Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 289 (2012) 41–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Normalisation of Loops with Covariant Variables

Marianne De Michiel[1] Armelle Bonenfant[1] Hugues Cassé[1]

IRIT
Université de Toulouse
Toulouse, France

Abstract

Temporal property verification is utterly important to ensure safety of critical real-time systems. A main component of this verification is the computation of Worst Case Execution Time (WCET) that requires, in turn, the determination of loop bounds. Although a lot of efforts have been performed in this domain, it remains relatively common cases which are unsolved. For example, to our knowledge, no fast automatic method can cope with the loop bound of a simple binary search look-up. In this paper, we present an approach to solve such loops by using arithmetico-geometric series, that is, loops with arithmetic and/or geometric incrementation with several variables. We have implemented and experimented this approach in our tool oRange.

Keywords: Timing analysis, worst-case execution time, loop bound analysis, abstract interpretation

1 Introduction

Critical hard real-time systems are composed of tasks which must imperatively finish before their deadline. In order to guarantee this, a task scheduling analysis, which requires the knowledge of the WCET of each task, is performed.

The static WCET analysis is performed by a timing analysis tool which needs loop upper bounds. Such bounds may be given by the developer with manual annotations in the programs or by automatic evaluation. The automatic approach is more user-friendly and less error-prone but no automatic method for loop bounds analysis can give an exact answer for all loops. This paper presents a new method to extend the domain of computable loop bounds with covariant induction variables. We focus on finding a numeric value for a bound (iteration number), not on the termination of the program [4]. The WCET quality depends on the quality of the found bounds.

[1] Email: {michiel,bonenfant,casse}@irit.fr

doi:10.1016/j.entcs.2012.11.005

Several approaches have been explored in flow analysis concerning loop bounds [2,1,7,11,14,5]. Most of these approaches use Abstract Interpretation [6] to build a map which associates to the variables an abstract representation of its value. This permits to give safe, but possibly overestimated, sets of the states at the different program points.

A classical example of binary search is the program "`bs.c`" of the Mälardalen benchmark suite [3]. According to our knowledge the only approach resolving it are described in [14] and in [9]. Yet, [14] uses model checking by Binary Tightening and Widening and embedded it into a more general approach using a set of analysis methods. As an overall result it increases computational complexity for proving or dis-proving relevant time bounds of a program. SWEET [9] resolves loop bounds by using abstract execution that expands each loop iteration. In both cases, the genericity of the solution is at the cost of computation time, sometimes prohibitive. Moreover these tools obtain loop bound values and cannot be instanciated in different call contexts.

In [12] 74% bounds of Mälardalen benchmark loops are successfully computed, but there is no detail on which loops are processed (especially "`bs.c`").

oRange [13] combines loop bound expression built on C programs with abstract interpretation. For a loop L_i, two kinds of loop bounds are computed: $total_i$ (the total number of iterations in the overall execution of the program) and $maxi_i$ (the maximum number of iterations for each loop start-up). The analysis proceeds in three steps: 1) context-insensitive identification and normalization of increment and loop bound expressions [8], 2) context sensitive construction of $total_i$ and $maxi_i$ symbolic expressions, and finally 3) computation final $total_i$ and $maxi_i$ expressions and values. oRange supports numerous condition types (including && for example), the most frequent induction variable expressions (i.e. expressions containing +,-,×, /) and any type of C loop statements. The complex control flow statements (`break`, `goto`, etc) are processed by a semantics-preserving simplifier of the program representation. Complete description of oRange can be found in [13].

This paper is organised as follow: Section 2 is the motivation , Section 3 presents our method, Section 3.5 describes some limitations of the method, Section 4 is a typical example and we conclude in Section 5.

