An automatic parametric approach for WCET analysis of C programs

D. Kebbal

Institut de Recherche en Informatique de Toulouse
118 route de Narbonne - F-31062 Toulouse Cedex 9 France
Djemai.Kebbal@iut-tarbes.fr

Abstract: In this paper, we propose a static worst-case execution time (WCET) analysis approach aimed to automatically extract flow information related to program semantics. This information is used to reduce the overestimation of the calculated WCET. We focus on flow information related to loop bounds and infeasible paths. The approach handles loops with multiple exit conditions and non-rectangular loops in which the number of iterations of an inner loop depends on the current iteration of an outer loop. The WCET of loops is analytically computed and expressed as summations function of the loop bounds. This avoids unfolding loops while providing tight and safe WCET estimate. Furthermore, the provided WCET expressions are expressed symbolically function of the program input parameters. This allows to reduce the WCET computing cost while providing tight WCET values. Indeed the WCET of each piece of code is expressed as symbolic expression which is instantiated each time that piece is called in the program. The flow analysis uses an enhanced symbolic execution approach based on symbolic execution and an analytic method in order to avoid unfolding loops performed by symbolic execution-based approaches.

Keywords: automatic parametric flow analysis, block-based symbolic execution, symbolic expression simplification.

1 Introduction

Real-time systems validation requires the knowledge of the execution time or bounds on the execution time of programs. WCET analysis is a well used approach in the validation of the temporal constraints of hard real-time systems.

WCET analysis consists of computing an upper bound on the execution time of a program rather than the exact WCET values as this problem is undecidable in the general case. However, it is imperative that WCET analysis must guarantee the *Safeness* and the *Tightness* of the provided WCET values in order to keep real-time systems predictable and their cost financially reasonable.

Static WCET analysis performs a high-level static analysis of the program source or object code. This avoids working on the program input data. For each component of the program, an upper bound of the execution time is estimated. Static WCET analysis techniques proceed generally in three phases [3, 6]: flow analysis, low-level analysis and WCET estimate computing. Flow analysis characterizes the execution sequences of the program's components, their execution frequency, etc. (execution paths). In this phase, the execution costs of basic blocks are assumed to be constant. Generally, two types of flow information may be extracted. The first category is related to the program structure and may be extracted automatically. The second category is related to the program functionality and semantics. This includes information about loop bounds and feasible/infeasible paths especially. Low-level analysis computes an execution time estimate of each program component on the target hardware architecture. Finally in the calculation phase, a WCET estimate of the whole program can be computed based on the results of the two previous steps.

The remainder of the paper is organized as follows: in the next section we review the related work. Section 3 describes and discusses the flow-analysis approach. In section 4, we address the algebraic evaluation of iterative scopes issue. Section 5 presents and discusses a formalism used to handle the generated symbolic expressions and their simplification. Finally, in section 6, we conclude the paper and present some perspective issues.

2 Related work

One of the most popular methods for static WCET analysis are based on path analysis, which proceed by explicitly enumerating the set of the program execution paths [9, 1]. [8] describes a method based on cycle-level symbolic execution to predict the WCET of real-time programs on high performance processors. The main drawbacks of those approaches lie in the important number of the generated paths which scales exponentially with the program size. Another class

of approaches called IPET[1] do not explicitly enumerate all program paths. In this class of techniques, the problem of the WCET estimation is converted to an ILP[2] problem [6, 10, 3].

However, all those approaches involve the programmer in the flow information determination process, especially the flow information related to program semantics (feasible/infeasible paths, loop bounds, etc.). Though the provided flow information may be highly precise, this is an error-prone problem. Interval-based abstract interpretation and symbolic execution methods [4, 2] allow to automatically extract flow information related to program semantics. These methods proceed by rolling out the program (especially loops) until it terminates, which is very costly in time and memory. In [5], an approach for determining loop bounds analytically without unfolding them is presented. They consider loops with multiple exit conditions and non-rectangular loops, in which the number of iterations of an inner loop depends on the current iteration of an outer loop. However, they provide only numerical WCET values which may lead to overestimation in the case of multiple calls to the same subprogram. [7] presents a parametric approach based on abstract interpretation and a method for counting integer points in polyhedra. The approach seems complex in practice. [1] proposes a method to compute symbolic WCET. However, they use an annotation mechanism involving the programmer in the flow analysis process.

