Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis

D. Kebbal and P. Sainrat
Institut de Recherche en Informatique de Toulouse
118 route de Narbonne - F-31062 Toulouse Cedex 9 France

Abstract

This paper examines the problem of determining bounds on execution time of real-time programs. Execution time estimation is generally useful in real-time software verification phase, but may be used in other phases of the design and execution of real-time programs (scheduling, automatic parallelizing, etc.). This paper is devoted to the worst-case execution time (WCET) analysis. We present a static WCET analysis approach aimed to automatically extract flow information used in WCET estimate computing. The approach combines symbolic execution and path enumeration. The main idea is to avoid unfolding loops performed by symbolic execution-based approaches while providing tight and safe WCET estimate.

1. Introduction

Real-time systems are systems in which the execution time is subject to some constraints, which may lead to undesirable consequences when they are not respected, especially in hard real-time systems. The constraint validation process requires the knowledge of the execution time or bounds on the execution time of programs. WCET analysis is a popular approach used in the temporal constraints validation of hard real-time systems.

Static WCET analysis performs a high-level static analysis of the program source or object code. This avoids working on the program input data. Static WCET analysis consists of determining an upper bound on the program execution time. For each component of the program (block, task, etc.), an upper bound on the time of its execution is estimated. This definition implies that WCET analysis is only able to provide upper bounds on WCET values rather than exact values. Therefore, WCET analysis must guarantee two main properties in order to keep real-time systems predictable and their cost financially reasonable: *Safeness*; and *Tightness of provided WCET values*.

Static WCET analysis proceeds generally in three phases [4, 7]: flow analysis, low-level analysis and WCET estimate computing. Flow analysis characterizes the execution sequences of the program's components and their execution frequency (execution paths). Generally, two types of flow information are 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, 2]. Low-level analysis evaluates the execution time of each program component on the target hardware architecture. The calculation phase uses the results of the two previous steps to compute a WCET estimate for the program.

The remainder of this paper is organized as follows: the next section presents the related work. In section 3, we introduce some concepts which will be used by the flow analysis approach. Section 4 describes and discusses the proposed block-based symbolic execution method. Finally, 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. Path-based approaches proceed by explicitly enumerating the set of the program execution paths [10, 1]. [9] 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 program paths which scales exponentially with the program size. Another category of approaches called IPET[1] 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 con-

[1] Implicit Path Enumeration Techniques.

ECRTS 2006
6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2006/675

verted to the one of solving an ILP[2] problem [7, 11, 4].

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. [5] uses an interval-based abstract interpretation method that associates ranges to the program variables and allows to automatically extract flow information related to program semantics. The method proceeds by rolling out the program (especially loops) until it terminates, which is very costly in time and memory. [8] presents an approach for automatic parametric WCET analysis. The method uses abstract interpretation, a symbolic method to count integer points in polyhedra and a symbolic ILP technique. The approach seems complex in practice. 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. Moreover, only the flow information related to loop bounds is determined. 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 [3]. Symbolic execution-based methods are capable to work with small programs, but are not well suited for long and complex programs.

Our aim is to determine an approach which automatically extracts flow information related to program functionality and computes a safe and tight upper bound on the program WCET with a lower cost. We use a hybrid method based on symbolic execution and path enumeration. Loops are not unfolded, rather a path analysis is performed on each loop block.

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. The entry point is situated at the beginning of the block and the exit point at its end. 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 represents the program basic blocks and E the precedence constraints between them.

[2] Integer Linear Programming.

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 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 ($E_b^e \subseteq E_b$).

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

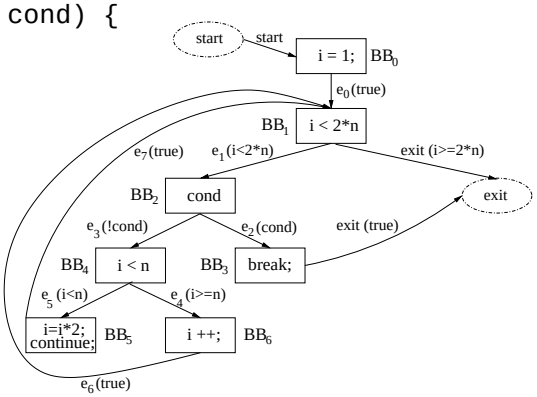
Each block b of level l is defined by a *block graph* describing its structure. The b's blocks set B_b is composed of blocks of higher levels ($m > l$). The set of edges E_b is constructed as follows: each edge of the CFG connecting two nodes belonging to two different blocks b_i and b_j of B_b forms an edge of level l from b_i to b_j. Edges to blocks outside B_b produce edges of *exit* type. Redundant edges are eliminated. In figure 2, in the graph of block b_0^1 corresponding to the *while* loop, the edge e_4 connecting the basic blocks BB_4 and BB_6 in the CFG yields the edge e_4^2.