2 Bounding loops with arithmetico-geometric series

In the program below the function *oracle* returns an integer, possibly random, value:

Example 1: Basic Covariance

```
void main() {
   int i = 0,j = 10;
   while(i < j) {
     if(oracle()) i = (i + j) / 2;
     else j = (i + j) / 2;
   }
 }
```

The iteration number of the *while* loop depends on both variables, i and/or j, which are separately modified in both branches of the selection. As these variables are condition induction variables, they are called **covariant** and the loop iteration bound requires to survey the covariation of these variables. This type of loop is called **covariant**.

The goal of normalisation is to get a loop for which the loop variable get through all values of [0..borneMax] incrementing by 1. This permits to simplify computation, especially in case of nested loops.

In the initial analysis performed by oRange, if two or more paths lead to different incrementations of a condition induction variable, the associated increment value is approximated by $\top$ (any possible incrementation) because we cannot support multiple incrementations. To survey how both variables i and j evolve, a new meta-variable X is introduced such that $X = j - i$ sums up the evolution of i and j for the loop condition. X is a temporary variable which permits to process the regular normalization. To achieve this, we insert at the end of each block containing i or j modifications, a new assignment for X to take into account the effect of i and j changes on X. The `if` instruction of the example 1 becomes now:

Example 2: Introducing X for series determination

```
if(oracle()){
  i = (i + j) / 2;
  X = j - i;
}
else {
  j = (i + j) / 2;
  X = j - i;
}
```

Then, X is analyzed and the application of the variable rewriting algorithm of oRange on the assignment of the `then`-part of the selection produces $X_n \leftarrow j_{n-1} - (i_{n-1} + j_{n-1})/2 = (j_{n-1} - i_{n-1})/2$, that is, $X_n = X_{n-1}/2$. The assignment of the `else`-part is processed in the same way: $X_n \leftarrow (i_{n-1} + j_{n-1})/2 - i_{n-1} = (j_{n-1} - i_{n-1})/2$, that is, $X_n = X_{n-1}/2$. Finally, the forms of X on each branch can be joined to obtain, in this particular case, the geometric series $X_n = X_{n-1}/2$.

If the join can produce known series, the initial program is rewritten to embed X as a condition induction variable (initialization, loop condition, body incrementation) whose bound represents a safe estimation of the whole loop bound (as below).

Example 3: Final rewriting of the loop

```
void main() {
  int i=0, j=10, X;
  X = j - i;
  while(0 < X && i < j) {
    if (oracle()) i = (i + j ) / 2;
    else    j = (i + j) / 2;
    X = X/2;
  }
}
```

The first approach is correct (trivial by renaming) when the induction variables are floating point values. The integers make things a bit trickier because of properties of discrete arithmetics for division and modulo. In fact, the series we obtain

for integer cases (X has the form $X = X * q + k$ with $0 < q < 1$) do not always bound the actual executed series.

A table like table 1 represents the trace of a loop and will be used further. The two first columns represent the variation of i, j variables, the third column is the comparison used as right member of the loop condition. The X column is the values of X and the last is the values of the left member of the loop condition. The two last columns are the representation of the modified loop. We compare the behavior of the two conditions ($i < j$ and $0 < X$) in order to exhibit when X is not an adequate series. X represents the exact series if for all trace the two conditions change of value simultaneously, if the $0 < X$ changes ultimately after the other one, then we have found an overestimation. Otherwise, the series cannot bound the loop properly.

Table 1
Trace of example 3 and 3 corrected ($X = X/2 + 1$)

(a) without/with X condition

i	j	$i < j$	X	$0 < X$
0	10	*true*	10	*true*
5	10	*true*	5	*true*
7	10	*true*	2	*true*
9	10	*true*	1	*true*
9	10	*true*	0	*false*

(b) example 3 corrected

i	j	$i < j$	X	$0 < X$
0	10	*true*	10	*true*
5	10	*true*	6	*true*
7	10	*true*	4	*true*
9	10	*true*	3	*true*
9	10	*true*	2	*true*
9	10	*true*	2	*true*

Indeed, the table 1 (a) shows that the loop of example 1 (second column) do not terminate before to the rewritten loop of example 3 (fifth column) because of the condition $0 < X$. Series $X_n = X_{n-1}/2$ is not a correct model of the loop bound.

