Automatic flow analysis using symbolic execution and path enumeration

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. Indeed, these information is at the origin of important overestimation of the WCET. 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 number of loop iterations is expressed as summations function of the loop bounds. The flow analysis approach combines symbolic execution and path enumeration in order to avoid unfolding loops performed by symbolic execution-based approaches while providing tight and safe WCET estimate.

Keywords: hard real-time systems, static WCET analysis, automatic parametric flow analysis, block-based symbolic execution, path enumeration.

1. Introduction

Today, real-time and embedded systems occupy a dominating place in many areas of industrial and economic fields (aircraft avionics, space, manufacturing process control, cars, domestic equipments field, etc.). Real-time systems require time constraints which must be met in order to avoid unpleasant consequences when they are not respected, especially in hard real-time systems. Therefore, the timing analysis of software is needed in order to verify the temporal correctness of these systems. This paper is devoted to the WCET analysis where the purpose is to find upper bounds of the execution time of systems on hardware platforms.

Static WCET analysis performs a high level static analysis of the source code in order to avoid working on the input data of programs which is intractable for complex systems. The result is an upper bound estimate of the program execution time, rather than the WCET exact values. Therefore, the WCET analysis must guarantee two main properties: *safeness* and *tightness* of the provided values in order to reduce the system cost.

WCET analysis proceeds usually in three steps: 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 though they may differ from one execution of the basic block to another. 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. This type of flow information is complex to automate and therefore is generally provided by the programmer as annotations [7, 3].

The remainder of this paper is organized as follows: in the next section we review some related WCET analysis methods. Section 3 presents some concepts used by the flow analysis approach. Section 4 describes the proposed block-based symbolic execution method and discusses some special cases. Finally, we conclude the paper and present some perspective issues.

2. Related work

Static WCET analysis approaches may be classified into three main categories: path-based, tree-based and IPET[1] approaches. Path-based approaches proceed by explicitly enumerating the set of the program execution paths [10, 2, 9]. The main drawbacks of those approaches lie in the important number of the generated program paths which scales exponentially with the program size. [10] uses regular expressions to express and enumerate all possible execution

[1] Implicit Path Enumeration Techniques.

Proceedings of the 2006 International Conference on Parallel Processing Workshops (ICPPW'06)
0-7695-2637-3/06 $20.00 © 2006 IEEE

paths of a program. [2] proposes a path-based WCET analysis approach to symbolically compute the WCET of program parts as symbolic expressions. Unlike the path-based approaches, IPET methods do not enumerate all program paths, but rather consider that they implicitly belong to the problem solution. The problem of the WCET estimation may then be converted to the one of solving an ILP[2] problem [7, 11]. [4] describes an approach to WCET estimate calculation, in which the scope graph model is used. Scopes correspond to complex language features (loops, if statements, etc.) and allow expressing the dynamic behavior of the program by means of a flow fact language.

In those approaches, the programmer is involved 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 and may lead to unpleasant consequences in the case of incorrect information. Abstract interpretation and symbolic execution may be used to automatically infer a subset of the flow information related to program functionality. In [5], an interval-based abstract interpretation method is proposed. The approach allows to automatically extract flow information related to program semantics by rolling out the program until it terminates. However rolling out the program, especially loops, is very costly in time and memory required by the important number of generated states. [8] describes a method for automatic parametric WCET analysis. The approach is based on abstract interpretation and a symbolic method to count integer points in polyhedra. A symbolic ILP technique is used to solve the WCET calculation based on IPET. The approach seems complex in practice and the author uses the interval-based abstract interpretation proposed in [5]. In [6], an approach for determining loop bounds 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 handle only loops with the induction variable being increased by a constant amount between two successive iterations. Symbolic execution is another technique for automatically extracting the flow information related to program functionality. The program is rolled out which allows to determine the values of variables as expressions of the program inputs. Symbolic execution-based approaches are very used in the formal verification of safety critical software. However, their complexity renders them less attractive for WCET analysis, especially for long and complex programs.

Our aim is to develop a practical approach which automatically extract flow information related to program functionality and compute a safe and tight upper bound on the program WCET with a lower cost. We propose a hybrid approach based on symbolic execution and path enumeration. Loops are not unfolded rather than a path analysis is performed on each loop block.

