Fully Bounded Polyhedral Analysis of Integers with Wrapping

Stefan Bygde, Björn Lisper [1]

School of Innovation, Design and Technology
Mälardalen University
Västerås, Sweden

Niklas Holsti [2]

Tidorum Ltd
Helsinki, Finland

Abstract

Analysis of convex polyhedra using abstract interpretation is a common and powerful program analysis technique to discover linear relationships among variables in a program. However, the classical way of performing polyhedral analysis does not model the fact that values typically are stored as fixed-size binary strings and usually have a wrap-around semantics in the case of overflows. In embedded systems where 16-bit or even 8-bit processors are used, wrapping behaviour may even be used intentionally. Thus, to accurately and correctly analyse such systems, the wrapping has to be modelled.
We present an approach to polyhedral analysis which derives polyhedra that are bounded in all dimensions and thus provides polyhedra that contain a finite number of integer points. Our approach uses a previously suggested wrapping technique for polyhedra but combines it in a novel way with limited widening, a suitable placement of widening points and restrictions on unbounded variables. We show how our method has the potential to significantly increase the precision compared to the previously suggested wrapping method.

Keywords: Abstract Interpretation, Abstract Domains, Numerical Domains, Convex Polyhedra, Widening, Overflows

1 Introduction

A general and commonly used application of program analysis is to derive which numerical values the program variables can take at each point in the

[1] Email: {stefan.bygde,bjorn.lisper}@mdh.se
[2] Email: niklas.holsti@tidorum.fi

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

program. This is typically done using abstract interpretation [3] and some numerical abstract domain to get an approximation of the possible values. Many relational and non-relational domains have been developed [3,5,6,11,4], all with the common assumption that variables can take arbitrary integer values. However, in real programs, a value is usually stored as a fixed-size binary string. A common source of subtle bugs is overflows, meaning that the result of a computation is too large to be stored in a binary string of the given size. An overflow could result in a run-time error, saturation of the result at the largest or smallest representable value of the integer type, or a wrap-around. In general purpose processors, wrap-around is the most common approach to handle overflows. Although wrap-arounds may be the reason for some bugs, it is not uncommon that wrap-arounds are used intentionally, in particular on processors with a short word-length. Unfortunately, under the mentioned assumption about variables, most abstract numerical domains derive unsound results if wrap-arounds are present in a program.

To make analysis results sound w.r.t. wrap-arounds it is necessary to make modifications to the selected domain. Sen and Srikant [13] present a variation of the reduced product of the integer and congruence domain which handles special cases when overflow occurs. In addition they use a relational analysis of affine equalities. Gustafsson et al. [7] modify the interval domain so that variables are bounded to within their range, and wrap-arounds are handled by using a more powerful representation of intervals. Relational domains are more challenging. Müller-Olm and Seidl [12] present an analysis that can derive all affine equalities among variables of programs which is safe in the case of wrap-arounds. Brauer and King [1] suggest a method to derive transfer functions for relational domains, and do so for the octagon domain, while considering wrap-around effects by using a SAT-solver. Finally, Simon and King [14] present a way to use polyhedral analysis (originally presented in [4]) in such a way that it is sound even when wrap-arounds are present.

Our work builds on Simon and King's approach to use the classical polyhedral analysis. We show in this paper that just using Simon and King's approach directly can lead to unnecessary loss of precision, in particular loss of relational information.

We present a polyhedral analysis which derives fully bounded polyhedra that are sound in the presence of wrap-arounds in the program. Our approach is based on a combination of wrapping polyhedra (using the approach in [14]), limited widening [9], an appropriate placement of the widening, and imposing bounds on variables based on type information. The benefits of our approach are the following:

- Increased precision compared to using the approach outlined in [14] with standard abstract interpretation.

- Bounded polyhedra are particularly useful for analyses that depend on the number of integer points inside a polyhedron (e.g., [10], [2]), since a bounded polyhedron guarantees that this number is finite.

Section 2 contains preliminaries to our approach, explaining classical polyhedral analysis and wrapping of polyhedra. We show in Section 3 a motivational example of our method and how it differs from other methods. In Section 4 we detail our approach to a bounded polyhedral analysis. Section 5 discusses our approach to widening, which is the core of the method. Finally, we conclude in Section 6.