We propose an automatic flow analysis approach which allows to automatically compute WCET of each piece of code (function, loop, etc.) as symbolic expression. This allows to obtain tighter but safe WCET values as the computed symbolic expression can be instantiated for each specific call to the subprogram and nested loops leading to tighter WCET values with a low cost. We use an enhanced symbolic execution method which avoids unfolding complex blocks by evaluating each complex block analytically.

3 Flow analysis approach

In this section, we describe our approach aimed to automatically extract the flow information related to program functionality (bounds on loop iterations, infeasible paths, etc.). We use a data flow analysis approach based on symbolic execution in order to derive values of variables at different points in the program.

3.1 Program representation

We use the control flow graph (CFG) formalism to express the control flow of the program to be analyzed. The source code of the program is decomposed into a set of basic blocks. Two fictitious blocks, label-ed $start$ and $exit$ are added. We assume that all executions of the CFG start at the $start$ block and end at the $exit$ block. Figure 2 illustrates an example of a control flow graph of a program where the C source code is shown in figure 1.

Formally, the program is represented by the graph $G = (B, E)$, where B, the set of the graph nodes, represents the program basic blocks and the set of edges (E) the precedence constraints between the basic blocks.

```
void sort(unsigned int n) {
  unsigned int i, j;

  for (i=0; i<n-1; i++) {
    for (j=i+1; j<n; j++) {
    }
  }
}
```

Figure 1: Example of a C program

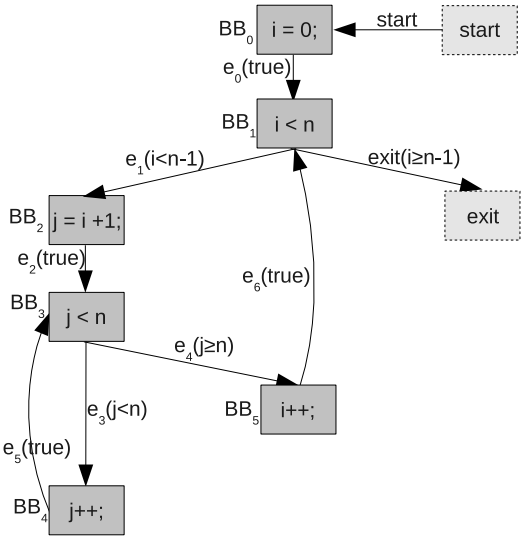

Figure 2: CFG of the C example

3.2 Scopes and scope graphs

In addition, we use the notion of scope where a set of scopes of level l are grouped into a scope of level $l - 1$. Scopes correspond to complex programming language features (loops, conditional statements, functions, modules, etc.). The scope composition starts at the lowest level and may be recursively carried out until the CFG level. Figure 3 illustrates the

[1] Implicit Path Enumeration Techniques.
[2] Integer Linear Programming.

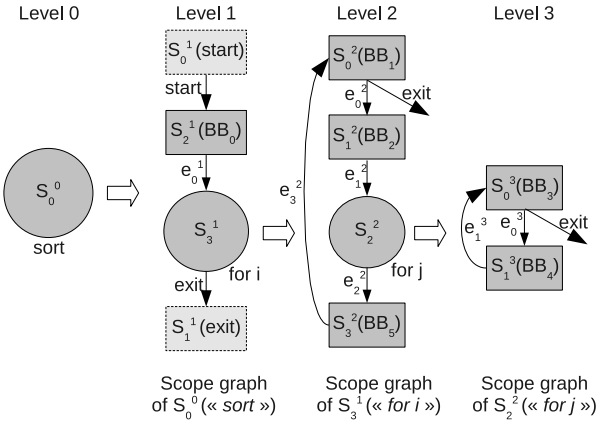

Figure 3: The scope graphs of the example

scope graphs constructed for the C example of the figure 1. A scope S is composed of a set of sub-scopes related by edges, a set of header blocks and one or more exit edges. Each scope S of level l is defined by a *scope graph* describing its structure (figure 3).

Formally, a scope S of level l is defined by the formula 1 and composed of: a number of sub-scopes (S_s); a set of header blocks ($S_s^h \subseteq S$); a set E_s of edges connecting the sub-scopes; and one or more exit edges ($E_s^e \subseteq E_s$).

$$S = \{S_s, S_s^h, E_s, E_s^e\}. \quad (1)$$

The S's sub-scopes set S_s is composed of scopes of higher levels ($m > l$). The set of edges E_s is constructed as follows: each edge of the CFG connecting two nodes belonging to two different sub-scopes s_i and s_j of S_s forms an edge of level l from s_i to s_j. Edges to scopes outside S_s produce edges of *exit* type. Redundant edges are eliminated. In figure 3, in the graph of scope S_3^1 corresponding to the *for i* loop, the edge e_2 connecting the basic blocks BB_2 and BB_3 in the CFG yields the edge e_1^2.