[2] Integer Linear Programming.

3. Flow analysis concepts

In the following, we present a set of concepts used by our flow analysis approach.

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. A basic block is a set of instructions with a single entry point and a single exit point [1]. The entry point is situated at the beginning of the block and the exit point at its end. Describing a program using the control flow graphs formalism consists of building all possible successions of the basic blocks constituting the program. Two fictitious blocks, labeled $start$ and $exit$ are added. We assume that all executions of the CFG start at the $start$ block and end at the $exit$ block. Figure 1-b illustrates an example of a control flow graph of a program where the C source code is shown in figure 1-a. 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.

3.2. Blocks and block graphs

We use the notion of block where a set of blocks of level l are grouped into a block of level $l-1$. Complex blocks correspond to complex programming language features (loops, conditional statements, functions, modules, etc.). The block composition starts at the lowest level and may be recursively carried out until the CFG level. Figure 2 illustrates the block graphs constructed for the C example of the figure 1. Formally, a block b of level l is defined by the formula 1 and composed of: a number of sub-blocks (B_b); a set of header blocks ($B_b^h \subseteq B$, $B_b^h \subseteq B_b$); a set E_b of edges connecting the sub-blocks; and one or more exit edges ($E_b^e \subseteq E_b$).

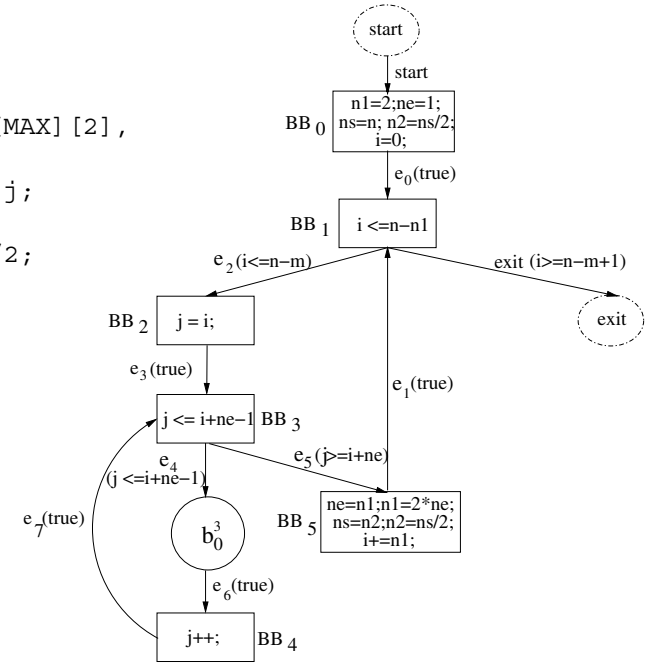
$$b = \{B_b, B_b^h, E_b, E_b^e\}. \quad (1)$$

Each block b of level l is represented by a *block graph* describing its structure. The b's blocks set B_b is composed of blocks of higher levels. The set of edges E_b is constructed as follows: each edge of the CFG connecting two nodes belonging to two different blocks b_i and b_j of B_b forms an edge of level l from b_i to b_j. Edges to blocks not in B_b produce edges of *exit* type. Redundant edges are eliminated. In

```
void fft(unsigned int n, double f[MAX][2],
double t[MAX][2]) {
  unsigned int ne, n1, ns, n2, i, j;

  ne = 1; n1 = 2; ns = n; n2 = ns/2;
  for (i=0; i<=n-n1; i+=n1) {
    for (j=i; j<=i+ne-1; j++) {
      update t[j] from f
    }
    ne = n1; n1 = 2*ne;
    ns = n2; n2 = ns/2;
  }
}
```

a) C source code



b) Control flow graph

Figure 1. Example of a C program.