2 Preliminaries

2.1 The Polyhedral Domain

The classical abstract domain of convex polyhedra [4] abstracts a finite set of integer points $S \subseteq \mathbb{R}^n$ by $\alpha(S)$, the smallest convex polyhedron enclosing all integer points in S. An efficient implementation of convex polyhedra needs a dual representation. One representation is a set of linear constraints C; the polyhedron $\mathcal{P}(C)$ then consist of all points in $\mathbb{R}^n$ fulfilling the constraints in C. The other representation is a *frame* F, which is a tuple $\langle V, R\rangle$ of vertices $V = \{\mathbf{v}_0, ..., \mathbf{v}_{v-1}\} \subseteq \mathbb{R}^n$, and rays $R = \{\mathbf{r}_0, ..., \mathbf{r}_{r-1}\} \subseteq \mathbb{R}^n$. The polyhedron $\mathcal{P}(F)$ of a frame represents the points in $\mathbb{R}^n$ which are a convex combination of the vertices plus a linear combination of the rays (allowing unbounded polyhedra). Note that we use $\mathcal{P}$ to distinguish a polyhedron from its representation.

In this paper we will model integer-valued variables, hence we will be interested in the integer points inside a polyhedron. Thus, if P is a polyhedron then $\gamma(P) \subseteq \mathbb{Z}^n$ (that is, the concretisation function) will denote the integer points inside P.

2.2 Finite Integer Variables

To correctly model variables which may wrap around, we need to be more specific about how we model variables. Let $X = \{x_0, ..., x_{n-1}\}$ be the set of program variables. Each variable x_j is associated with a size w_j defining how many bits are used to store the variable (we allow variables to have different sizes) and a type, signed (**int**) or unsigned (**uint**). We define the *range* of a variable x as a function returning a set of constraints.

$$\text{range}(x_j) = \begin{cases} \{x_j \geq 0, x_j \leq 2^{w_j} - 1\} & \text{if } x_j \text{ is } \textbf{uint} \\ \{x_j \geq -2^{w_j-1}, x_j \leq 2^{w_j-1} - 1\} & \text{if } x_j \text{ is } \textbf{int} \end{cases}$$

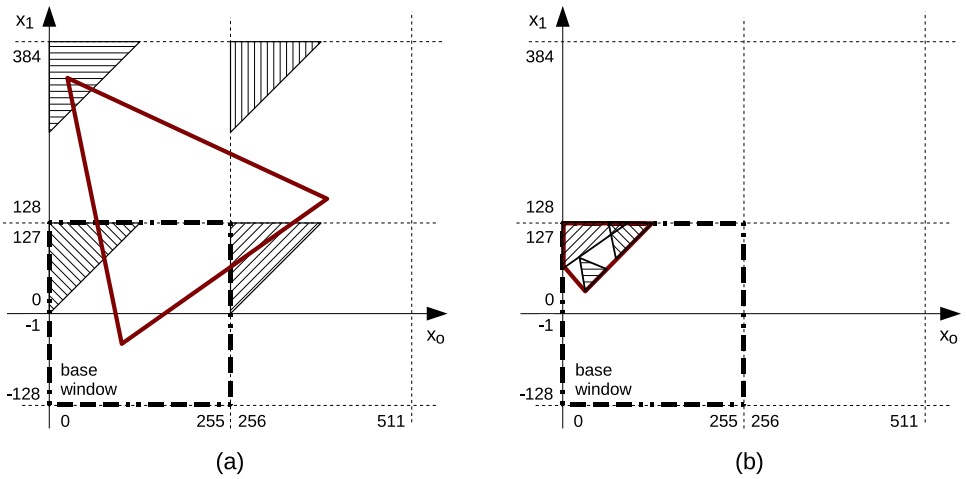

Fig. 1. The picture on the left (a), shows a polyhedron before wrapping. The base window is shown outlined by a dot-dashed square. The polyhedron covers a part of the base window and parts of the three neighbouring windows. The grid of variously hatched triangles shows the condition $x_0 \leq x_1$ taken as a signed comparison of the 8-bit unsigned residue of x_0 with the 8-bit signed residue of x_1. The polyhedron intersects three components of this condition, one in the base window, one in the next window to the right of the base window, and one in the next window above the base window. To the right (b), the intersections of the condition with the unwrapped polyhedron are shown, shifted to the base window, and their convex hull, which is the resulting wrapped polyhedron.

