Loop normalization (suite)

IRIT/RR–2010-17–EN

Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, Pascal Sainrat

Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)
118, route de Narbonne, 31062 Toulouse Cedex, France
Hipeac European Network of Excellence
E-mail: [michiel, bonenfant, casse, sainrat]@irit.fr

Abstract

One of the important steps in processing the worst case execution time (WCET) of a program is to determine the loops upper bounds. Such bounds are crucial when verifying real-time systems. In a previous paper we have presented our loop bound evaluation method oRange based on three steps: normalization of loops, building expression of the bounds, applying the context to the expression. This paper presents some extensions of the first step (normalization), in particular to resolve some multi-increment of loop variable and some cases where the increment variable is initially undefined.

1. Introduction

In order to provide bound for loops of a program, we proceed to a normalization. The goal of the normalization is to rewrite all loops into a single form: a for with a increment of +1. In a previous report, [3], we present how we deal with the different type of loops (for, while, do-while...), with the different type of constant increment (+,-,*, /), with several condition type (==, !=,<, <=, >, >=) and also with the nested loops. We also demonstrate equivalence in term of iteration number between the initial loop and its normalized form.

In this actual report we present an extension under the same conditions of the previous normalization report.

In section 2, we give an overview of the general method of normalization. In section 3 we describe all type of loop of this report. The sections 4 to 8 we present how each type of loop is normalized. Section 9 gives a synthesis of normalization for the loop examples of section 3. And then we give our conclusion.

2. Method

For each identified loop we proceed to a normalization. According to the loop condition, we identify the variables of the condition and we try to find their induction in order to construct an expression of the loop bound and to normalize the loop.

The normalization consists mainly of transforming each loop L_i into a basic `for` loop having an induction variable var_i, an increment of $+1$, an expression of the number of executions exp_i (derived from loop condition and increment), and a body $body_i$ (this body is represented by a type $inst$ instruction). The normalization step is described in detail in [3]. All loops may be normalized but in some cases exp_i is not defined (for instance, if we are not able to find an increment for the loop variable).

The normalization computes -1)- $filterBody_i$: a restriction of instructions from $body_i$ having an effect on the loop condition (program slicing is used to remove irrelevant code), then -2)- $getInc\ [varOfCond_i]\ [filterBody_i]$: $getInc$ gets the increment of each variable of the condition in filtered body, it uses $evalStore$ in some cases to build the *abstract stores*

of the $filterBody_i$ instructions independently of the loop context -3)- Finally we combine the $getInc$ result with the condition and obtain exp_i. For instance, a loop like:

```
for (i=k; i <=n ; i+=c);
```

has for body: $filterBody_i = \{i+ = c; \}$ $getInc[i, n]\,[filterBody_i] = [(i, +c), (n, Constant)]$

thus exp_i = $\lfloor (n - k)/c + 1 \rfloor$.

Note: When exp_i is not computed it is set to top (not defined).

The function $getInc$ uses rules to get increment for each variable of the condition (looking for their assignments into the loop body). The rules depend on the kind of control structure containing the assignment.

let $nbIt$ the number of time the loop is iterated for one loop call.

3. Types of loop treated in this report

We are able to identify most cases of *multiple increment* (Algorithm 1), *indirect increment* (Algorithm 2), boolean conditions (Algorithm 3), array-dependent increments (Algorithm 4-5) and , *multiple variable loops* (Algorithm 2),

3.1. Multiple increment

We call multiple increment the case of a loop where multiple instructions modify alternatively the increment variable(s), especially in different conditional branch.

Algorithm 1 Example of multiple increment

```
i=0;
while (i<N)
{...
if (condition) i+=2; else i+=3; i++;
...
}
```

3.2. Simple Indirect increment

The loop variable is not directly increased but assign to an other variable witch is increased itself (+k, -k, /k or *k form).

Algorithm 2 Example of indirect increment

```
i=0; j=3; while (i<N) { ... j++; i=j; ... }
```

3.3. Simple Boolean condition

The loop variable is variable of type boolean. Its value depends on a value of a numerical variable i which creases or decreases with +k, -k, /k or *k form. The boolean variable is modified in a conditional branch depending on i (i.e. i<... or i>...). If tmp is the loop boolean variable we are able to normalise

```
if (cond) tmp=0 (respectively tmp=1);
```

where cond is a BINARY comparaison expression (operator < >= > >= == !=) classified in this report as in the previous one.