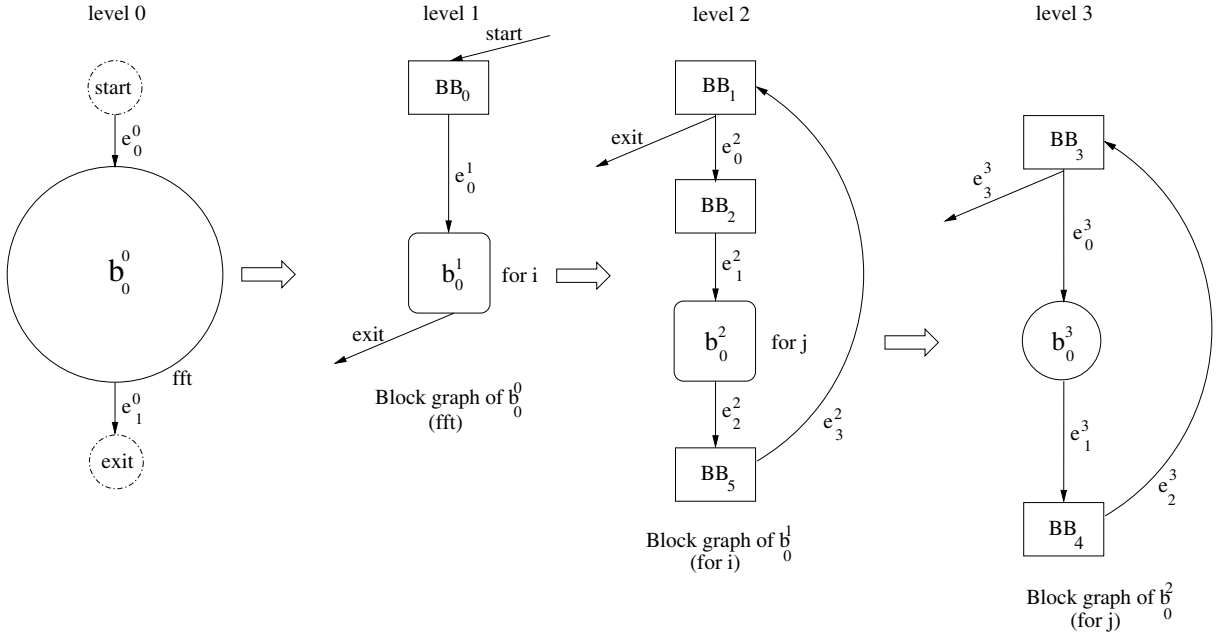


Figure 2. The block graphs of the example.

figure 2, in the graph of block b_0^1 (*for* loop), the edge e_3 connecting the basic blocks BB_2 and BB_3 in the CFG yields the edge e_1^2.

3.3. Paths and iterations

A path in a block graph b is a sequence of edges in E_b where the end-node of each edge is the starting-node of the next edge in the path. A path p can be decomposed into a sequence of sub-paths $p_0, p_1, \ldots, p_{k-1}$ such that $p = (p_0, p_1, \ldots, p_{k-1})$.

Each complex block b is characterized by the set of its block-paths. A *block-path* of b is a *one-iteration-path* in b. We distinguish two types of block-paths: *exit-paths* and *loop-paths*. An exit-path in b is a *block-path* starting at the header block of b and ending by an exit-edge of b. Likewise, a loop-path in b is a *block-path* starting at the header block of b and ending by a back-edge of b. In figure 2 and table 1, the block b_0^1 has one exit path p_0^2 composed of the only edge *exit* and one loop-path p_1^2 composed of edges e_0^2, e_1^2, e_2^2 and e_3^2.

Each block is executed a number of times (0 or more). We use the notion of *iteration* to denote an execution of a block. An iteration of a block b is defined as an execution of a *block path* of b.

4. Flow analysis approach

In this section, we describe our method aimed to automatically extract the flow information related to program functionality. We use a data flow analysis approach in order to derive values of variables at different points in the program. The approach combines symbolic execution with path enumeration. The flow analysis is performed for each block without unfolding iterative blocks. Rather than, the number of times the blocks are executed is analytically computed which reduces the complexity of the approach. In our approach, only a subset of the symbolic states set of the program are computed[3]. Exit points of a block are the starting points of its exit edges.