We then define the set of *range constraints* $\mathcal{R}_V$ for a set of variables $V \subseteq X$ as

$$\mathcal{R}_V = \bigcup_{x \in V} \text{range}(x)$$

If V is the set of all program variables X we simply write $\mathcal{R}$ for $\mathcal{R}_X$. The set of range constraints forms a polyhedron $\mathcal{P}(\mathcal{R})$.

2.3 Wrap-arounds

Simon and King [14] have developed a method to make polyhedral analysis sound for finite-sized integers with wrap-around behaviour. Since our method builds on this method, the basics are presented here.

Define the *base window* B as the set of integer points in $\mathcal{P}(\mathcal{R})$, the range constraint polyhedron. Let $M : \mathbb{Z}^n \to B$ be defined as $\langle p_0, p_1, ..., p_{n-1} \rangle \mapsto (p_0 \mod 2^{w_0}, p_1 \mod 2^{w_1}, ..., p_{n-1} \mod 2^{w_{n-1}})$ where the *mod* residue is taken as signed or unsigned depending on the type of the variable x_j. Simon and King's concretisation function is then defined as $\gamma_{\text{SK}} = M \circ \gamma$. This means that $\gamma_{\text{SK}}(P) \subseteq B$ for any polyhedron P.

Intuitively, $\gamma_{\text{SK}}(P)$ can be seen as partitioning $\mathbb{Z}^n$ into a grid of rectangular windows, each of with dimensions 2^{w_j}, taking the intersection of P with each window, and shifting this intersection into the base window by the required multiples of 2^{w_j} in each coordinate x_j. Then, $\gamma_{\text{SK}}(P)$ is the mosaic composed of these shifted "residue fragments" of P, which may overlap each other. Note that the points in $\gamma_{\text{SK}}(P)$ might not form a convex polyhedron if P is not

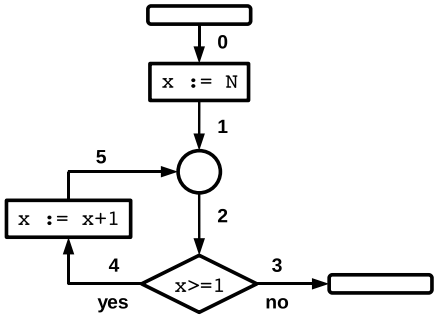

Fig. 2. An example program L.

contained in B.

Addition, subtraction, and multiplication by constants can be handled as usual, because these operations commute with modular residue. Arithmetic inequality comparisons $(<, \leq, \geq, >)$ of finite-sized integers, however, do not commute with modular residue. Therefore, it is necessary to explicitly wrap a polyhedron when a constraint is applied. Let c be a constraint, P be a polyhedron and $V \subseteq X$ be a set of variables. Then, informally, the function $\text{wrap}(P, c, V)$ physically performs the partitioning and shifts previously mentioned, applies c on every fragment and computes the convex hull of the result. However, only the subspace of $\mathbb{Z}^n$ involving the variables in V is partitioned and shifted by $\text{wrap}(P, c, V)$. In typical use, V is the set of variables involved in the linear constraint c. This wrapping is illustrated in Figure 1.

In practice Simon and King compute this approximation of c applied to P only when P has a finite number of residue fragments, in other words only if all the variables involved in the constraint are bounded in P. Otherwise, Simon and King discard all information about these variables in P and substitute the condition c itself as the only constraint on these variables.

3 Motivation and Illustrating Example

We model programs as flow charts, as seen in Figure 2, where we associate each edge q in the flow chart with a convex polyhedron $P(q)$. Using classical polyhedral analysis [4], the program L in Figure 2 gives rise to the following recurrence equation system:

$$\begin{aligned} P_n(0) &= \top & P_n(1) &= P_n(0)[x \to N] \\ P_n(2) &= P_{n-1}(2) \nabla (P_n(1) \sqcup P_{n-1}(5)) & P_n(3) &= P_n(2) \sqcap \{x \leq 0\} \\ P_n(4) &= P_n(2) \sqcap \{x \geq 1\} & P_n(5) &= P_n(4)[x \to x + 1] \end{aligned} \tag{1}$$

Assume that L (in Figure 2) is executed on an 8-bit processor. Let x and N be 8-bit unsigned integer variables. Then L terminates because x wraps around when $255 + 1$ returns 0. Thus, (1) will give unsound results.