A header block is a basic block executed when the execution flow reaches the scope for the first time. Informally, header blocks correspond to loop and selection condition test blocks. The set of headers of the scope S is denoted by S_s^h. The set of header-blocks of the scope S_3^1 is $\{BB_1\}$ (figures 2 and 3). Notice that a well-structured code yields scopes with one header block. When the execution of the scope is terminated, the control flow leaves the scope through an exit-edge. The set of outgoing edges from a scope S is denoted by E_s^e. The set of exit-edges of the scope S_3^1 is $\{exit\}$ (figure 3). When the execution of a scope must be repeated, this is done by transferring the execution flow to the header block through a back-edge. The set of back-edges of a scope S is denoted by E_s^b. The set of back-edges of the scope S_3^1 is $\{e_3^2\}$ (figure 3).

3.3 Path condition and path action

Each elementary edge e in the CFG is associated a path condition $PC(e)$ which is a Boolean predicate conditioning the execution of that edge with respect to the program state s at the source node of the edge. Likewise, each basic block bb applies a block action $BA(bb)$ which represents the effect of the execution of the sequence of all statements of the block on the program state s (symbolic execution rules). The path action of a path p denoted $PA(p)$ is the sequence of the block action of all blocks constituting that path. Likewise the path condition of a path is the *"logical and"* of the path condition of all edges forming that path.

In order to compute the path action of a path p, we consider the set of program variables assigned in different blocks of the path V_p. Let $BA(bb, v)$ be the function applied by the basic block bb on the variable $v \in V_p$ which represents the effect of the execution of all statements of the block on v. The action applied on v by p ($PA(p, v)$) is the sequence of block action applied by all blocks forming p in the order they appear in p. $PA(p, v) = (BA(bb_0, v); BA(bb_1, v); \ldots; BA(bb_{n-1}, v))$ may be represented by an expression of the form $av+b$ such that a and b are integer constants ($a > 0$). This hypothesis implies that the loop induction variable (ex. i) update statement is of the form $i = a * i + b$.

4 Algebraic evaluation of iterative scopes

As we have seen, a key idea of our approach is the way the iterative scopes are handled. Indeed, iterative scopes are not folded, rather than they are analytically evaluated. To do, the number of iterations of such iterative scope must be algebraically computed and the WCET of that scope is estimated based on the computed number of iterations. Furthermore, the values of assigned variables inside the scope must be computed. In the following, we address those issues in more details.

4.1 Analytic evaluation of the number of iterations of loops

In order to analytically compute an estimate of the number of iterations of a loop path, we define the suite (i_n) as follows:

$$\begin{cases} i_0 = N_1 \in \mathbb{N} \\ i_{n+1} = ai_n + b \quad \forall n \in \mathbb{N} \end{cases} \quad (2)$$

This suite can be redefined as follows:

$$\begin{cases} i_0 = N_1 \\ i_{n+1} = i_n + a^n(b + N_1(a-1)) \quad \forall n \in \mathbb{N} \end{cases} \quad (3)$$

The general term of this suite can be infered using the following relation:

$$i_n = i_0 + \sum_{k=0}^{n-1}[i_{k+1} - i_k] = N_1 + \sum_{k=0}^{n-1}[i_k + a^k(b + N_1(a-1)) - i_k] = N_1 + \sum_{k=0}^{n-1}[a^k(b + N_1(a-1))]$$
$$= \begin{cases} N_1 + (b + N_1(a-1))\frac{a^n - 1}{a-1} & \text{when } a > 1 \\ N_1 + nb & \text{when } a = 1 \end{cases} \quad (4)$$

The number of iterations I is defined by the relation $i_{I-1} \leq N_2$ (the loop limit). That is:

$$\begin{cases} N_1 + (b + N_1(a-1))\frac{a^{I-1} - 1}{a-1} & \leq N_2 \text{ when } a > 1 \\ N_1 + (I-1)b & \leq N_2 \text{ when } a = 1 \end{cases}$$