4.1. 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 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 p. Likewise, the path condition of a path is the *"logical and"* of the path condition of all edges forming that path ($PC(p) = PC(e_0) \wedge PC(e_1) \ldots \wedge PC(e_{n-1})$). These properties may be applied on sub-paths as well. Let p be a path composed of a sequence of sub-paths ($p = (p_0, p_1, \ldots, p_{k-1})$), then $PC(p) = PC(p_0) \wedge PC(p_1) \ldots \wedge PC(p_{k-1})$ and $PA(p) = (PA(p_0); PA(p_1); \ldots; PA(p_{k-1}))$.

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 ($a > 0$). That means that the loop induction variable (ex. i) update statement is of the form $i = a * i + b$.

4.2. Evaluating block paths

The evaluation of a path p involves the path condition expression $PC(p)$ and the path action $PA(p)$. $PC(p)$ is decomposed into elementary Boolean expressions related by logical operators. Each elementary expression e is of the form i *op expr*, where *op* is a relational operator and *expr* is an integer valued expression. Hence, each elementary expression e defines a bound of an integer interval. The other bound is determined by the initial value of i (ex. $i \leq 100$, the interval is $[\alpha_0, 100]$). Then, the following parameters are evaluated for each expression:

- *Interval type:* the interval is qualified as raised if *op* is "$<$" or "$\leq$", constant if *op* is "$=$" and undervalued if *op* is "$>$" or "$\geq$".
- *Direction:* if the variable i in increased in $PA(p)$, 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 null.

A preliminary step consists of checking for empty and unbounded loop paths using the defined *interval type* and the *direction*. Then, the loop parameters (a, b, N_1 and N_2) are determined and passed to the algebraic module in order to compute the number of iterations I_p^e and the resulting symbolic state s_p^e (formula 2).

In order to compute the number of iterations of a loop path p, we define the following suite (v_n), the suite of the values taken by the loop induction variable:

$$\begin{cases} v_0 & = & N_1 \\ v_{n+1} & = & av_n + b \quad \forall n \in \mathcal{N}. \end{cases}$$

The number of iterations I is defined by the formula $N_2 - N_1 \geq \sum_{n=0}^{I-2} s_n$. N_1 is the initial value of the loop

[3] Symbolic states at the entry point and at each exit point of the blocks.

induction variable, N_2 is the loop limit and $s_n = v_{n+1} - v_n$. The right-hand side of the inequality would be the greatest integer less than or equal the expression $N_2 - N_1$.

$$I = \begin{cases} \lfloor \log_a(1 + \frac{(N_2-N_1)(a-1)}{b+(a-1)N_1}) \rfloor + 1 & when \quad N_2 > N_1 \\ \wedge\, a > 1 \wedge N_1(1-a) \neq b & \\ \lfloor \frac{N_2-N_1}{b} \rfloor + 1 & when \quad a = 1 \\ \infty & when \quad b = N_1(1-a) \end{cases} \tag{2}$$

The number of iterations is then given by the formula 2. When $b = N_1(1-a)$, the induction variable v would have the same value during the different iterations $v_n = N_1 \; \forall n \geq 0$. Therefore, the number of iterations is infinite ($I = \infty$).

The final number of iterations of p is determined from the number of iterations of all elementary expressions of $PC(p)$ as follows: $I_p^{e_1 \vee e_2 \vee \cdots} = \max(I_p^{e_1}, I_p^{e_2}, \ldots)$ and $I_p^{e_1 \wedge e_2 \wedge \cdots} = \min(I_p^{e_1}, I_p^{e_2}, \ldots)$.

4.3. Block-based symbolic execution

The block-based symbolic execution algorithm is performed in a post-order manner i.e. the blocks of level $l+1$ are evaluated before the blocks of level l, starting by the top level blocks (most inner blocks). Evaluating a block b is achieved in three steps:

a- Path information In this step, the information characterizing the block paths is determined. The block is characterized by the set of its block paths ($block\_paths(b)$). For each path, the following informations are determined: path type (*loop path* or *exit path*); the set of edges forming the path; path condition; path action; and finally for exit paths, the edge of the enclosing block graph on which the flow will go after taking that path. The starting point of that edge constitutes an *exit point* of the block. Table 1 summarizes the parameters calculated for the program of the figure 1 where the block graphs are shown in figure 2.