A header block is a basic block executed when the execution flow reaches the block for the first time. Informally, header blocks correspond to loop and selection condition test blocks. The set of header-blocks of the block b_0^1 is $B_b^h = \{BB_1\}$ (figure 2). We handle only well-structured code programs yielding blocks with one header block. When the execution of the block is terminated, the control flow leaves the block through an exit-edge. The set of exit-edges of the block b_0^1 is $\{e_1^2, exit\}$. When the execution of a block 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 the block b_0^1 is $\{e_3^2, e_5^2\}$ (figure 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. In the following we refer by *block-paths* to the *one-iteration paths* in a block b. We distinguish

```
void some_func(int n, int cond) {
  int i = 1;
  while (i < 2*n) {
    if(cond)
      break;
    ... //statements
    if(i < n) {
      i = 2*i;
      continue;
    }
    i++;
  }
}
```



a) C source code. b) Control flow graph.

Figure 1. Example program.

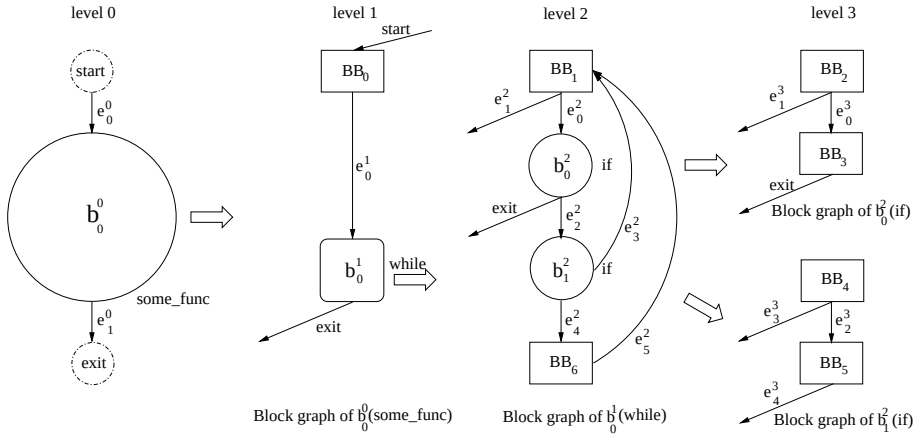


Figure 2. The block graphs of the example.

two types of block-paths: *exit_paths* and *loop_paths*. An exit-path in b is a path starting at the header block of b and ending by an exit-edge of b. Likewise, a loop-path in b is a 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 two exit paths: p_0^2 and p_1^2, and two loop-paths: p_2^2 and p_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, which is defined as one execution of a block-path.

4. Flow analysis approach

In this section, we describe our method aimed to automatically extract the flow information related to program semantics. 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 method. Only a subset of the symbolic states set of the program are computed (states at the entry point and the exit points of the block). Exit points of a block are the starting points of its exit edges.

A symbolic execution state may be represented by a triple $< V, PC, IP >$, where: $V = \{< v_1, e_1 >, \ldots, < v_n, e_n >\}$ is the set of pairs $< variable, expression >$, where the variables $v_1, \ldots, v_n$ have been assigned the expressions $e_1, \ldots, e_n$; PC is the path condition expressing the conditions under which that path is taken; and IP refers to the next instruction to execute. Initially, the input parameters are initialized using symbols and the other variables to the special value $undef$. Symbolic execution of a program takes a symbolic state and a rule which corresponds to the

current statement referred by IP and returns the symbolic states resulting from the execution of the statement.

In order to ensure the analyzability of real-time software, the flow analysis method imposes some limitations on the handled programs. First, potentially non-deterministic and complex programming language features like recursion, dynamic memory allocation and unstructured code are not allowed. Moreover, for instance the loop induction variable update statement is limited to the form $i = a * i + b$. We think that this restriction is compatible with hard real-time systems, knowing that many works use more restrictive formulas [6]. Furthermore, the expressions can be easily extended to Presburger formulas.

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 block-path p denoted $PA(p)$ is the sequence of the block action of all blocks constituting that path. Likewise the path condition of a block-path is the *"logical and"* of the path condition of all edges forming that path. $PC(p) = PC(e_0) \wedge PC(e_1) \ldots \wedge PC(e_{n-1})$.

In order to compute the path action of a block-path p, we consider the set V_p of program variables assigned in different blocks of 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. $PA(p, v) = (BA(bb_0, v); \ldots; BA(bb_{n-1}, v))$ is represented by an expression of the form $av + b$ (for loop induction variables) such that a and b are integer constants ($a > 0$).