By knowing that the number of iterations I is a positive integer, we can deduce its value as the greatest integer less than or equal to the right-hand side of the inequality, which yields:

$$I = \begin{cases} \lfloor \log_a(1 + \frac{(N_2 - N_1)(a-1)}{b + (a-1)N_1}) \rfloor + 1 & \text{when } a > 1 \\ \lfloor \frac{N_2 - N_1}{b} \rfloor + 1 & \text{when } a = 1 \end{cases} \quad (5)$$

For each elementary path condition expression e of the form $i\ op\ expr$, the following parameters are evaluated:

- *Interval type:* the interval is qualified as raised if op is "$\leq$", constant if op is "$=$" and undervalued if op is "$\geq$". Expressions containing "$<$" and "$>$" operators are converted to expressions containing "$\leq$" and "$\geq$" operators exclusively.
- *Direction:* if the variable i in increased in the path action ($PA(p)$), the direction is positive and negative if i is decreased in $PA(p)$. If i is never updated along with the path, the direction is constant. The direction is determined by the expression $a * i + b$ and is positive when the expression $(a-1)N_1 + b$ is positive and negative when it is negative and constant when it is 0.

The direction and the interval type are used to determine if a loop is empty before applying the formula 5. Thus, in case when the interval is raising and the direction is positive the number of iterations is given by the formula 5 if $N_1 \leq N_2$ and 0 otherwise. Likewise, when the interval is undervalued the number of iterations is ∞ when $N_1 \geq N_2$ and 0 otherwise. For example, in the "$for\ i$" loop of the sort example ($i = 0$, $i < n-1$, $i = i + 1$), $a = 1$, $b = 1$, $N_1 = 0$. The expression $(a-1)N_1 + b$ is evaluated to 1. The direction is then positive and the interval type is undervalued. Therefore the loop is not empty. The computed number of iterations and the WCET are expressed as symbolic expressions using the guarded expressions mechanism presented in the next section. The provided symbolic guarded expressions will be instantiated when numerical values are provided.

Complex path condition expressions are evaluated by decomposing them recursively into elementary Boolean expressions related by logical operators ($e1 \wedge e2$ or $e1 \vee e2$). Each elementary conditional expression e is of the form $i\ op\ expr$, where op is a relational operator and $expr$ is an integer valued expression. Each expression is then analytically evaluated as described above. Then the number of iterations I^e related to e and the resulting symbolic state S^e are calculated. The total number of iterations is determined from the number of iterations of all subexpressions of e as follows: $I^{e_1 \vee e_2} = \max(I^{e_1}, I^{e_2})$ and $I^{e_1 \wedge e_2} = \min(I^{e_1}, I^{e_2})$.

4.2 Updating variables assigned inside iterative scopes

In this subsection, we address the algebraic evaluation of variables assigned inside the iterative scope body. For this purpose, we assume a general form on the assignment statements of variables inside iterative scopes that can be updated analytically.

$$\begin{aligned} s &= c * s + d * i + e; \\ i &= a * i + b; \\ p &= k; \\ q &= f(v_1, v_2, \cdots); \end{aligned} \quad (6)$$

Such that: a, b, c, d, e, k are expressions which remain constant within the iterative scope body. $a, b \in \mathbb{N}$. f is a function. $v_1, v_2, \cdots$ are variables which may be assigned inside the loop scope and different from p.

In order to analytically compute the values of the variables at the end of iterative scope, we first consider the following suite:

$$\begin{cases} u_0 \in \mathbb{R} \\ u_{n+1} = gu_n + rt^n + s \quad \forall n \in \mathbb{N} \end{cases} \quad (7)$$

We can prove that the general term of this suite can

be expressed by the following:

$$u_n = g^n u_0 + r\sum_{k=0}^{n-1} g^k t^{n-k-1} + s\sum_{k=0}^{n-1} g^k \qquad \forall n > 0 \tag{8}$$

$$= \begin{cases} g^n u_0 + rt^{n-1}\frac{1-(\frac{g}{t})^n}{1-\frac{g}{t}} + s\frac{g^n-1}{g-1} & \\ \qquad = g^n u_0 + r\frac{t^n-g^n}{t-g} + s\frac{g^n-1}{g-1} & \text{when } p_1 \\ g^n u_0 + rng^{n-1} + s\frac{g^n-1}{g-1} & \text{when } p_2 \\ u_0 + r\frac{t^n-1}{t-1} + sn & \text{when } p_3 \\ u_0 + rn + sn & \text{when } p_4 \\ g^n u_0 + s\frac{g^n-1}{g-1} & \text{when } p_5 \\ s & \text{when } p_6 \\ u_0 + sn & \text{when } p_7 \end{cases} \qquad \forall n > 0 \tag{9}$$