This is because the classical polyhedral domain does not model wrap-arounds. To make the result sound, we apply Simon and King's wrapping operator, substituting the equations P_n^{SK} for P_n. The equations P_n^{SK} for edges $0, 1, 2$ and 5 are equal to P_n, but for edges 3 and 4 we have:

$$P_n^{\mathrm{SK}}(3) = \mathrm{wrap}(P_n(2), \{x \leq 0\}, \{x\}) \tag{2}$$
$$P_n^{\mathrm{SK}}(4) = \mathrm{wrap}(P_n(2), \{x \geq 1\}, \{x\}) \tag{3}$$

Iter	$n = 1$	$n = 2$
$P_n^{\mathrm{SK}}(0)$	$\top$	$\top$
$P_n^{\mathrm{SK}}(1)$	$x = N$	$x = N$
$P_n^{\mathrm{SK}}(2)$	$x = N$	$\top$
$P_n^{\mathrm{SK}}(3)$	$x = 0 \wedge 0 \leq N \leq 255$	$x = 0 \wedge 0 \leq N \leq 255$
$P_n^{\mathrm{SK}}(4)$	$1 \leq x \leq 255 \wedge 0 \leq N \leq 255$	$1 \leq x \leq 255 \wedge 0 \leq N \leq 255$
$P_n^{\mathrm{SK}}(5)$	$2 \leq x \leq 256 \wedge 0 \leq N \leq 255$	$2 \leq x \leq 256 \wedge 0 \leq N \leq 255$

Table 1
Iterating abstract interpretation of program L using wrapped polyhedra.

The result of the iterates is shown in Table 1. The polyhedron $P_2^{\mathrm{SK}}(3)$ correctly implies that N has to be in the range 0 to 255. However, during the process, the relational information between x and N has been lost. This is due to the wrapping of the unbounded polyhedron $P_1^{\mathrm{SK}}(2) = \mathcal{P}(\{x = N\})$, which has to discard the relational information between x and N in order to make a sound and safe wrapping. While this is a simple example that could have been avoided by imposing bounds on x and N at the start of the program, unbounded polyhedra are frequent (caused by any widening, non-linear assignment or unbounded initial state) and apparently make the wrapping algorithm lose a lot of precision. This has led us to devise a fully bounded polyhedral analysis. Our method uses limited widening, places the widening points in a suitable way, and uses type information to bind variables. To show the idea we sketch here how our method analyses L. The equations for our method are the following:

$$P_n^{\mathrm{BD}}(0) = \mathcal{P}(\mathcal{R}) \qquad P_n^{\mathrm{BD}}(1) = P_n^{\mathrm{BD}}(0)[x \to N]$$
$$P_n^{\mathrm{BD}}(2) = P_n^{\mathrm{BD}}(1) \sqcup P_{n-1}^{\mathrm{BD}}(5) \qquad P_n^{\mathrm{BD}}(3) = \mathrm{wrap}(P_n^{\mathrm{BD}}(2), \{x \leq 0\}, \{x\})$$
$$P_n^{\mathrm{BD}}(4) = P_{n-1}^{\mathrm{BD}}(4) \; \nabla_{\mathcal{R} \cup \{x \geq 1\}} \; \mathrm{wrap}(P_n^{\mathrm{BD}}(2), \{x \geq 1\}, \{x\})$$
$$P_n^{\mathrm{BD}}(5) = P_n^{\mathrm{BD}}(4)[x \to x + 1]$$

where $\mathcal{R} = \{0 \leq x \leq 255, 0 \leq N \leq 255\}$.

The difference between P_n^{BD} and P_n^{SK} can be seen in three places. First, the initial program point $P_n^{\mathrm{BD}}(0)$ bounds the variables according to their type. Second, the widening point has been moved to $P_n^{\mathrm{BD}}(4)$, and finally, the widen-

ing has been replaced by limited widening (explained in Section 5.1).

All these polyhedra are fully bounded so we can represent their frames with sets of vertices only, no rays are needed. This representation is more convenient for our method. The iterates shown in Table 2 use this representation. Here we let the first dimension correspond to x and the second to N. Note that $\mathcal{R}$ represented as a set of vertices is $\{(0,0),(0,255),(255,0),(255,255)\}$.