Algorithm 3 Example of boolean condition

```
i=0; bool=1;
while (bool)
{... if (i>=N) bool = 0; ... i++; }
```

3.4. Simple Array condition

The loop variable is an array element. By assumption, in Algorithm 4, the C code is considered to be correct, and therefore it does not contain out-of-bound accesses. We therefore replace the condition by $t[i] < N \ \&\& \ i < M$.

Algorithm 4 Example of array

```
int t[M]; i=0;
while (t[i]<N) { ... i++; ... }
```

In Algorithm 5, if k is known, $t[k]$ is used to evaluate the value of the bound. If k is unknown and $t[k]$ not changed in the context, we replace $t[k]$ by SET(1,5)[1], expressing the lowest and the highest values of $t[k]$.

Algorithm 5 Example of array (2)

```
int t[] = {1,2,3,5}; i=0;
while (i<t[k]) { ... i++; ... }
```

3.5. Simple Multi variable condition

Two variables in the same comparative expression are modified in the loop.

Algorithm 6 Exemple of multiple variables

```
i=0; j=10;
while (i<j) { ... i++; j–;... }
```

4. Multi-increment

In this section, we give details on how we treat multiple increment of multi-increment.

Let a loop i be represented by $instLoop_i$ = LOOP of ($identifier$: int) * ($loopvar$: string) *($loop$: exp) * ($body$: inst) which represents a LOOP into our instruction grammar for the abstract interpretation. To find the first expression of the loop bound, we extract of the loop expression exp (that is the loop condition) the set of variable used into the exp. Let Vc_i be this set. To reduce the loop normalization step of our method, we filter the loop instructions ($body$) to keep only the assignment which affects a variable of Vc_ior a variable used into an assignment expression of one of this variable and so one. So we construct a fixed point operator to keep only the useful assignments. Let Av_i be the subset of loop assigned variable of Vc_i.

In order to construct the initial expression of the loop bound, we search the increment of each variable $v \in Av_i$.

The type of increment used can be:$+k$, $-k$, $\times k$, $/k$, some multiple increments, where k is a constant (i.e. an expression containing only constant and/or invariant variables). The value of this kind of variable may be known locally or not.

For a non evaluated loop the iteration variable, $loopvar_i \in [0.. + \infty[$ more generally for a given call of the loop k (k is the call function number where the loop is executed) $loopvar_{ik} \in [0..nbItMax_{ik}[$ and the number of loop iteration is $nbIt_{ik} \in [nbItMin_{ik}..nbItMax_{ik}]$. $nbItMin_{ik}$ and $nbItMax_{ik}$are positives or nil values. These values are generally not known during the first step but some manual annotations can sometime give them.

In order to define an increment, we define an incrementType which represents the type of the increment (+,-...) and a value which represents the absolute value of the increment. The value of the increment may be $NOINC$ which means empty instruction or instruction which do not change x value (i.e. not assigned or assigned with the neutral value for the operator), $NOCOMP$ if we cannot compute the increment either $INC(type, expression)$ if we have evaluated an increment. We define an increment grammar for a variable x:

$$
\begin{aligned}
&type\ incrementType = + | - | * | NODEF \\
&type\ incrementValue = NOINC \\
&| NOCOMP \\
&| INC\ of\ incrementType * expression
\end{aligned}
$$

Note: * and / operators are equivalent if we change $x = x/2$ increment by $x = x * 1/2$)

[1] SET is a construction used to deal with different possible values, it helps to choose the one which maximize (or minimize) the expression. .

4.1. Research of the loop recurrence expression of a variable x

We now define how to evaluate the increment resulting on each case of our grammar instruction for the abstract interpretation: an instruction may be a variable assignment, an array assignment, a sequence instruction, an if instruction, a loop instruction or a function call instruction.

Notations: Let v_l a positive constant for the loop i.
Let s_l be an increment type.
Let e_l be an extended constant expression for the loop i.
Let a_l be an other kind of expression.
Let $inst1$and $inst2$ be two instruction sequences having respectively $inc1(x)$ and $inc2(x)$ increments for the x variable.
Let i be a normalized loop.

Variable and array assignment: For a single variable x, let $exp(x)$ be its assignment, we consider the function $getInc(x, exp(x))$ which extract an increment like ($+k$, $-k$, $\times k$, $/k$ or $NOCOMP$ where $k >= 0$ is for $+$ and - operator and $k >= 1$ for * and / operator) of the $exp(x)$ expression (as an example $getInc(x, x \longleftarrow x + 1) = INC(+, 1)$).
Let the access function to the type and value of the increment be.
$Let\ gettypeinc = match\ inc\ with\ (t, \_) > t|\_- > NODEF$
$Let\ getvalueinc = match\ inc\ with(\_, v) > v|\_- > NOCOMP$
Array assignment of x are firstly considered as $NOCOMP$.