with:

$$\begin{aligned} p_1 &= (t \neq 0 \wedge g \neq 1 \wedge g \neq t) \\ p_2 &= (g = t \wedge g \neq 0 \wedge g \neq 1) \\ p_3 &= (g = 1 \wedge t \neq 0 \wedge t \neq 1) \\ p_4 &= (g = t = 1) \\ p_5 &= (t = 0 \wedge g \neq 1 \wedge g \neq 0) \\ p_6 &= (g = t = 0) \\ p_7 &= (g = 1 \wedge t = 0) \end{aligned}$$

The evaluation of the two variables s and i at the end of the iterative scope, may be done by means of the two suites (s_n) and (i_n) defined as follows:

$$\begin{cases} s_0 & \in \mathbb{R}. \\ i_0 & \in \mathbb{Z}. \\ s_{n+1} = cs_n + di_n + e & \\ i_{n+1} = ai_n + b & \end{cases} \tag{10}$$

Using the formulas 7 and 9 ($g = a$, $r = 0$, $s = b$), we can prove easily that the value of the variable i at the iteration n may be given by the following expression:

$$i_n = \begin{cases} b & \text{when } a = 0 \\ i_0 + nb & \text{when } a = 1 \\ a^n i_0 + b\frac{a^n-1}{a-1} & \text{when } a > 1 \end{cases} \tag{11}$$

The value of the variable s can be infered as follows:

$$s_{n+1} = c^{n+1}s_0 + di_0 A_n + B_n \tag{12}$$

Such as:

$$\begin{aligned} A_0 &= 1 \\ A_{n+1} &= cA_n + a^{n+1} = cA_n + aa^n \end{aligned}$$

And:

$$\begin{aligned} B_0 &= e \\ B_{n+1} &= cB_n + \sum_{k=0}^{n} da^k b + e = cB_n + db\sum_{k=0}^{n} a^k + e \\ &= cB_n + db\frac{a^{n+1}-1}{a-1} + e \end{aligned}$$

Again using the formulas 7 and 9 ($g = c$, $r = t = a$, $s = 0$), we can prove easily that the value of the variable A at the iteration n may be given by the following expressions:

$$A_n = \begin{cases} c^n(n+1) & \text{when } c = a \\ \frac{a^{n+1}-c^{n+1}}{a-c} & \text{when } c \neq a \end{cases}$$

Finally still using the formulas 7 and 9 ($g = c$, $r = \frac{dba}{a-1}$, $t = a$, $s = e - \frac{db}{a-1}$), we can deduce the value of B_n as follows:

$$B_n = \begin{cases} c^n e + \frac{dba}{a-1}\frac{a^n-c^n}{a-c} + (e - \frac{db}{a-1})\frac{c^n-1}{c-1} & \\ \qquad \text{when } c \neq a \wedge c \neq 1 \wedge a \neq 1 \wedge a \neq 0 & \\ c^n e + \frac{dbnc^n}{c-1} + (e - \frac{db}{c-1})\frac{c^n-1}{c-1} & \\ \qquad \text{when } c = a \wedge c \neq 1 \wedge c \neq 0 & \\ e + \frac{dba(a^n-1)}{(a-1)^2} + (e - \frac{db}{a-1})n & \\ \qquad \text{when } c = 1 \wedge a \neq 1 \wedge a \neq 0 & \\ c^n e + (e+db)\frac{c^n-1}{c-1} \text{ when } a = 0 \wedge c \neq 1 \wedge c \neq 0 & \\ e + db \text{ when } c = a = 0 & \\ e + n(e+db) \text{ when } c = 1 \wedge a = 0 & \end{cases}$$

$$\forall n > 0$$

In case of $a = 1$, we have:

$$\begin{aligned} B_{n+1} &= cB_n + db\sum_{k=0}^{n} 1^k + e = cB_n + (n+1)db + e \\ B_0 &= e \end{aligned}$$

Therefore, we can easily demonstrate[3] that:

$$\begin{aligned} B_n &= e\sum_{k=0}^{n} c^k + db\sum_{k=0}^{n-1}(n-k)c^k \\ &= \begin{cases} \frac{c^{n+1}}{c-1}(e + \frac{db}{c-1}) - \frac{e}{c-1} - \frac{db}{c-1}(\frac{c}{c-1} + n) & \\ \qquad \text{when } a = 1 \wedge c \neq 1 & \\ e + n(e + \frac{n+1}{2}db) \text{ when } c = a = 1 & \end{cases} \\ &\forall n > 0 \end{aligned}$$

[3] Using for instance the recurrence proof.