In this example, our approach takes a few more iterations before stabilisation, but this result, while still sound w.r.t. wrap-arounds, is more precise than the previous approach. In particular, notice that in $P_4^{\mathrm{BD}}(3)$ we have that x remains zero, but N can be any number between 0 and 255 (soundness), and we have kept the valuable relation between x and N in $P_4^{\mathrm{BD}}(2), P_4^{\mathrm{BD}}(4)$ and $P_4^{\mathrm{BD}}(5)$, as the polyhedra have triangular shapes. This information was not retained in $P_2^{\mathrm{SK}}(2), P_2^{\mathrm{SK}}(4)$ or $P_2^{\mathrm{SK}}(5)$.

Iter	1	2
$P^{\mathrm{BD}}(0)$	$(0,0),(0,255),(255,0),(255,255)$	$(0,0),(0,255),(255,0),(255,255)$
$P^{\mathrm{BD}}(1)$	$(0,0),(255,255)$	$(0,0),(255,255)$
$P^{\mathrm{BD}}(2)$	$(0,0),(255,255)$	$(0,0),(255,255),(2,1),(256,255)$
$P^{\mathrm{BD}}(3)$	$(0,0)$	$(0,0),(0,255)$
$P^{\mathrm{BD}}(4)$	$(1,1),(255,255)$	$(1,1),(255,255),(255,1)$
$P^{\mathrm{BD}}(5)$	$(2,1),(256,255)$	$(2,1),(256,255),(256,1)$
Iter	3	4
$P^{\mathrm{BD}}(0)$	$(0,0),(0,255),(255,0),(255,255)$	$(0,0),(0,255),(255,0),(255,255)$
$P^{\mathrm{BD}}(1)$	$(0,0),(255,255)$	$(0,0),(255,255)$
$P^{\mathrm{BD}}(2)$	$(0,0),(255,255),(256,255),(256,1)$	$(0,0),(255,255),(256,255),(256,0)$
$P^{\mathrm{BD}}(3)$	$(0,0),(0,255)$	$(0,0),(0,255)$
$P^{\mathrm{BD}}(4)$	$(1,0),(1,1),(255,255),(255,0)$	$(1,0),(1,1),(255,255),(255,0)$
$P^{\mathrm{BD}}(5)$	$(2,0),(2,1),(256,255),(256,0)$	$(2,0),(2,1),(256,255),(256,0)$

Table 2
Iterating abstract interpretation for L using fully bounded polyhedra. The iterates are shown as sets of vertices rather than constraints.

4 Bounded Polyhedral Analysis

In classical analysis, a polyhedron may become unbounded in three cases: First, in the initial program point, where nothing is known about the program variables. Second, any non-linear assignment drops any information about a variable, leaving the polyhedron unbounded in the direction of that variable. Third, widening often produces an unbounded polyhedron.

Bounded polyhedra have several benefits. First, the wrapping algorithm loses all relational information between variables which are unbounded. If a polyhedron is bounded, this information is not necessarily lost. Second, analyses that count the number of integer points in polyhedra (such as [10], [2]) greatly benefit from having this number finite.

4.1 Making Polyhedra Bounded

We consider each of the possible ways of making a polyhedron unbounded and argue how it is possible to soundly and precisely make it bounded. This section details how the equations P_n^{SK} are replaced by P_n^{BD}.

4.2 Entry point

The classical polyhedral domain uses the unbounded polyhedron as starting point to denote that nothing is known about the variable values. That is, if p_{init} is the initial program point, the initial equation for the classical polyhedral domain using wrapping is $P_n^{\mathrm{SK}}(p_{\mathrm{init}}) = \top$. However, since each variable is associated with a type, it is possible to bind them. That is, at program start, the constraints $\mathcal{R}$ all hold. Thus, we define

$$P_n^{\mathrm{BD}}(p_{\mathrm{init}}) = \mathcal{P}(\mathcal{R})$$

Note that $P_n^{\mathrm{BD}}(p_{\mathrm{init}})$ is sound and more precise than $P_n^{\mathrm{SK}}(p_{\mathrm{init}})$.