Loop instruction: We defined a function$extractIncOfLoopinstLoop_i\ x$ which extract the x increment value of the loop body.
The result of the function may be

- $NOINC$: in case x is not assigned in the loop body
- $NOCOMP$ in the other cases

It seems difficult to evaluate x multi-increment into an internal loop so we just consider the resulting assignment of x, currently $NOCOMP$.

Sequence and alternate instructions: We only define how join of instruction can be done in these two cases.
* in fact it has been extended in some case :

```
while(i<N){
if (i>=5) i*=2
else i+=2;
}
```

if i is an integer, in each cases the loop variable is increased and because of the condition $i* = 2$ is executed if $i >= 5$ so the increase of x in that case is $<= 5 * 2 - 5 = 5$ so the minimal increment is 2 for i in the loop.
In order to do that we need to introduce interval for integer. We will do it in the normalization step.

Function call instruction: To simplify we do not curently consider multiple increment into a function call of the loop. So we defined the $getIncOfCall$ function.

- x may be not changed by the function ($NOINC$)
- x may be a global variable changed by the function $NOCOMP$
- x may be an output of the function $NOCOMP$

Let $INC(x)$ be the commutative join of $INC_1(x)$ $INC_2(x)$	for a sequence $inst_1 inst_2$ joinSequence x $inst_1$ $inst_2$=	for an alternate $if(...) then\ inst_1\ else inst_2$ joinAlternate x $inst_1$ $inst_2$=
$INC(+, v_1), INC(+, v_2)$	$INC(+, (v_1 + v_2))$	$INC(+, min(v_1, v_2))$
$INC(-, v_1), INC(-, v2)$	$INC(-, (v_1 + v_2))$	$INC(-, min(v_1, v_2))$
$INC(*, v_1), INC(*, v2)$	$INC(*, (v_1 * v_2))$	$INC(*, min(v1, v2))$
$INC(/, v_1), INC(/, v2)$	$INC(/, (v_1 * v_2))$	$INC(/, min(v1, v2))$
$INC(*, v_1), INC(/, v_2)$	$if\ v_1/v_2 >= 1\ then$ $INC(*, (v_1/v_2))$ $else INC(/, (v_1/v_2))$	$NOCOMP$
$INC(-, v_1), INC(+, v_2)$	$if\ -v_1 + v_2 >= 0\ then$ $INC(+, (-v_1 + v_2))$ $else INC(-, (-v_1 + v_2))$	$NOCOMP$
$INC(s_1, a_1), NOCOMP$	$NOCOMP$	$NOCOMP$
$INC(s_1, a_1), NOINC$	$INC(s_1, a_1)$	$NOCOMP$
$NOINC, NOINC$	$NOINC$	$NOINC$
$others$	$NOCOMP$	$NOCOMP$ in general *

Table 1. join of two inc

Variable and array assignment: We have defined how to evaluate each kind of instruction and here we define $getIncOfInstList$ a function describing how a list of instruction is evaluated.

```
Let rec getIncOfInstList x iList =
if iList = [] then NOINC
else
let (firstinst, nextInst) =
  (List.hd iList, List.tl iList) in
match firstInst with
VAR (id, exp) ->
let inc1 =
  if id = x then
getInc(x, firstInst)
else NOINC
in
joinSequence(x, inc1,
getIncOfInstList (x, nextInst))

| ARRAY (id, _, _) ->   getIncOfInstList (x, nextInst)
|  MEMASSIGN (id, _, _) -> //refer variable *x = *x+1
let inc1 =
  if id = x then
NOCOMP
else NOINC
in
joinSequence(x, inc1,
getIncOfInstList (x, nextInst))

| BEGIN list ->
joinSequence(x, getIncOfInstList (x, list),
getIncOfInstList (x, nextInst))
```

```
    | IF (_, i1, i2) ->
```
$joinSequence(x,$

$joinAlternate(x,$
$getIncOfInstList\ (x,\ (list(i1))),$
$getIncOfInstList\ (x,\ (list(i2)))),$
$getIncOfInstList\ (x,\ nextInst))$

```
    | LOOP ( _, body) ->
```

$joinAlternate(x,$
$extractIncOfLoop\quad (body,\ x)$
$getIncOfInstList\ (x,\ nextInst))$

```
    | CALL (_)->
```
$joinSequence(x,$
$getIncOfCall\ (x, firstInst),$

$getIncOfInstList\ (x,\ nextInst))$

5. Simple Indirect increment

We present here simple indirect increment loops which are the loops where the loop variable is not directly dependent of an increment but where the loop variable is assigned with an other loop variable which creases or decreases with +k, -k, /k or *k form. The following examples presented contain *i1* and *i2* instructions list which do not infer onto the loop iteration number.