4.2. Algebraic evaluation of paths

Paths are evaluated by decomposing each conditional expression $PC(p)$ into elementary Boolean expressions related by logical operators. Each conditional expression e is of the form $i\ op\ expr$, where op is a relational operator and $expr$ is an integer valued expression. For each expression e, the following parameters are evaluated:

- *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 is increased in the path action of p ($PA(p)$), 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.

The *direction* and the *interval type* are used to check for empty end unbounded paths before evaluating the number of iterations of the path. This step allows to determine the path parameters (a, b, N_1 and N_2) used by the formula 2.

In figure 3, for the state ($V = \{< i, \alpha_0 >, < n, \alpha_1 > , < cond, \alpha_2 >\}, PC = \alpha_0 \leq \alpha_1 - 1 \wedge \alpha_2 = 0$), and the path p_2^2, there are two expressions $e_1 = \alpha_0 \leq \alpha_1 - 1$ and $e_2 = \alpha_2 = 0$. The set of variables involved in e_1 is $\{i\}$, $PA(p_2^2, i) = i \leftarrow 2i$. Therefore the interval type of e_1 is *"raised"* and the direction is *"positive"*. The number of iterations is never empty nor unbounded. Then the number of iterations I_p^e of the path p related to e and the resulting symbolic state S_p^e are calculated.

In order to analytically compute the number of iterations of a loop path p, we define the following suite v_n:

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

The number of iterations I is defined by the formula $N_2 - N_1 \geq \sum_{n=0}^{I-2} s_n$. $s_n = v_{n+1} - v_n$. 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 equation 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 unbounded ($I = \infty$). Let us consider the loop: $for(i = 1; i <= 100; i = 2 * i + 1)$, the parameters characterizing the loop path are: $a = 2, b = 1, N_1 = 1$ and $N_2 = 100$. The algebraic tool evaluates the number of iterations to 6.

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

4.3. Block-based symbolic execution

The block-based symbolic execution proceeds in a post-order manner. The blocks of the level $l + 1$ are evaluated before the blocks of the level l. The evaluation of a block b is performed in the following steps:

Block	Path	Type	Path composition	Path condition	Path action	Edge
b_0^2	p_0^3	$exit$	$(e_1^3) = (BB_2)$	$cond \neq 0$		e_2^2
	p_1^3	$exit$	$(e_0^3, exit) = (BB_2, BB_3)$	$cond = 0$		$exit$
b_1^2	p_2^3	$exit$	$(e_3^3) = (BB_4)$	$i \geq n$		e_4^2
	p_3^3	$exit$	$(e_2^3, e_4^3) = (BB_4, BB_5)$	$i \leq n-1$	$(i \leftarrow 2i)$	e_3^2
b_0^1	p_0^2	$exit$	$(e_1^2) = (BB_1)$	$i \geq 2n$		$exit$
	p_1^2	$exit$	$(e_0^2, exit) = (BB_1, b_0^2)$	$i \leq 2n-1 \wedge cond \neq 0$		$exit$
	p_2^2	$loop$	$(e_0^2, e_2^2, e_3^2) = (BB_1, b_0^2, b_1^2)$	$i \leq n-1 \wedge cond = 0$	$(i \leftarrow 2i)$	
	p_3^2	$loop$	$(e_0^2, e_2^2, e_4^2, e_5^2) = (BB_1, b_0^2, b_1^2, BB_6)$	$n \leq i \leq 2n-1 \wedge cond = 0$	$(i \leftarrow i+1)$	
b_0^0	p_0^1	$exit$	$(start, e_0^1, exit) = (BB_0, b_0^1)$	$true$	$(i \leftarrow 1; PA(b_0^1))$	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.

a- Path information The first step consists of determining a set of information characterizing the block. We determine the set of *block-paths* of the block. For each path a set of parameters is calculated: path type (*loop* 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. This information is kept in a table where a summary for the example of the figure 1 is shown in table 1.

b- Block evaluation The symbolic execution of a block b is performed by evaluating all the *block_paths* $P = block\_paths(b)$ starting at the entry point of the block with a symbolic state in which all variables used in the block are assigned symbols. For evaluating each path p we use a symbolic state in which PC corresponds to the path condition of p. This step yields the set of the block exit states S_b and the number of iterations of the block (figure 3).

c- Path evaluation The path evaluation takes a symbolic state s, the path action $PA(p)$ and performs the algebraic evaluation of the path using the formula 2. The result is the number of iterations of the path I_p and the generated symbolic states S_p.