The real loop series have to be fixed by a constant E so as to ensure that the new series gives a sound upper bound of the real series (see proof in appendix 6). We define $E = \frac{n.m.(k-1)}{k} + \frac{m.(k-1)}{k}$ where n is the number of induction variables, m the number of terms divided and $k = 1/q$ the constant divisor of each term. In example 3, $n = 2$, $m = 1$ and $k = 2$ so $E = 2(2-1)/2 = 1$. So the resulting arithmetico-geometric series is $X_n = X_{n-1}/2 + 1$. By replacing $X = X/2$ by $X = X/2 + 1$ in example 3, we get the new trace displayed in table 1 (b). As expected, the loop of Example 1 and the rewritten one exhibit now the same behaviour (no termination in some cases).

3 Method

To solve this problem, we have to detect the covariant loops, insert X variable processing and bound X based on arithmetico-geometric series properties. This applies only in the first analysis step: the next two steps of oRange remain unchanged.

The original abstract interpretation performed in oRange uses the domain of abstract stores, AS: $AS : Id \to Expr$ where Id are the variables of the C program and $Expr$ are C expressions. This domain is applied on the program using *update*: $Stmt \times AS \to AS$ that asserts the effects of a statement $s \in Stmt$ on the program $\sigma \in AS$ and using $join : AS \times AS \to AS$ that merges the abstract stores of two

execution paths.

In order to compute covariance, we extend it to $AS^* : Id \cup \mathbb{X} \rightarrow Expr$ where $\mathbb{X}$ is the set of X meta-variables (which are introduced to represent the covariant variables). The *update* and $join$ functions remain the same for each variable $v \notin \mathbb{X}$. The aim of the following paragraphs is now to define $update'$ and $join'$, respectively, update and join functions for $\mathbb{X}$ variables. From this, $update^*$ and $join^*$ are easily defined by:

$$update^*(s,\sigma) = \begin{cases} update'(s,\sigma) \ if \ s = [X = e], X \in \mathbb{X} \\ update(s,\sigma) \ otherwise \end{cases}$$

$$join^*(\sigma_1,\sigma_2) = \sigma \text{ s.t. } \begin{cases} \sigma[X] = join^*(\sigma_1,\sigma_2)[X] \ if \ X \in \mathbb{X} \\ \sigma[v] = join'(\sigma_1,\sigma_2)[v] \ otherwise \end{cases}$$

3.1 Definitions

Let L be a loop, $c \in Expr$ its condition (expression) and $b \in Stmt$ its body. Let $LVE(c)$ be the set of variables in c and $LVA(b)$ the set of variables assigned inside b. Let $I = LVE(c) \bigcap LVA(b)$ and $n = |I|$ the number of condition induction variables in c. If $n > 1$ and c is a binary expression of the form $exp = exp_1 \ op \ exp_2$ [2] where $op \in \{<, \leq, \geq, >\}$ and at least $\exists x \in I, \exists y \in I$ such as x and y are modified at least in two different paths of b, then x and y are **covariant variables**.

3.2 Condition and its Induction Variable

We show here how to determine the performed increments in the loop body and how to check if they can be processed. Our current analysis only supports affine forms ($exp = k_0 + \Sigma_{i \in [1..n]} k_i \times x_i$).

Let k_i be the coefficient of the variable $x_i \in LVE(exp)$. k_i are in $\mathbb{Z}$ or in $\mathbb{R}$ depending on the type of the comparison operator. If all k_i can be resolved, we can find the increment of loop variables: $X = \Sigma_{i \in [1..n]} k_i \times x_i$.