- Let the loop :

```
j=v1;
i=v2;
while(j<=N)
{
i1
i++;
j=i;
i2
}
```

For these loop, $nbIt$ = if v1 > N : 0 else : $nbIt$ for the next do while loop.

An equivalent code is :

```
j=v1;
i=v2;
if (j<=N)
{
do
{
i1
i++;
j=i;
i2
}while(j<=N);
}
```

Form $nbIT$ = if inf+inc >sup : 1 else : $\lfloor(\lfloor sup\rfloor - inc - inf)/inc + 1\rfloor + 1$as it has been proved into [3].

then for the previous loop $nbIt$= $if\ v2+1 > N : 1\ else : \lfloor(\lfloor N \rfloor - v2 - 1)/1 + 1\rfloor$ (* if the loop is executed we have to introduce an if instruction*)

Remark bounds depend on where i is assigned (before or after $j = i$).

- Do while case :

```
...
i=v2;
do
{
   i++;
   j=i;
}while(j<=N);
```

Form $nbIT$ = if inf+inc >sup : 1 else :$\lfloor(\lfloor sup \rfloor - inc - inf)/inc + 1\rfloor + 1$

then for the previous loop nbIt = $if\ v2+1 > N : 1\ else : \lfloor(\lfloor N \rfloor - v2 - 1)/1 + 1\rfloor + 1$

- i is incremented next :

```
...
i=v2;
do
{
   j=i;
   i++;
}while(j<=N);
```

Form $nbIT$ = if inf >sup : 1 else :$\lfloor(\lfloor sup \rfloor - inf)/inc + 1\rfloor + 1$

for the previous loop $nbIt = if\ v2 > N : 1\ else : \lfloor(\lfloor N \rfloor - v2)/1 + 1\rfloor + 1$

The loop for (i = sup ; i >= inf; i-=inc) and the loop for(i=inf; i <= sup; i+=inc);
have the same number of iteration as it has been proved into [3].
We consider here the arithmetic loops :

```
L1: for(i=inf,j=v1;j<=sup;i+=inc,j=i) i1;
L2: for (i=sup, j=v1; j>=inf;i-=inc,j=i)i1;
L3: for(i=inf,j=v1;j<=sup;j=i,i+=inc)i1;
L4: for (i=sup,j=v1;j>=inf;j=i,i-=inc)i1;
L5: j=v1; i=inf;
   while(j<=sup) {i1;i+=inc;j=i;i2;};
L6: j=v1; i=sup;
   while(j>=inf) {i1;i-=inc;j=i;i2;};
L7: j=v1; i=inf;
   while(j<=sup) {i1;j=i;i+=inc;i2;};
L8: j=v1; i=sup;
   while(j>=inf) {i1;j=i;i-=inc;i2;};
L9: j=v1; i=inf;
   do{i1;i+=inc;j=i;i2;}while(j<=sup);
L10: j=v1; i=sup;
   do{i1;i-=inc;j=i;i2;}while(j>=inf);
L11: j=v1; i=inf;
   do{i1;j=i;i+=inc;i2;}while(j<=sup);
L12: j=v1; i=sup;
   do{i1;j=i;i-=inc;i2;}while(j>=inf);
```

And the geometric loops :

loop type	loop bound	adding test :	Before	Case Num
L1,L5	if inf+inc >sup : 1else : $\lfloor(\lfloor sup\rfloor - inc - inf)/inc + 1\rfloor + 1$	if (v2<=sup)	yes	1
L2, L6	if inf+inc >sup : 1else : $\lfloor(\lfloor sup\rfloor - inc - inf)/inc + 1\rfloor + 1$	if (v2>=inf)	yes	2
L9,L10	if inf+inc >sup : 1 else : $\lfloor(\lfloor sup\rfloor - inc - inf)/inc + 1\rfloor + 1$		yes	3
L3, L7	if inf >sup : 1 else : $\lfloor(\lfloor sup\rfloor - inf)/inc + 1\rfloor + 1$	if (v2<=sup)	no	4
L4, L8	if inf >sup : 1 else : $\lfloor(\lfloor sup\rfloor - inf)/inc + 1\rfloor + 1$	if (v2>=inf)	no	5
L11, L12	if inf >sup : 1 else : $\lfloor(\lfloor sup\rfloor - inf)/inc + 1\rfloor + 1$		no	6