We then simplify the formula which gives:

$$B_n = \begin{cases} c^n e + \frac{dba}{a-1}\frac{a^n-c^n}{a-c} + (e - \frac{db}{a-1})\frac{c^n-1}{c-1} \\ \quad \text{when } c \neq a \wedge c \neq 1 \wedge a \neq 1 \wedge a \neq 0 \\ c^n e + \frac{dbnc^n}{c-1} + (e - \frac{db}{c-1})\frac{c^n-1}{c-1} \\ \quad \text{when } c = a \wedge c \neq 1 \wedge c \neq 0 \\ e + \frac{dba(a^n-1)}{(a-1)^2} + (e - \frac{db}{a-1})n \\ \quad \text{when } c = 1 \wedge a \neq 1 \wedge a \neq 0 \\ \frac{c^{n+1}}{c-1}(e + \frac{db}{c-1}) - \frac{e}{c-1} - \frac{db}{c-1}(\frac{c}{c-1} + n) \\ \quad \text{when } a = 1 \wedge c \neq 1 \\ e + n(e + \frac{n+1}{2}db) \text{ when } c = a = 1 \end{cases}$$
$$\forall n > 0 \tag{13}$$

To summarize, variables assigned within iterative scopes are evaluated in many manners following the way they are assigned inside the scope i.e. the assignment statement type:

- Statements of type s in the formula 6. In this case, the variable is evaluated using the formula 12.
- Statements of type i in the formula 6. In this case, the variable is evaluated using the formula 11.
- Statements of type q in the formula 6. In this case, the variable value is given by replacing in the expression of the function f the variables $v_1, v_2, \cdots$ by their final values.
- Statements of type p in the formula 6. The variable remains unchanged ($p = k$).

4.3 Scope-based symbolic execution

Symbolic execution consists of using symbols instead of numbers as input values for a program execution. The program (especially loops) is then rolled out until it terminates which represents a symbolic execution. The values of program variables are represented with symbolic expressions and the program output values are expressed as function of the program input symbols [2].

We use an enhanced symbolic execution approach which avoids rolling out loops reducing thus the complexity of the symbolic execution. In our approach, a symbolic execution state is represented by a triple $< V, PC, SP >$, where:

- V is the set of pairs $< v, e >$, which holds all combinations of variables values that are possible at the control point. $V = \{< v_1, e_1 >, < v_2, e_2 > , \ldots, < v_n, e_n >\}$ denotes a symbolic state where the variables $v_1, v_2, \ldots, v_n$ have been assigned the expressions $e_1, e_2, \ldots, e_n$;
- PC is the path condition expressing the conditions under which that path is taken;
- SP is the instruction pointer referring to the next sub-scope in the scope graph to execute.

Initially, the input parameters and all the other variables are initialized using symbols. SP points to the header block of the scope and PC is set to *true*. Symbolic execution of a program takes a symbolic state and a rule which corresponds to the block action of the sub-scope referred to by SP and returns the symbolic states resulting from the execution of the rule. When SP points to the header block for the second time, the defined loop path is analytically evaluated following the process described in subsection 4.1, which reduces the complexity of the approach. Furthermore, only a subset of the symbolic states set of the program are computed. For this purpose, we associate to each scope a set of control points which correspond to its terminal symbolic states set. A control point state is composed of the set of the scope variables with their initial values V_{in} as well as the corresponding terminal state. The evaluation of a scope yields the set of its control points and a symbolic WCET expression associated to each exit control point. Figure 4 shows the scope-based symbolic execution of the scope S_2^2. A WCET value based on the number of iterations is computed and associated to each symbolic state. The number of iterations, WCET expressions and the variable values are computed using the formulas 5, 11 and 12 discussed in the previous subsection.