In order to replace the analysis of the covariant variables by the one of X, we insert in the code the assignment $X = \Sigma_{i \in [1..n]} k_i \times x_i$ into specific places: before the loop body and after each block modifying an x_i. In addition, the condition c is replaced by $c' = 0 \ op \ X + k_0 \ \&\& \ c$. This definition allows to cope with bound of X even if an overestimation of X is exhibited.

3.3 Increment Evaluation

In this exploration step, we consider that the incrementation of induction variable is only performed in the body of the loop and neither in called functions nor in the body of an internal loop. In the actual implementation, we consider that in the

[2] $exp = exp_1 \ op \ exp_2$ when $op \in \{>, \geq\}$ and $exp = exp_2 \ op \ exp_1$ if $op \in \{<, \leq\}$ to have only to support $<$ or $\leq$ comparators

later cases the incrementation cannot be analyzed (although a conservative specific value of $\top$ is assumed).

In order to shorten the presentation, we only present cases where induction variables are modified in sequence or in conditional paths. Actually, other cases are processed in the actual implementation.

Abstract interpretation applies to a sequence of instructions and an abstract store σ and result in a new abstract store. Initial context is $\sigma_0 = \emptyset$. The statements are evaluated in sequence. Let pl be the last evaluated program point and σ_{pl} its context. We evaluate the next point p_j with σ_{pl} as input abstract store and the statement after the program point pl to the program point p_j. The result is the output context σ_{pl} and X symbolic expression.

Example 4: Computation points location

```
while(...) {          // p0
    if (oracle()) {
     i = (i + j) / 2;
     X = j - i;       // p1
    }
    else {
     j = (i + j) / 2;
     X = j - i;       // p2
    }                 // p3
}
```

The first evaluation, in p_1 obtains X ← j - (i + j) / 2. The second one, in p_2 obtains X ← (i + j) / 2 - i. According to the symbolic form of X at point p_j, we try to identify a form such as $X \leftarrow q_j X + k_{j0}$ known as arithmetic-geometric series [10]. We search if $\exists q_j \geq 0$ s.t. $\forall i \in [1..n]\ k_j = q_j \times k_{j,i}$. If we do not find the suitable form, the X is evaluated to $\top$.

3.3.1 Increment evaluation in a sequence: update′

Let $X = q_1 X + k_{10}$ be the first assignment of X in a sequence, and $X = q_2 X + k_{20}$ be the following one.

Table 2
update′

Case	Resulting increment	Example
$q_1 \neq 0,\ q_2 \neq 0$	$X = q_1 \times q_2 X + k_{10} \times q_2 + k_{20}$	$\left.\begin{array}{c} X = 3X + 1 \\ X = 4X + 2 \end{array}\right\} X = 12X + 6$
$q_1 = 0,\ q_2 \neq 0$	$X = k_{10} \times q_2 + k_{20}{}^*$	$\left.\begin{array}{c} X = 1 \\ X = 4X + 2 \end{array}\right\} X = 6$
$q_1 \neq 0,\ q_2 = 0$	$X = k_{20}{}^*$	$\left.\begin{array}{c} X = 4X + 2 \\ X = 1 \end{array}\right\} X = 1$

* when this X value is propagated along the paths, if it leads to exit the loop, then the loop is bounded, otherwise the loop may never terminate.

To enlarge the application domain of oRange, the covariance coefficients q and k are not only single values but may also be sets noted $SET(q_1, q_2)$ [3] where q_1, q_2 are single values. In order to apply this representation to our arithmetic-geometric

[3] oRange bounds the set to two values in order to avoid complexity explosion. Bigger sets are abstracted to $\top$.

series, we redefine operators $(+, \times ...)$ for SET in table 3, where q_1, q_2 are single values.