Table 2. Arithmetic loop evaluation

loop type	loop bound	adding test :	Before	Case num
L13,L17	if sup *inc < inf : 1 else : $\lfloor log_{inc}(sup * inc/inf) + 1\rfloor + 1$	if (v2<=sup)	yes	7
L14, L18	if sup *inc < inf : 1 else : $\lfloor log_{inc}(sup * inc/inf) + 1\rfloor + 1$	if (v2>=inf)	yes	8
L21,L22	if sup *inc < inf :1 else : $\lfloor log_{inc}(sup * inc/inf) + 1\rfloor + 1$		yes	9
L15, L19	if sup < inf : 1 else: $\lfloor log_{inc}(sup/inf) + 1\rfloor + 1$	if (v2<=sup)	no	10
L16, L20	if sup < inf : 1 else: $\lfloor log_{inc}(sup/inf) + 1\rfloor + 1$	if (v2>=inf)	no	11
L23, L24	if sup < inf : 1 else: $\lfloor log_{inc}(sup/inf) + 1\rfloor + 1$		no	12

Table 3. Geometric loop evaluation

```
L13: for(i=inf,j=v1;j<=sup;i*=inc,j=i) i1;
L14: for (i=sup,j=v1;j>=inf;i/=inc,j=i)i1;
L15: for(i=inf,j=v1; j<=sup;j=i,i*=inc)i1;
L16: for (i=sup,j=v1;j>=inf;j=i,i/=inc)i1;
L17: j=v1; i=inf;
   while(j<=sup) {i1;i*=inc;j=i;i2;};
L18: j=v1; i=sup;
   while(j>=inf) {i1;i/=inc;j=i;i2;};
L19: j=v1; i=inf;
   while(j<=sup) {i1;j=i;i*=inc;i2;};
L20: j=v1; i=sup;
   while(j>=inf) {i1;j=i;i/=inc;i2;};
L21: j=v1; i=inf;
   do {i1;i*=inc;j=i;i2;}while(j<=sup);
L22: j=v1; i=sup;
   do {i1;i/=inc;j=i;i2;}while(j>=inf);
L23: j=v1; i=inf;
   do {i1;j=i;i*=inc;i2;}while(j<=sup);
L24: j=v1; i=sup;
   do {i1;j=i;i/=inc;i2;}while(j>=inf);
```

Into [3]the loop bound of for (i = sup ; i >= inf; i/=inc)
and the loop bound of for(i=inf; i <= sup; i*=inc) are defined by

Form of for or while loops : $nbIT$ = if sup < inf : 0, else$\lfloor log_{inc}(sup/inf) + 1\rfloor$

Form of do while loops : $nbIT$ = if sup *inc< inf : 1 else : $\lfloor log_{inc}(sup * inc/inf) + 1\rfloor + 1$

6. Simple Boolean condition

The loop variable is variable of type boolean. Its value depends on a value of a numerical variable i which crease or decrease with +k, -k, /k or *k form. The boolean variable is modified in a conditional branch depending on i (i.e. $i <$... or $i >$...).

The following examples contain *i1* and *i2* instructions list which do not infer onto the loop iteration number.

6.1. Classical form

```
i=inf;
tmp = 1;
while(tmp)
{
   if (i>sup)tmp=0; //=> exit the next iteration, when i>sup similar to 2 .
   i+=inc;
}
```

In the first sequence the loop is iterated as soon as in the second case.

```
i=inf;
tmp = 1;
if(tmp) for(i=inf,j=sup;j<=sup;j=i,i+=inc)
```

Form $nbIt = if\ tmp = 0 : 0$ else if inf >sup : 1 else :$\lfloor(\lfloor sup \rfloor - inf)/inc + 1\rfloor + 1$ (after)

```
i=inf;
tmp = 1;
while(tmp)
{
   i+=inc;
   if (i>sup)tmp=0;
//tmp =!( i>sup)

}
```

Form $nbIt = if\ tmp = 0 : 0$ else if inf+inc >sup : 1else :$\lfloor(\lfloor sup \rfloor - inc - inf)/inc + 1\rfloor + 1$

```
other possibilities not currently treat
i=inf;
tmp = 1;
while(tmp)
{
   i+=inc;
   tmp=i<=sup;

}

tmp =1;
j=1;
while (tmp)
{
          j++;
          if (j>20) tmp = 0;
          j++;
}
```

Form $nbIt = if \quad tmp = 0 : 0\ else : ....\lfloor(\lfloor sup \rfloor - incbefore - inf)/inctotal + 1\rfloor + 1$

Conclusion : These cases have loop bounds like in indirect cases (tables2 and 3). There is one case less and no adding test.