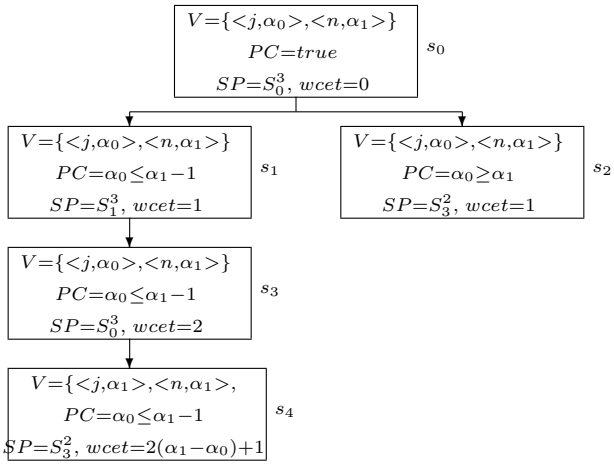

Figure 4: Evaluation of the scope S_2^2

In order to understand how the non rectangular loops problem is handled by the block-based symbolic execution, we consider the evaluation of the scope S_3^1 (the "$for\ i$" outer loop) shown on the figure 5. The application of the scope S_2^2 on the state s_3 generates only one state (s_4) because on the figure 4, applying the control point corresponding to the state s_2 leads to a false path condition (a false path) since $\alpha_0 \leq \alpha_2 - 2 \Rightarrow \alpha_0 \geq \alpha_2 - 1$ leads to a $false$ value. The WCET expression in the state s_7 is computed as follows:

$$wcet(s_7) = \sum_{x=0}^{\alpha_2-\alpha_0-2} (2\alpha_2 - 2\alpha_0 - 2x + 2) = (\alpha_2 - \alpha_0)^2 + 3(\alpha_2 - \alpha_0) - 4$$

The final WCET of the scope S_0^0 corresponding to the "$sort$" function is given by $n^2 + 3n - 2$ rather than $(n-1)(2n+1)+2) = 2n^2-n-1$ induced by the methods that don't handle rectangular loops, which reduces the estimated WCET value by a factor of 2.

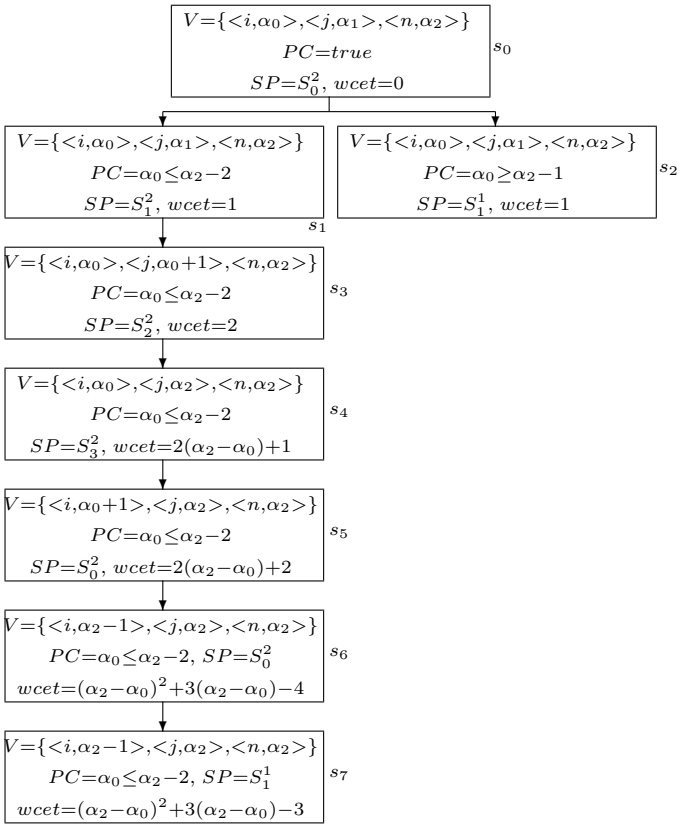

Figure 5: Evaluation of the scope S_3^1

5 Handling symbolic expressions

The block-based symbolic execution proceeds by evaluating each scope independently and uses symbols as input values for all unknown variable values. Consider the evaluation of of the scope S_2^2 presented in the figure 4.