4.3 Non-Linear Assignments

In polyhedral analysis, a non-linear assignment discards all information about the assigned variable as well as its relation to other variables. As an example let $p_{\mathbf{x}:=?}$ be an edge immediately succeeding a non-linear assignment. Then, $P_n^{\mathrm{SK}}(p_{\mathbf{x}:=?}) = \pi_{\mathrm{x}}(P_n(\mathrm{prev}(p_{\mathbf{x}:=?})))$, where π_{x} is the projection operation in x, adding a line to the frame in the direction of x. The function $\mathrm{prev}(p)$ returns the edges that enter the node that p leaves. In the case of an assignment, it is guaranteed to be just one, so we slightly abuse notation in this case to refer to the single element of the return from the prev function. After the non-linear assignment we can, since relational information pertaining to x has been discarded, claim that $\mathrm{range}(x)$ is true. Thus, we define:

$$P_n^{\mathrm{BD}}(p_{\mathbf{x}:=?}) = \pi_{\mathrm{x}}(P_n^{\mathrm{BD}}(\mathrm{prev}(p_{\mathbf{x}:=?}))) \sqcap \mathcal{P}(\mathrm{range}(x))$$

Again, $P_n^{\mathrm{BD}}(p_{\mathbf{x}:=?})$ is sound and more precise than $P_n^{\mathrm{SK}}(p_{\mathbf{x}:=?})$.

4.4 Widening

For a program containing cycles, widening is necessary to ensure termination. In classical abstract interpretation, the widening is usually inserted immediately after the loop merge points. Let p_{loop} be a program point immediately succeeding a loop merge node (for example edge 2 for program L). Then, the

classical polyhedral analysis defines:

$$P_n(p_{\text{loop}}) = P_{n-1}(p_{\text{loop}}) \nabla \bigsqcup_{q \in \text{prev}(p_{\text{loop}})} P_{n-1}(q)$$

This commonly results in an unbounded polyhedron, since widening often removes constraints. It would not be sound with respect to wrap-arounds to apply any range constraints in this case, so we have to take another approach.

5 Making Widening Bounded

The standard widening operation, as mentioned, often makes polyhedra unbounded. However, with the help of *limited widening* it might be possible to intersect the result with some finite constraints. Our idea is to use widening in such a way that it is always possible to intersect the result with a fully bounded polyhedron.

5.1 Limited Widening

Limited widening was suggested in [9]. The idea with limited widening is to have a set of constraints C and define limited widening ∇_C as follows:

$$P \nabla_C Q = (P \nabla Q) \sqcap \{c \in C | P \sqsubseteq \mathcal{P}(\{c\}) \wedge Q \sqsubseteq \mathcal{P}(\{c\})\}$$

That is, the result of the widening is intersected with all constraints in C which hold in both P and Q. It can be shown that this is a widening operation for any set of constraints C. The set C is typically selected strategically for each program.

Our idea is to use the range constraints $\mathcal{R}$ of a program as the set C in limited widening. Our goal is to be able to intersect with *all* range constraints, to make the polyhedron fully bounded. To avoid wrapping variables more than necessary, the widening point should not be put at the loop merge point.

5.2 Placement of the Widening Point

In the classical polyhedral analysis it is common to place the widening point immediately after the loop-merge node. However, doing this without wrapping often results in an unbounded polyhedron (see Table 1). Our goal is to intersect the result with the range constraints and to reduce unnecessary wrappings. To this end, we place the widening point at the conditionals where wrapping must be done anyway. We do this in a way so that we still have

exactly one widening per cycle in the flow chart. This means that we replace

$$P_n^{\mathrm{SK}}(p_{\mathrm{loop}}) = P_{n-1}^{\mathrm{SK}}(p_{\mathrm{loop}}) \nabla \bigsqcup_{q \in \mathrm{prev}(p_{\mathrm{loop}})} P_{n-1}(q)$$

with

$$P_n^{\mathrm{BD}}(p_{\mathrm{loop}}) = \bigsqcup_{q \in \mathrm{prev}(p_{\mathrm{loop}})} P_{n-1}^{\mathrm{BD}}(q)$$