Table 3
Sequential operators

$\bullet \in \{\times, +\}$	q_2	$SET(f_3, f_4)$
q_1	$q_1 \bullet q_2$	$SET \begin{pmatrix} q_1 \bullet f_3, \\ q_1 \bullet f_4 \end{pmatrix}$
$SET(f_1, f_2)$	$SET \begin{pmatrix} q_2 \bullet f_1, \\ q_2 \bullet f_2 \end{pmatrix}$	$SET \begin{pmatrix} min \begin{pmatrix} f_1 \bullet f_3, f_1 \bullet f_4, \\ f_2 \bullet f_3, f_2 \bullet f_4 \end{pmatrix}, \\ max \begin{pmatrix} f_1 \bullet f_3, f_1 \bullet f_4, \\ f_2 \bullet f_3, f_2 \bullet f_4 \end{pmatrix} \end{pmatrix}$

3.3.2 Increment evaluation of join′

In the case of two expressions in X: one into each part of join, we set $X = q_1X + k_{10}$ and $X = q_2X + k_{20}$ as the two X assignments to join. The resulting increment depends on the four next ordered cases else it is undefined ($\top$). *join′* uses the operation *compose* defined in the table 4.

(i) if $q_1 = 0$ and if this X assignment is the one which leads to exit the loop, then $X = q_2X + k_{20}$ is the default increment. When the first branch of `if` is executed, either the constant value of X gives that the loop condition is false at the next iteration so the maximum number of iteration of the loop is obtained when the second branch is always executed. Either the loop condition is true so the loop may never terminate (resp. if $q_2 = 0$ is the last iteration case then $X = q_1X + k_{10}$) else it is undefined.

(ii) if $q_1 = q_2 = 1$ then it is a classical arithmetic form: if $k_{10} \times k_{20} > 0$ the resulting assignment of X is $X = X + compose(k_{10}, k_{20})$ else it is undefined.

(iii) if $k_{10} = k_{20} = 0$ then it is a classical geometric form: if ($q_1 < 1$ and $q_2 < 1$) or ($q_1 > 1$ and $q_2 > 1$) the resulting assignment of X is $X = X \times compose(q_1, q_2)$ else it is undefined.

(iv) else it is a arithmetico-geometric form, the resulting increment is:
if ($q_1 < 1$ and $q_2 < 1$ and $k_{10} < 0$ and $k_{20} < 0$) or ($q_1 > 1$ and $q_2 > 1$ and $k_{10} > 0$ and $k_{20} > 0$) then the series is bounded by $X = X \times compose(q_1, q_2)$ $+compose(k_{10}, k_{20})$ else it is undefined.

Table 4
compose

compose	q_2	$SET(f_3, f_4)$
q_1	if $q_1 = q_2$ then q_2 else $SET(q_1, q_2)$	if $q_1 < min(f_3, f_4)$ $SET(q_1, max(f_3, f_4))$ if $q_1 > max(f_3, f_4)$ $SET(min(f_3, f_4), q_1)$ else $SET(f_3, f_4)$
$SET(f_1, f_2)$	if $q_2 < min(f_1, f_2)$ $SET(q_2, max(f_1, f_2))$ if $q_2 > max(f_1, f_2)$ $SET(min(f_1, f_2), q_2)$ else $SET(f_1, f_2)$	$SET \begin{pmatrix} min \begin{pmatrix} f_1, f_3, f_1, f_4, \\ f_2, f_3, f_2, f_4 \end{pmatrix}, \\ max \begin{pmatrix} f_1, f_3, f_1, f_4, \\ f_2, f_3, f_2, f_4 \end{pmatrix} \end{pmatrix}$

We apply increment evaluation in example 5. When $compare() == 0$, $X = i - 1 - i = 1$, the test $0 < X$ is false it is the last loop iteration. The loop bound is obtained when this case is never taken. The obtained incrementation is $X = X/2 - 1$. The adjustment is $E = \frac{n.m.(k-1)}{k} + \frac{m.(k-1)}{k} = 2(2-1)/2 = 1$. So the resulting arithmetico-geometric series are $X_n = X_{n-1}/2 - 1 + 1$, pure geometric series. This sample 5 is equivalent to "`bs.c`".