The direction value i.e. the expression $(a-1)N_1 + b$ in evaluating s_3 is 1. If the direction is evaluated to a symbolic expression, the number of iterations and WCET expressions will be conditioned by the effective value of the direction parameter. For instance consider that the assignment statement of j is $j = 2*j+1$ in the loop body, ($a = 2, b = 1, N_1 = \alpha_0$). Then the direction evaluates to $\alpha_0 + 1$. Therefore, the number of iterations I in s_4 is given by: $\lfloor \frac{\alpha_1-\alpha_0-1}{2} \rfloor + 1$ when the direction is positive i.e. $\alpha_0 + 1 > 0$ since $N_1 \leq N_2 = \alpha_0 \leq \alpha_1 - 1$ is always true (implied by the path condition of the state s_3). Likewise when the direction is negative i.e. $\alpha_0 + 1 < 0$, I evaluates to ∞ (since we always have $N_1 \leq N_2$). And when the direction is constant i.e. $\alpha_0 + 1 = 0$ the number of iterations is ∞.

In order to express such conditional expressions when computing the number of iterations and variable values expressions, we use the symbolic guarded expression mechanism [1]. Hence, the number of iterations of the scope S_2^2 can be expressed by the following guarded expression: $[\alpha_0 > -1](\lfloor \frac{\alpha_1-\alpha_0-1}{2} \rfloor + 1) + [\alpha_0 \leq -1]\infty$

Guarded expressions constitute a powerful mechanism which is used to evaluate symbolically the WCET of a piece of code using conditional expressions. A guarded expression denoted $[c]e$ is evaluated as follows:

$$[c]e = \begin{cases} e & if\ c \\ 0 & otherwise \end{cases} \quad (14)$$

However, handling symbolic guarded expressions implies the use of simplification properties which may be different from the standard algebra. Hence, in addition to standard properties (commutativity, associativity, distributivity, ...), we adopt some other properties to simplify arithmetic guarded expressions.

$$ge\ op\ [false]e = ge\ with\ op\ \in \{+, \times\}$$
$$ge + [c]0 = ge$$
$$[c]e \times [c]1 = [c]e$$
$$ge \times [c]0 = [true]0$$
$$-[c]e = [c](-e)$$
$$[c]e_1 + [c]e_2 = [c](e_1 + e_2)$$
$$[c]e_1 \times [c]e_2 = [c](e_1 \times e_2)$$
$$\sum_{i=[c]e_1}^{[c]e_2} [c]e = [c] \sum_{i=e_1}^{e_2} e$$

These properties can be easily demonstrated. The last property is particularly used when evaluating analytically nested loops and generating new symbolic states as shown by the example presented at the end of the previous subsection.

6 Conclusion

Real-time systems must be predictable in time and memory in order avoid undesirable consequences when the temporal constraints are not respected especially in critical environments. WCET analysis has

became an important research field in the area of embedded systems in which the emphasis is on reducing the gap between the theoretical worst case values and the observed ones.

WCET analysis may be done statically on the program source or object code which avoids dealing with the program input data and target hardware platforms, but results in overestimated values. Therefore, techniques allowing to tighten the WCET estimates are required. However, these techniques are complex because they deal with program semantics.

We proposed a practical symbolic approach aimed to automatically extract flow information related to program semantics which will be used to tighten the WCET estimates. The method presents a reduced complexity, at least on the theoretical level, in terms of time and memory by avoiding unfolding iterative scopes. Moreover, the approach provides tight WCET values since it provides WCET of the system components (functions, etc.) as symbolic expressions function of the component input parameters, which allows to instantiate them with numerical values for each call of the component. Furthermore, the approach handles non rectangular loops and loops with multiple exit conditions and eliminates implicitly most of the infeasible paths. Indeed, non rectangular nested loops and infeasible paths constitute an important source of overestimation in WCET analysis.

We are implementing and evaluating a prototype of the method. Furthermore, we plan to extend the expression used to evaluate loops to Presburger formulas and use the results obtained on those formulas.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[2] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using symbolic execution for verifying safety-critical systems. In *8th European software engineering conference*, pages 142–151, New York, USA, 2001. ACM Press.

[3] A. Ermedahl. *A modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[4] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1(2):61–74, 1998.

[5] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, , and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18(2-3):129–156, May 2000.

[6] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-time Systems, La Jolla, California*, June 1995.

[7] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003*, pages 99–102, Polytechnic Institute of Porto, Portugal, 2003.

[8] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[9] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Journal of Real-Time Systems*, 1(2):160–176, September 1989.

[10] P. Puschner and A. V. Schedl. Computing maximum task execution- a graph-based approach. *Real-Time Systems*, 13(1):67–91, July 1997.