tests, which are specifically tailored to octagon constraints. For instance, satisfiability of a system of octagonal constraints is performed by computing its tight closure [25, 2, 3]. Analyzing loops using octagons usually requires widenings [15] to guarantee termination. In our case, the sum of two variables is bounded by twice the width of the domain.

8 Conclusion

Discussion In this paper we have shown how to refine the control structure of loops over bounded integers, based on pre- and postconditions that are expressed as octagon constraints. Computing postconditions is based on applying affine updates to octagonal inputs. This technique strengthens the invariants computed by abstract interpreters and also allows to detect termination bugs stemming from rather intricate cornercases.

Future Work Interestingly, the octagons obtained through SAT are tightly closed [25]. However, this also implies that they contain many redundant inequalities, which can negatively influence the runtime. Thus, it would be interesting to see if either a reduced representation [3] can be obtained in advance, or how reducing constraints prior to the analysis affects the runtime.

Another issue that calls for future work is the handling of nested loops: Affine transfer functions for inner loops could be compiled from least inductive invariants [26]. Then, modeling the effects of inner loops amounts to applying a single transfer function, which could significantly improve the performance.

Further, in [MC]SQUARE, several abstraction techniques based on abstract interpretation are applied during model checking, which have to preserve divergence properties, e.g., a partial-order reduction method [32] for interrupt-driven software. A drawback is that the execution of interrupt handlers must not be moved across the boundaries of loops since their termination is not guaranteed. Slicing [35] suffers from similar problems. Consequently, integrating the refinement technique and the corresponding termination results appears promising.

Acknowledgment

The work of Jörg Brauer and Stefan Kowalewski was, in part, supported by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89. The work of Volker Kamin was, in part, funded by the European FP7 project MULTIFORM, contract number INFSO-ICT-224249.

References

[1] BAGNARA, R., HILL, P., AND ZAFFANELLA, E. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program. 72*, 1-2 (2008), 3–21.

[2] BAGNARA, R., HILL, P. M., AND ZAFFANELLA, E. An improved tight closure algorithm for integer octagonal constraints. In *VMCAI* (2008), vol. 4905 of *LNCS*, Springer, pp. 8–21.

[3] BAGNARA, R., HILL, P. M., AND ZAFFANELLA, E. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design 35*, 3 (2009), 279–323.

[4] BALAKRISHNAN, G., AND REPS, T. W. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* (2010). To appear.

[5] BALAKRISHNAN, G., SANKARANARAYANAN, S., IVANCIC, F., AND GUPTA, A. Refining the control structure of loops using static analysis. In *EMSOFT* (2009), ACM Press, pp. 49–58.

[6] BALAKRISHNAN, G., SANKARANARAYANAN, S., IVANCIC, F., WEI, O., AND GUPTA, A. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS* (2008), vol. 5079 of *LNCS*, Springer, pp. 238–254.

[7] BRAUER, J., AND KING, A. Automatic abstraction for intervals using boolean formulae. In *SAS* (2010). To appear.

[8] BRAUER, J., KING, A., AND KOWALEWSKI, S. Range analysis of microcontroller code using bit-level congruences. In *FMICS* (2010), LNCS, Springer. To appear.

[9] BRAUER, J., NOLL, T., AND SCHLICH, B. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *SCOPES* (2010), ACM. To appear.

[10] CHVÁTAL, V. *Linear Programming*. W. H. Freeman and Company, 1983.

[11] CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design 19*, 1 (2001), 7–34.

[12] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *TACAS* (2004), vol. 2988 of *LNCS*, Springer, pp. 168–176.

[13] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL* (1977), ACM, pp. 238–252.

[14] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *POPL* (1979), ACM, pp. 269–282.

[15] COUSOT, P., AND COUSOT, R. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP* (1992), vol. 631 of *LNCS*, Springer, pp. 269–295.

[16] GIACOBAZZI, R., AND RANZATO, F. Optimal domains for disjunctive completion. *Science of Computer Programming 32*, 1–3 (1998), 177–210. 6th European Symposium on Programming.

[17] GURFINKEL, A., AND CHAKI, S. BOXES: A symbolic abstract domain of boxes. In *SAS* (2010). To appear.

[18] HARRIS, W. R., SANKARANARAYANAN, S., IVANCIC, F., AND GUPTA, A. Program analysis via satisfiability modulo path programs. In *POPL* (2010), ACM Press, pp. 71–82.

[19] KARR, M. Affine relationships among variables of a program. *Acta Informatica 6* (1976), 133–151.

[20] KING, A., AND SØNDERGAARD, H. Inferring congruence equations using SAT. In *CAV* (2008), vol. 5123 of *LNCS*, Springer, pp. 281–293.

[21] KING, A., AND SØNDERGAARD, H. Automatic abstraction for congruences. In *VMCAI* (2010), vol. 5944 of *LNCS*, Springer, pp. 281–293.

[22] KROENING, D., AND STRICHMAN, O. *Decision Procedures*. Springer, 2008.

[23] LE BERRE, D. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. `http://www.sat4j.org/`.

[24] MAUBORGNE, L., AND RIVAL, X. Trace partitioning in abstract interpretation based static analyzers. In *ESOP* (2005), vol. 3444 of *LNCS*, Springer, pp. 5–20.

[25] MINÉ, A. The octagon abstract domain. *Higher-Order and Symbolic Computation 19*, 1 (2006), 31–100.

[26] MONNIAUX, D. Automatic Modular Abstractions for Linear Constraints. In *POPL* (2009), ACM Press, pp. 140–151.

[27] PLAISTED, D. A., AND GREENBAUM, S. A structure-preserving clause form translation. *Journal of Symbolic Computation 2*, 3 (September 1986), 293–304.

[28] REPS, T., SAGIV, M., AND YORSH, G. Symbolic implementation of the best transformer. In *VMCAI* (2004), vol. 2937 of *LNCS*, Springer, pp. 252–266.

[29] SANKARANARAYANAN, S., IVANCIC, F., SHLYAKHTER, I., AND GUPTA, A. Static analysis in disjunctive numerical domains. In *SAS* (2006), vol. 4134 of *LNCS*, Springer, pp. 3–17.

[30] SCHLICH, B. *Model Checking of Software for Microcontrollers*. Dissertation, RWTH Aachen University, Germany, June 2008.

[31] SCHLICH, B., BRAUER, J., AND KOWALEWSKI, S. Application of static analyses for state space reduction to microcontroller binary code. *Sci. Comput. Program.* (2010). To appear.

[32] SCHLICH, B., NOLL, T., BRAUER, J., AND BRUTSCHY, L. Reduction of interrupt handler executions for model checking embedded software. In *HVC* (2009), LNCS, Springer. To appear.

[33] SIMON, A. Splitting the control flow with boolean flags. In *SAS* (2008), vol. 5079 of *LNCS*, Springer, pp. 315–331.

[34] WAYNE, K. A polynomial combinatorial algorithm for generalized minimum cost flow. In *Theory of Computing* (1999), ACM, pp. 11–18.

[35] WEISER, M. Program slicing. In *Software engineering (ICSE 81)* (1981), IEEE Press, pp. 439–449.

[36] XIE, Y., AND AIKEN, A. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst. 29*, 3 (2007), 1–43.