Example 5: Increment evaluation application

```
void main() {
   int i = 0,j = 10;
   while(i < j) {
     if(compare()==0) j= i -1 ;
     else
        if(compare()>0) j= (i + j) / 2 - 1;
        else i = (i + j) / 2 + 1 ;
   }
 }
```

3.4 The arithmetico-geometric form

We consider strictly decreasing or increasing forms. Other forms are undefined. Classical arithmetic or geometric forms are not considered here (previous cases). We want to translate it into a geometric form using mathematical results. In order to do that, we replace the X series by new series V where $V_n = X_n + c$ and $c = -k_{10}/(1 - q_1)$. The normalisation is done by introducing V assignment in place of X one and change the condition according to V variable.

If it is not possible (because of SET), sometimes we may approximate the form $X = q_1X + k_{10}$ by bounding arithmetic (see the next ordered formulea) or geometric series.

(i) if *lower* $(q_1) = 1$ then if *lower* $(k_{10}) > 0.0$ then $X = X + k_{10}$ else undefined.
Ex.$X = X \times SET(2, 1) + 1$ then a bound is $X = X + 1$

(ii) if *upper* $(q_1) = 1$ then if *upper* $(k_{10}) < 0.0$ then $X = X + k_{10}$ else undefined.
Ex.$X = X \times SET(1/2, 1) - 1$ then a bound is $X = X - 1$

(iii) if $q_1 > 1$ then
  (a) if $X > 0$ then $X = q_1X$
  (b) if $X \geqslant 0$ then the same as in the previous case with $+1$ for the loop bound because if X = 0 when k_{10} is positive the loop increases too.
  (c) else undefined

(iv) if $0 < q_1 < 1$
  (a) if $X > 0$ then $X = q_1X$
  (b) if $X \geqslant 0$ then the same as in the previous case with $+1$ for the loop bound because if X = 0 when k_{10} is negative the loop decreases too.
  (c) else undefined

Definition: Let $q_1 \in C$ and $q_2 \in C$. Let *lower* (q_1) (respectively *lower* (q_2))=
(i) if $q_1 \in R$ then if q_1
(ii) if $q_1 = SET(f_1, f_2)$ then $min(f_1, f_2)$ (respectively $max(f_1, f_2)$)

3.5 Limitations

In this section, we present two variations of the initial example. Let modify the condition $i < j$ into $i + 1 < j$ (resp. $i + 2 < j$), we rewrite the condition on X as $0 < X - 1$ (resp. $0 < X - 2$). Table 5 represents the resulting traces.

Table 5
Trace of example 1 with modified condition

(a) with condition $i + 1 < j$

i	j	$i+1<j$	X	$0<X-1$
0	10	*true*	10	*true*
5	10	*true*	6	*true*
7	10	*true*	4	*true*
9	10	*false*	3	*true*
9	10	*false*	2	*true*

(b) with condition $i + 2 < j$

i	j	$i+2<j$	X	$0<X-2$
0	10	*true*	10	*true*
5	10	*true*	6	*true*
7	10	*true*	4	*true*
9	10	*false*	3	*true*
9	10	*false*	2	*false*

In the first case, the loop is bounded whereas the series is not: we cannot determine if the loop is bounded. In the second case, the loop and the series are bounded but the series reach the bound after the loop: the series are pessimistic.

In conclusion, when we find series, they will always bound the original loop. The limitation of our technique is that the bound may be exact, pessimistic or even undetermined.

4 Significant example

We present here a significant example inspired by industrial real time applications (this form of loop is not represented in Mälardalen benchmark). It features array and several covariant variables.