This is possible since we will be putting a widening point elsewhere in the cycle. Let σ be a linear inequality constraint involving the variables X_σ and let p_σ be the edge immediately succeeding a conditional within a cycle [3]. When using Simon and King's wrapping, $P_n(p_\sigma)$ is defined as

$$P_n^{\mathrm{SK}}(p_\sigma) = \mathrm{wrap}(P_n^{\mathrm{SK}}(\mathrm{prev}(p_\sigma)), \{\sigma\}, X_\sigma)$$

where X_σ are the variables involved in σ. We replace this with:

$$P_n^{\mathrm{BD}}(p_\sigma) = P_{n-1}^{\mathrm{BD}}(p_\sigma) \ \nabla_{\mathcal{R} \cup \{\sigma\}} \ \mathrm{wrap}(P_n^{\mathrm{BD}}(\mathrm{prev}(p_\sigma)), \{\sigma\}, X) \qquad (4)$$

at one conditional in every cycle. Note that we use X, the set of all program variables, rather than X_σ (however, this can be improved, see discussion below). Placing the widening at the conditional avoids unnecessary wrapping but does not affect the soundness of the method. We put the widening in conjunction with the wrapping, and we use a limited widening with the range constraints of the program variables $\mathcal{R}$ and the conditional itself σ. Since limited widening is a widening, and since we have a widening in every cycle, this is a sound and safe thing to do, and it still guarantees termination. Moreover, this always results in a fully bounded polyhedron, as shown by the following proposition.

Proposition 5.1 *Let $P_0^{\mathrm{BD}}(q) = \bot$ for all edges q in a program. Let σ be a linear constraint, let $C = \mathcal{R} \cup \{\sigma\}$, and let $P_n^{\mathrm{BD}}(p_\sigma)$ be defined by (4). Then:*

$$P_n^{\mathrm{BD}}(p_\sigma) \sqsubseteq \mathcal{P}(C)$$

for all $n > 0$. Moreover, $P_n^{\mathrm{BD}}(p_\sigma)$ is a fully bounded polyhedron since $\mathcal{P}(C)$ is.

Proof. First, let $Q_n = P_{n-1}^{\mathrm{BD}}(p_\sigma)$ and $R_n = \mathrm{wrap}(P_n^{\mathrm{BD}}(\mathrm{prev}(p_\sigma)), \{\sigma\}, X)$, so we have

$$P_n^{\mathrm{BD}}(p_\sigma) = Q_n \ \nabla_C \ R_n$$

We will prove by induction over n that the proposition holds. Let $n = 1$, then $Q_1 = P_0^{\mathrm{BD}}(p_\sigma) = \bot$. The wrapping operator guarantees that all variables X

[3] Note that σ can either be the conditional corresponding to the false-branch or the true-branch depending on the form of the loop; if it is the false-branch σ is simply negated.

are within their respective range. Thus, $\forall c \in \mathcal{R} : R_n \sqsubseteq \mathcal{P}(\{c\})$ for all $n > 0$. Also we have that $R_n \sqsubseteq \mathcal{P}(\{\sigma\})$, since the wrapping operation applies the condition after wrapping. Now we have

$$\begin{gathered} Q_1 \nabla_C R_1 = \bot \nabla_C R_1 \\ = \bot \nabla R_1 \sqcap \mathcal{P}(\{c \in C | \bot \sqsubseteq \mathcal{P}(\{c\}) \wedge R_1 \sqsubseteq \mathcal{P}(\{c\})\}) \\ = R_1 \sqcap \mathcal{P}(\{c \in C | \bot \sqsubseteq \mathcal{P}(\{c\}) \wedge R_1 \sqsubseteq \mathcal{P}(\{c\})\}) \\ = R_1 \sqcap \mathcal{P}(C) \sqsubseteq \mathcal{P}(C) \end{gathered}$$

where the last equation comes from the fact that $\bot \sqsubseteq \mathcal{P}(\{c\})$ for any c and $R_n \sqsubseteq \mathcal{P}(\{c\})$ has already been established for any $c \in C$. Thus, the proposition holds for $n = 1$. Now assume that $Q_n \nabla_C R_n \sqsubseteq \mathcal{P}(C)$ holds, then

$$Q_{n+1} \nabla_C R_{n+1} = (Q_{n+1} \nabla R_{n+1}) \sqcap \{c \in C | Q_n \sqsubseteq \mathcal{P}(\{c\}) \wedge R_n \sqsubseteq \mathcal{P}(\{c\})\}$$