6.2. Combining boolean condition and comparative expression

This subsection treats of the combination of two conditions: boolean variable and comparison. We explore the case when the combination is "&&".

```
i=inf;
tmp = 1;
while(tmp&&i<=sup2)
{
   if (i>sup1)tmp=0;
   i+=inc;

}
```

Form $nbIt = if \quad tmp = 0 \ or \ i > sup2 : 0 \ else : minimum$between form 4 for tmp and classical form for the other condition.

$$\min(\lfloor(\lfloor sup1\rfloor - inf)/inc + 1\rfloor + 1, \lfloor(\lfloor sup2\rfloor - inf)/inc + 1\rfloor) = 1 + \min(\lfloor(\lfloor sup1\rfloor - inf + inc)/inc\rfloor, \lfloor(\lfloor sup2\rfloor - inf)/inc\rfloor)$$

$$= 1 + \lfloor(\tfrac{1}{inc}\min(\lfloor sup1\rfloor + inc), \lfloor sup2\rfloor) - inf\rfloor$$

in fact it is only the upper bound which have changed.

We do not consider test combining two different loop variables because of the complexity of formulea when we have internal loops. But it may be introduce for non internal loops.

For example :

```
void main()
{
char __tmp_0__ = 0;
int semecond = 1;
for(i = 0, j = 1; !__tmp_0__ && i < 100; __tmp_0__ || (i++, j += 3))
{
if(j > 75 && semecond || j > 300)
__tmp_0__ = 1 != 0;
}
__tmp_0__ = 0;
}
```

7. Simple Array condition

These cases (3.4) are resolved, during the normalization phase, only by adding condition (&&) to the initial loop condition.

8. Simple Multi variable condition

If then are two loop variables into the same relational expression, we evaluate the increment of each of these variables. In case of compatible increment we determine if the left-right expression may have a evaluated increment and in that case we construct an equivalent loop in term of number of iteration.

Example :

```
while(i<j)
{
i++;
j--;
}
```

The increment of i is : +1. The increment of j -1 so if we define the x variable x=i-j x increment is +2.

So the previous loop has the same number of iteration than the loop.

```
x=i-j;
while(x<0&&i<j)
{
if (i<2)i++;
j--;
x=x+2;
}
In that second case x increment is +SET(1,2).
while(i<j)
{
if (i<2)i++;
j--;
}
```

As conclusion each time we are able to minimize x increment value we are able to find a upper bound for the loop number of iteration.

Cases that are not actually bounded

```
while(i<j)
{
if (i<2)i++;
else j--;
}
We are working in that case.
```

9. Synthesis of cases

Loop	condition	information	expressions	increment	
id	$cond_i$		exp_i	variable	value
L1 (1)	$i < N$		$\lfloor$ (N - i - 1) / 3 $\rfloor$ +1	i	+3
L2 (2)	$i < N$&& $j < N$	i , before	$\lfloor$ (N - j - 1) / 1 $\rfloor$ +1	j	+1
L3 (3)	bool && $i < N$	bool , after (+1)	$\lfloor$(N-1-0)/1$\rfloor$+1)+1	i	+1
L4 (4)	t[i]<N &&$i < M$		$\lfloor$ (M - i - 1) / 1 $\rfloor$ +1	i	+1
L5 (5)	i<t[k]		$\lfloor$ (t[k] - i - 1) / 1 $\rfloor$ +1	i	+1

Table 4. Identification of increment variables, values and conditions, expression

10. Conclusions

In this report, we treat more loop as in the previous report. We now resolve sereval cases we have encounter in [1, 2].

References

[1] Mascotte homepage, 2006.

[2] Merasa project homepage,, 2007.

[3] Marianne De Michiel, Armelle Bonenfant, Pascal Sainrat, and Hugues Cassé. Normalisation de boucles pour l'évaluation du nombre maximum d'itérations. Rapport de recherche IRIT/RR-2008-3-FR, IRIT, Université Paul Sabatier, Toulouse, février 2008.