Example 6: Interval and series

```
int main() {
    int i=0,j=0,k=0;
    int tab[10][5][10]; // int X = i*50+10*j+k;
    while(i*50+10*j+k<500) { // c'= (i*50+10*j+k<500) && (X-500<0)
        // p0
        t[i][j][k]=0;
        if (k<9) // k in [0..8], j in [0..4]
            k++; // p1
        else if (j<4) { // k = 9 and j in [0..3]
                 k=0;
                 j++;  // p2 }
             else { // k= 9, j=4, i in [0..9]
                 k=0;
                 j=0;
                 i++;  // p3
             } // p4
    } // p5
}
```

First we proceed to interval analysis and obtain $i \in [0..9]$, $j \in [0..4]$ and $k \in [0..9]$. Then we compute the expression of X and obtain $X = X + 1$ for each point. In points p_2 and p_3, because we get neither the form $X = cte$ nor $X = cte1 * X + cte2$ we need to artificially re-introduce j and k thanks to the interval analysis (in p_3, we replace 0 by $k - 9$). The series obtained for this example is a single arithmetic form. The limit of the loop is 500.

5 Conclusion

The type of loops found in `bs.c` Mälardalen benchmark is now bounded by oRange. We call this family of loop *covariant*: the number of iteration depends on several increment variables having a correlated behavior. Such loops are found in industrial cases, and not only for binary search.

Our method consists to express the increment variables into a single one which has the behavior of arithmetico-geometric series. We can then apply oRange to this single increment variable.

The next work will be to introduce interval analysis. We think it will help to find bounding series when it was not possible before and improving the accuracy of the bounding series.

References

[1] Bound-t tool. http://www.tidorum.fi/bound-t, 2005.

[2] ait tool. http://www.absint.com, 2007.

[3] Wcet project. http://www.mrtc.mdh.se/projects/WCC08, 2008.

[4] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis (SAS 2010)*, volume 6337 of *LNCS*, 2010.

[5] Joel Coffman, Christopher A. Healy, Frank Mueller, and David B. Whalley. Generalizing parametric timing analysis. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07), San Diego, California, USA*, 2007.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[7] Christoph Cullmann and Florian Martin. Data-flow based detection of loop bounds. In *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy*, 2007.

[8] Marianne De Michiel, Armelle Bonenfant, Pascal Sainrat, and Hugues Cassé. Loop Normalisation to Evaluate Maximum Number of Iteration of Loop in WCET Context. Rapport de recherche IRIT/RR-2008-3-EN, IRIT, Université Paul Sabatier, Toulouse, février 2008.

[9] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy*, 2007.

[10] Jérôme Feret. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, 2005.

[11] Jan Gustafsson and Andreas Ermedahl. Experiences from applying wcet analysis in industrial settings. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), Santorini Island, Greece*, 2007.

[12] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[13] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Taiwan*, 2008.

[14] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland*, 2009.

6 Appendix

6.1 Preliminary proof

In this appendix, we prove that if we find a bound for a loop thanks to our method this bound is valid.

Let $I = \{x_i\}_{i=1}^n$ the set of the induction variables of the loop. Let $n = |I|$ its number of elements.

$\forall i \in [1..n]$, $x_i \in \mathbb{Z}$, $a_i \in \mathbb{Z}$ and $k \in \mathbb{N} - \{0\}$

In this proof we use the usual definition for integer / and % operators of C.

According to these definitions $\forall x \in \mathbb{Z}$, $x = (x/k) * k + r$ with $-(k-1) \leq r \leq k-1$.

So $\frac{x}{k} = \frac{-x}{k}$ and $-x\%k = (-x)\%k$. These properties permit us to consider that a difference is like a sum when using $-$ for each coefficient $a_i \in \mathbb{Z}$.

Let show that we can bound the following formula:

$A = \frac{\sum_{i=1}^n (a_i x_i)}{k}$. Let $a_i x_i = q_i k + r_i$ with $-(k-1) \leq r_i \leq k-1$.