The inductive hypothesis says that $Q_n \sqsubseteq \mathcal{P}(\{c\})$ and $R_n \sqsubseteq \mathcal{P}(\{c\})$ for all $c \in C$, so again we have that $\{c \in C | Q_n \sqsubseteq \mathcal{P}(\{c\}) \wedge R_n \sqsubseteq \mathcal{P}(\{c\})\} = C$. So,

$$\begin{gathered} (Q_{n+1} \nabla R_{n+1}) \sqcap \{c \in C | Q_n \sqsubseteq \mathcal{P}(\{c\}) \wedge R_n \sqsubseteq \mathcal{P}(\{c\})\} \\ = (Q_{n+1} \nabla R_{n+1}) \sqcap \mathcal{P}(C) \sqsubseteq \mathcal{P}(C) \end{gathered}$$

□

Proposition 5.1 proves that all variables are bounded after widening. This together with the previous steps to make sure a polyhedron is bounded results in an analysis where each polyhedron is bounded. On a final note, it is possible to improve the set X in Proposition 5.1, by observing that only the variables involved in constraints that are removed by the widening operator need to be wrapped, since they are the only ones being affected by the widening. However, we used X as the set of all variables to simplify the proof.

6 Conclusions and Future Work

We have developed an analysis using fully bounded convex polyhedra which is sound for programs with wrap-around semantics. This is done by imposing range bounds on variables at the initial program point and at non-linear assignments, wrapping polyhedra at conditionals (as in [14]) and finally by using limited widening with range constraints and placing this widening at conditionals.

We believe that this analysis is likely to be more precise than using Simon and King's approach on standard abstract interpretation. This is because their approach has to discard any relational information among unbounded variables, whereas our method never leaves any variable unbounded. Note that

we have not specified at which conditional in a cycle the widening point should be placed. We plan to develop heuristics for placing the widening points, although we expect that optimal placement depends on the program. Our method is being implemented in the SWEET tool [8] that performs control-flow analysis for bounding the worst-case execution time of embedded, real-time programs. We plan to evaluate the method on programs with and without wrap-arounds.

Acknowledgement. This work was supported by the EU FP7 project APARTS, Grant Number 251413.

References

[1] Brauer, J. and A. King, *Transfer function synthesis without quantifier elimination*, in: G. Barthe, editor, *ESOP*, Lecture Notes in Computer Science **6602** (2011), pp. 97–115.

[2] Bygde, S., *Static WCET analysis based on abstract interpretation and counting of elements*, Licentiate thesis (2010).
URL http://www.mrtc.mdh.se/index.php?choice=publications\&id=2144

[3] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *POPL*, 1977, pp. 238–252.

[4] Cousot, P. and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, in: *POPL*, 1978, pp. 84–96.

[5] Granger, P., *Static analysis of arithmetical congruences*, in: *International Journal of Computer Mathematics, Volume 30*, 1989, pp. 165–190.

[6] Granger, P., *Static analysis of linear congruence equalities among variables of a program*, in: *Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1* (1991), pp. 169–192.
URL http://portal.acm.org/citation.cfm?id=111310.111320

[7] Gustafsson, J., A. Ermedahl and B. Lisper, *Towards a flow analysis for embedded system C programs*, in: *The 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS05)*, 2005.
URL http://www.mrtc.mdh.se/index.php?choice=publications&id=0972

[8] Gustafsson, J., A. Ermedahl, C. Sandberg and B. Lisper, *Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution*, in: *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, 2006.

[9] Halbwachs, N., *Delay analysis in synchronous programs*, in: C. Courcoubetis, editor, *CAV*, Lecture Notes in Computer Science **697** (1993), pp. 333–346.

[10] Lisper, B., *Fully automatic, parametric worst-case execution time analysis*, in: J. Gustafsson, editor, *Proc. Third International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2003, pp. 77–80.
URL http://www.mrtc.mdh.se/index.php?choice=publications\&id=0629

[11] Miné, A., *The octagon abstract domain*, Higher Order Symbol. Comput. **19** (2006), pp. 31–100.

[12] Müller-Olm, M. and H. Seidl, *Analysis of modular arithmetic*, ACM Trans. Program. Lang. Syst. **29** (2007).
URL http://doi.acm.org/10.1145/1275497.1275504

[13] Sen, R. and Y. N. Srikant, *Executable analysis using abstract interpretation with circular linear progressions*, in: *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07 (2007), pp. 39–48.
URL http://dx.doi.org/10.1109/MEMCOD.2007.371251

[14] Simon, A. and A. King, *Taming the wrapping of integer arithmetic*, in: *Static Analysis*, Lecture Notes in Computer Science **4634**, Springer Berlin / Heidelberg, 2007 pp. 121–136.