Figure 3 illustrates the block-based symbolic execution of the block b_0^1. Edges are annotated by the number of iterations applied by the path on the symbolic state of the starting node. We assume that the variable $cond$ is updated in the *"instructions"* block but not n. The resulting symbolic states are the terminal nodes ($s_0^t, s_1^t, \ldots$) (figure 3). Merging of states may be performed which allows to reduce the number of resulting 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 $s =< V_1 \cup V_2, PC_1 \vee PC_2, IP >$. Merging of states may sometimes cause information loss. Then some pessimism will be incurred in the WCET estimate. Therefore, a trade-off must be done between the WCET precision and the number of generated states. States s_1^t and s_2^t may be merged into one state s_3^t ($< V = \{< i, \alpha_0^1 >, < n, \alpha_1 >< cond, \alpha_2^* >\}, PC = \alpha_0 \leq \alpha_1 - 1 \wedge \alpha_1 \leq \alpha_0^1 \leq 2\alpha_1 >$).

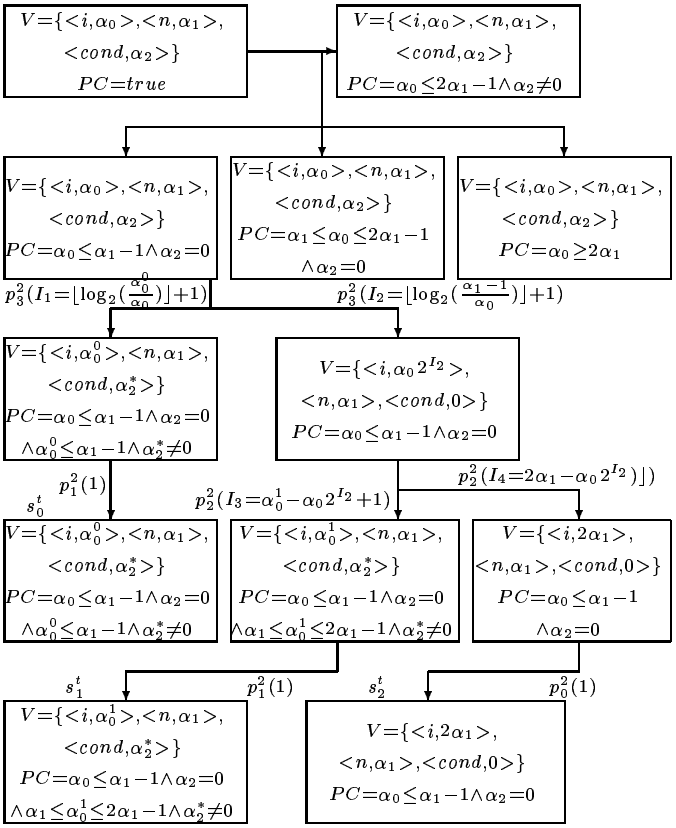


Figure 3. Block-based symbolic execution of the block b_0^1

These states constitute the block exiting symbolic states which would be used in the evaluation of lower level blocks (ex. b_0^0). Indeed, when the current block b^l is examined in the framework of a block b^r of lower level, the block action of the block $BA(b^l)$ is evaluated in one step.

When the execution reaches the block b_0^0 (higher level), the variable i is initialized to 1, then the incomplete branches of the figure 3 are discarded (PC becomes $false$), reducing thus the number of the block exiting states to only 3. Furthermore, it is possible to keep only the resulting states maximizing the WCET of the block.

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 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 over-estimation.

Assume that the *"statements"* area of the figure 1-a comprises a block b_0^3 consisting of the loop: $for(j = 0; j < i; j++)$. One can estimate the WCET of the loop nest to $2n \times (2n-1)$ since i starts with the value 1. When applying the block-based symbolic execution: $s_{in} =< \{< i, \alpha_0 >, < n, \alpha_1 >, < j, \alpha_3 >\}, PC = true >$, the results are $s_{out} =< \{< i, \alpha_0 >, < n, \alpha_1 >, < j, \alpha_1 >\}, PC = true >$ with a number of iterations of $\alpha_1 - \alpha_0$. When the analysis reaches the top level (i is initialized to 1), the number of iterations is evaluated to $\sum_{i=1}^{2^{I_2}} i[i \rightarrow 2i] + \sum_{i=2^{I_2}}^{2n-1} i$. For example, for $n = 10$, the intuitive method yields 380 while our method provides the actual WCET 85.

In addition, the block based symbolic execution eliminates implicitly most of the program infeasible paths and allows to express the WCET estimates as symbolic expressions function of the program parts input parameters (function parameters, etc.). The quality of the provided flow information is comparable to the one of symbolic execution and abstract interpretation schema since our approach is a symbolic execution method.

5. Conclusion

WCET analysis is a popular method used to validate the temporal correctness of real-time systems. WCET analysis may be done statically on the program source or object code, which 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 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 in terms of time and memory by avoiding unfolding iterative blocks. Moreover, the approach provides tight values since it handles non rectangular loops and loops with multiple exit conditions and eliminates implicitly most of the infeasible paths.

We are implementing a prototype of the method in order to evaluate its performance. 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] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

[3] 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.

[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.