b- Block evaluation The symbolic execution of a block is performed by computing the number of times each *block-path* $p \in block\_paths(b)$ is executed, starting by a symbolic state in which all variables used in b are assigned symbols. For evaluating each path p, a symbolic state in which the path condition corresponds to $PC(p)$ is generated. Each state is evaluated i.e. by applying a path on that state, as indicated in the step c and one or more new states are generated. The symbolic execution of the block is completed when all non-terminal states are evaluated. This step yields the number of iterations of the block I_b and the set of the block exiting symbolic states S_b.

c- Path evaluation Evaluating a *block-path* p takes a symbolic state s and the path action $PA(p)$ and performs an algebraic evaluation of the path using the formula 2 in the case of loop paths. The evaluation operation produces the number of iterations of that path I_p and the resulting symbolic states S_p.

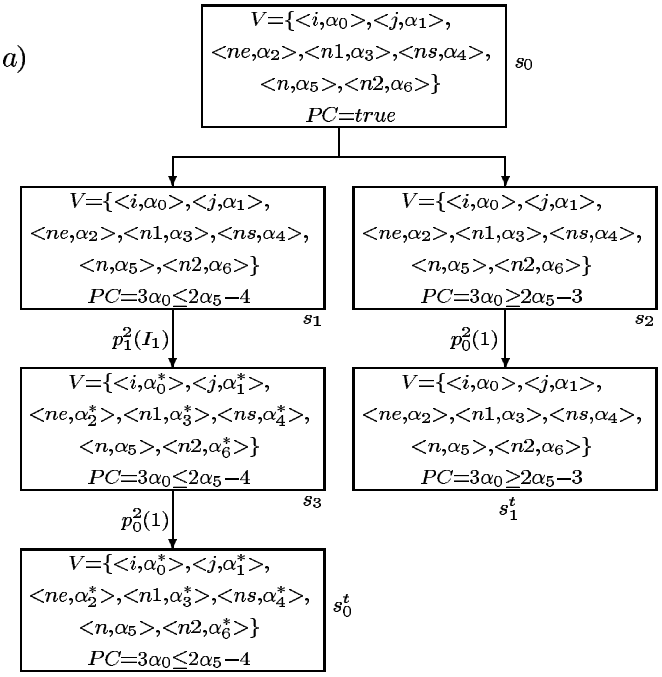


Figure 3. Block-based symbolic execution of the block b_0^1

Figure 3 shows the block-based symbolic execution of the block b_0^1. Edges are annotated by the path applied on the symbolic state of the starting node and the resulting number of iterations. The block paths set is $P = \{p_0^2, p_1^2\}$. Only p_1^2 is a loop path. $PA(p_1^2) = (j \leftarrow i; PA(b_0^2); ne \leftarrow n1; n1 \leftarrow 2 \times ne; ns \leftarrow ns/2; i \leftarrow i + n1)$. There is one expression in $PC(p_1^2)$ $e = i \leq n - n_1$, in which n_1 is non-constant. Our algebraic tool must reevaluate e in order to obtain a constant right-hand expression and $PA(p_1^2)$ in order to apply the formula 2. e is then reevaluated to $3i \leq 2n - 4$ and $PA(p_1^2, i)$ to $i \leftarrow 2i - \alpha_0 + 2\alpha_3$. The evaluation of the path in relation to s_1 and e yields I_1 and the new state s_2. The formula 2 is used for this purpose ($a = 2$, $b = 2\alpha_3 - \alpha_0$, $N_1 = \alpha_0$ and $N_2 = \lfloor \frac{2\alpha_5 - 4}{3} \rfloor$). $I_1 = \lfloor \log_2(1 + \frac{\frac{2\alpha_5-4}{3} - \alpha_0}{2\alpha_3}) \rfloor + 1$. The number of iterations of b_0^2 is updated as well, which evaluates to $\sum_{i=1}^{I_1} i[i \rightarrow 2i] = 2^{I_1} - 1$.