$A = \frac{\sum_{i=1}^n (q_i k + r_i)}{k} = \sum_{i=1}^n q_i + \frac{\sum_{i=1}^n r_i}{k}$

Let $B = \sum_{i=1}^n \frac{(a_i x_i)}{k} = \sum_{i=1}^n \frac{(q_i k + r_i)}{k} \overset{\text{integer division}}{=} \sum_{i=1}^n q_i$

Again, thanks to the integer division we can bound $\sum_{i=1}^n \frac{r_i}{k}$ in the following way: $-\frac{n(k-1)}{k} \leq \frac{\sum_{i=1}^n r_i}{k} \leq \frac{n(k-1)}{k}$ thus $B - \frac{n(k-1)}{k} \leq \sum_{i=1}^n q_i + \frac{\sum_{i=1}^n r_i}{k} \leq B + \frac{n(k-1)}{k}$

This leads to our theorem:

Theorem 6.1

$$\sum_{i=1}^{n} \frac{a_i x_i}{k} - \frac{n(k-1)}{k} \leq \frac{\sum_{i=1}^n (a_i x_i)}{k} \leq \sum_{i=1}^{n} \frac{a_i x_i}{k} + \frac{n(k-1)}{k}$$

6.2 Proof for bounding formulae

We would like to bound formula of the following type: with $b \in \mathbb{Z}$, $S = b + \sum_{i=1}^n (a_{0i} x_i) + \sum_{j=1}^m \frac{\sum_{i=1}^n (a_{ji} x_i + b_j)}{k}$

In order to make the prove simpler we first consider that $\forall j \in [1..m]$ $b_j = 0$ thus S becomes $S = b + \sum_{i=1}^n (a_{0i} x_i) + \sum_{j=1}^m \frac{\sum_{i=1}^n (a_{ji} x_i)}{k}$.

With $T_0 = \sum_{i=1}^n (a_{0i} x_i)$ and $A_j = \frac{\sum_{i=1}^n a_{ji} x_i}{k}$, $S = b + T_0 + \sum_{j=1}^m A_j$

By applying theorem 6.1 we obtain $b + T_0 + \sum_{j=1}^m \sum_{i=1}^n \frac{a_{ji} x_i}{k} - \frac{nm(k-1)}{k} \leq S \leq b + T_0 + \sum_{j=1}^m \sum_{i=1}^n \frac{a_{ji} x_i}{k} + \frac{nm(k-1)}{k}$

We introduce a new value $U = b + \frac{\sum_{i=1}^n (k a_{0i} + \sum_{j=1}^m a_{ji}) x_i}{k} = b + \frac{k T_0 + \sum_{i=1}^n \sum_{j=1}^m a_{ji} x_i}{k} = b + T_0 + \frac{\sum_{i=1}^n \sum_{j=1}^m a_{ji} x_i}{k}$

Thanks to theorem 6.1 we can bound U:

$b + T_0 + \sum_{j=1}^m \frac{\sum_{i=1}^n (a_{ji} x_i)}{k} - \frac{m(k-1)}{k} \leq U \leq b + T_0 + \sum_{j=1}^m \frac{\sum_{i=1}^n (a_{ji} x_i)}{k} + \frac{m(k-1)}{k}$

Thus, we can rewrite this inequality using S:

$S - \frac{m(k-1)}{k} \leq U \leq S + \frac{m(k-1)1}{k}$ so $S \leq U + \frac{m(k-1)}{k}$

This leads to bound S:

$S \leq U + \frac{m(k-1)}{k} \leq b + T_0 + \sum_{j=1}^m \sum_{i=1}^n \frac{(a_{ji} x_i)}{k} + \frac{nm(k-1)}{k} + \frac{m(k-1)}{k}$

Let $E = \frac{nm(k-1)}{k} + \frac{m(k-1)}{k}$

As we are able to compute $b + T_0 + \sum_{j=1}^m \sum_{i=1}^n \frac{(a_{ji} x_i)}{k}$ and E, we are able to bound S.

Remark: when $b_j \neq 0$, $E = \frac{(n+1)m(k-1)}{k} + \frac{m(k-1)}{k}$