The suite defined in subsection 4.2 is again used to compute the values of the program variables in the new state s_1 ($v = N_1 + \sum_{n=0}^{I-1} s_n$). Hence i gets the new value $\alpha_0^* = \sum_{k=0}^{I_1-1} [2^k(2\alpha_3 - \alpha_0 - \alpha_0) + 2^{k+1}\alpha_0] + \alpha_0 =$

Block	Path	Type	Path composition	Path condition	Path action	Edge
b_0^2	p_0^3	$exit$	$(e_3^3) = (BB_3)$	$j \geq i + ne$		e_2^2
	p_1^3	$loop$	$(e_0^3, e_1^3, e_2^3) = (BB_3, b_0^3, BB_4)$	$j \leq i + ne - 1$	$(PA(b_0^3); j \leftarrow j + 1)$	
b_0^1	p_0^2	$exit$	$(exit) = (BB_1)$	$i \geq n - n1 + 1$		$exit$
	p_1^2	$loop$	$(e_0^2, e_1^2, e_2^2, e_3^2) = (BB_1, BB_2, b_0^2, BB_5)$	$i \leq n - n1$	$(j \leftarrow i; PA(b_0^2); ne \leftarrow n1; n1 \leftarrow 2 \times ne; ns \leftarrow ns/2; i \leftarrow i + n1)$	
b_0^0	p_0^1	$exit$	$(start, e_0^1, exit) = (BB_0, b_0^1)$		$(ne \leftarrow 1; n1 \leftarrow 2; ns \leftarrow n; n2 \leftarrow n/2; i \leftarrow 0; PA(b_0^1))$	e_1^0
top	p_0^0	$exit$	$(e_0^0, e_1^0) = (start, b_0^0, exit)$	$true$	$(PA(b_0^0))$	

Table 1. Path definition and parameters of the program.

Block	V_{in}	edge	V_{out}	PC	Nb. Iter
b_0^2	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >\}$	e_2^2	$\{< i, \alpha_0 >, < j, \alpha_0 + \alpha_2 >, < ne, \alpha_2 >\}$	$true$	α_2
b_0^1	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >, < n1, \alpha_3 >, < ns, \alpha_4 >, < n, \alpha_5 >, < n2, \alpha_6 >\}$	$exit$	$\{< i, \alpha_0^* >, < j, \alpha_1^* >, < ne, \alpha_2^* >, < n1, \alpha_3^* >, < ns, \alpha_4^* >, < n, \alpha_5 >, < n2, \alpha_6^* >\}$	$true$	$\lfloor \log_2(1 + \frac{\frac{2\alpha_5 - 4}{3} - \alpha_0}{2\alpha_3}) \rfloor + 1$
b_0^0	$\{< i, \alpha_0 >, < j, \alpha_1 >, < ne, \alpha_2 >, < n1, \alpha_3 >, < ns, \alpha_4 >, < n, \alpha_5 >, < n2, \alpha_6 >\}$	e_1^0	$\{< i, \alpha_0^* >, < j, \alpha_1^* >, < ne, \alpha_2^* >, < n1, \alpha_3^* >, < ns, \alpha_4^* >, < n, \alpha_5 >, < n2, \alpha_6^* >\}$	$true$	1
top	$\{< i, u >, < j, u >, < ne, u >, < n1, u >, < ns, u >, < n, \alpha_5 >, < n2, u >\}$	e_1^0	$\{< i, \alpha_0^* >, < j, \alpha_1^* >, < ne, \alpha_2^* >, < n1, \alpha_3^* >, < ns, \alpha_4^* >, < n, \alpha_5 >, < n2, \alpha_6^* >\}$	$true$	1

Table 2. Block-based symbolic execution of the example.

$\alpha_0 + 2\alpha_3(2^{I_1} - 1)$. The values of the other variables are deduced at the same manner and denoted by asterisks in the figure 3 and table 2. The resulting symbolic states are the terminal nodes s_0^t and s_1^t (figure 3).

It is possible to keep only the resulting states maximizing the WCET of the block. Merging of states is also performed which allows to reduce the number of states. Two states $s_1 =< V_1, PC_1, IP >$ and $s_2 =< V_2, PC_2, IP >$ with the same instruction pointer IP can be merged into one state $s =< V_1 \cup V_2, PC_1 \vee PC_2, IP >$. Hence, states s_0^t and s_1^t in the figure 3 may be merged into one state s^t in which $V = \{< i, \alpha_0 \vee \alpha_0^* >, < j, \alpha_1 \vee \alpha_1^* >, < ne, \alpha_2 \vee \alpha_2^* >, < n1, \alpha_3 \vee \alpha_3^* >, < ns, \alpha_4 \vee \alpha_4^* >, < n, \alpha_5 >, < n2, \alpha_6 \vee \alpha_6^* >\}$ and $PC = true$. State merging may cause some information loss and therefore incurs some pessimism in the WCET estimate. Therefore, a trade-off between the WCET precision and the number of generated states must be done.

These states constitute the block exiting symbolic states. Table 2 summarizes the results of the block-based symbolic execution of the example of figures 1 and 2 (u means undefined value).

When the execution reaches the block b_0^0 (level 0), the variables i and $n1$ are initialized respectively to 0 and 2. The number of iterations of b_0^1 is updated which evaluates to $I_1 = \lfloor \log_2(\frac{2}{3} + \frac{n}{6}) \rfloor + 1$.

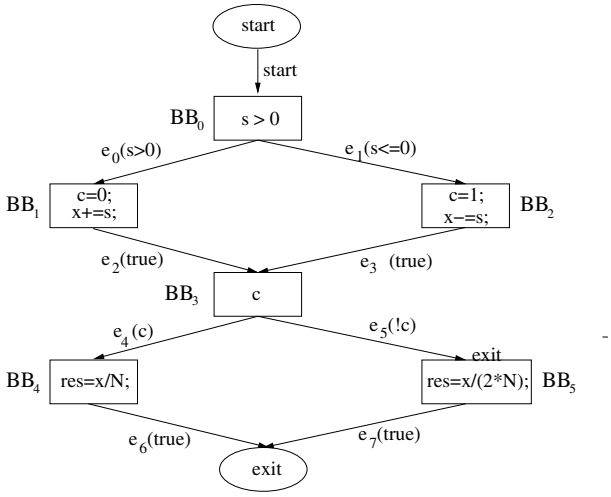
Furthermore, the flow analysis approach allows to express WCET as symbolic expressions function of the program parts input parameters (loop limits, function parameters, etc.). Thus, I_1 is expressed in terms of the fft function input parameters (n in this case). This is likely to improve the tightness of the WCET values and to reduce the complexity of the WCET analysis approach.

4.4. Iterative blocks with variant number of iterations

In the case of nested loops, the number of iterations of an inner loop may depend on the control variables of outer loops and thus varies following those dependencies. The worst-case number of iterations for such a block may be considered always its limit. This may result in an important WCET over-estimation. Therefore, the number of iterations

```
if (s > 0) {
  c = 0;
  x += s;
}
else {
  c = 1;
  x -= s;
}
if (c)
  res = x/N;
else
  res = x/(2*N);
```

a) C source code



b) Control flow graph

Figure 4. Example of false paths.

Block	Path	Type	Path composition	Path condition	Path action
b_0^1	p_0^2	$exit$	(BB_0, BB_1)	$s > 0$	$(c \leftarrow 0; x \leftarrow x + s)$
	p_1^2	$exit$	(BB_0, BB_2)	$s \leq 0$	$(c \leftarrow 1; x \leftarrow x - s)$
b_1^1	p_2^2	$exit$	(BB_3, BB_4)	c	$res \leftarrow x/N$
	p_3^2	$exit$	(BB_3, BB_5)	$\neg c$	$res \leftarrow x/(2 * N)$
b_0^0	p_0^1	$exit$	(b_0^1, b_1^1)	$true$	$(PA(b_0^1); PA(b_1^1))$

Table 3. Path definition and parameters of the false paths example.

Block	V_{in}	V_{out}	PC	Nb. Iter
b_0^1	$\{< s, \alpha_0 >, < c, \alpha_1 >, < x, \alpha_2 >\}$	$\{< s, \alpha_0 >, < c, 0 >, < x, \alpha_2 + \alpha_0 >\}$	$\alpha_0 \geq 1$	1
		$\{< s, \alpha_0 >, < c, 1 >, < x, \alpha_2 - \alpha_0 >\}$	$\alpha_0 \leq 0$	1
b_1^1	$\{< c, \alpha_0 >, < res, \alpha_1 >, < x, \alpha_2 >, < N, \alpha_3 >\}$	$\{< c, \alpha_0 >, < res, \alpha_2/\alpha_3 >, < x, \alpha_2 >, < N, \alpha_3 >\}$	$\alpha_0 \neq 0$	1
		$\{< c, \alpha_0 >, < res, \alpha_2/(2 \times \alpha_3) >, < x, \alpha_2 >, < N, \alpha_3 >\}$	$\alpha_0 = 0$	1
b_0^0	$\{< s, \alpha_0 >, < c, \alpha_1 >, < x, \alpha_2 >, < res, \alpha_3 >, < N, \alpha_4 >\}$	$\{< s, \alpha_0 >, < c, 0 >, < x, \alpha_2 + \alpha_0 >, < res, \frac{\alpha_2 + \alpha_0}{2 \times \alpha_4} >, < N, \alpha_4 >\}$	$\alpha_0 \geq 1$	1
		$\{< s, \alpha_0 >, < c, 1 >, < x, \alpha_2 - \alpha_0 >, < res, \frac{\alpha_2 - \alpha_0}{\alpha_4} >, < N, \alpha_4 >\}$	$\alpha_0 \leq 0$	1

Table 4. Block-based symbolic execution of the false paths program.

of an inner loop must be expressed in terms of control variables of outer loops values. The block-based symbolic execution approach is able to estimate a worst-case number of iterations of such blocks without overestimation.

A naive manner to compute the number of iterations of the block b_0^2 is to multiply the WCET of b_0^2 by the WCET of b_0^1, which gives $\alpha_2^* \times I_1$, for $n = 64$, this gives $8 * 4 = 32$. Our approach provides the value $2^{I_1} - 1 = 15$.

4.5. False paths

WCET analysis consists of finding the longest structural path in the program. However, some structural paths may be non-executed due to mutual exclusion control flow branches. Those paths are referred as infeasible paths. The block based symbolic execution eliminates implicitly most of the program infeasible paths. Figure 4a presents the C source code of a false path example where the CFG is shown in figure 4b. The set of block graphs of the program is composed of the blocks b_0^1 and b_1^1 corresponding to the two $if-else$ blocks respectively, and the block b_0^0 enclosing b_0^1 and b_1^1. Table 3 summarizes the path parameters and table 4 the block-based symbolic execution of the example. As we can see, the two infeasible paths going through the blocks BB_1, BB_4 and BB_2, BB_5 are implicitly excluded from the evaluation by the flow analysis approach.

5. Conclusion

Real-time systems must be predictable in time and memory because errors may lead to unpleasant consequences. Determining the execution time of programs is in the general case undecidable. Fortunately, given certain restrictions are met, bounds of the program execution time (BCET, WCET, etc.) may be estimated. static WCET analysis avoids dealing with the program input data and execution platforms, but results in overestimated values. Therefore, techniques allowing to tighten the WCET estimates are needed. However, these techniques are complex to automate because they deal with program semantics.

We proposed a practical approach aimed to automatically extract flow information related to program semantics. The method has two main advantages. It allows to tighten the WCET estimate values through the provided flow information related to program functionality. The approach handles iterative blocks with variant number of iterations (non-rectangular loops) and eliminates implicitly most of the infeasible paths. Moreover, the WCET estimate values are given as symbolic expressions function of the program input variables (function parameters, etc.). The second advantage concerns the enhanced complexity of the approach in terms of time and memory, through the block-based symbolic execution which avoids unfolding iterative blocks. Furthermore, the approach may be used to enhance the complexity of the symbolic execution.

We are implementing a prototype which integrates the flow information method with a WCET estimate calculation module in order to evaluate the performance of the approach. We plan also to extend the expressions used to evaluate loops to Presburger formulas and exploit the results obtained on those formulas.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, March 1986.

[2] 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.

[3] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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