SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.406

A systematic representation of path constraints for implicit path enumeration technique

Tai Hyo Kim[1], Ho Jung Bang[1] and Sung Deok Cha[2,*,†]

[1]*Formal Works Inc., II Song building 202, 663-26 Yuksam dong, Gangnam gu, Seoul, Republic of Korea*
[2]*College of Information & Communication, Division of Computer Science and Communication Engineering, Korea University, Anam-dong, Seongbuk-Gu, Seoul, Republic of Korea*

SUMMARY

Accuracy of implicit path enumeration technique (IPET), which statically estimates the worst-case execution time of a program using integer linear programming, relies on flow information captured as flow facts. Unfortunately, flow facts are inadequate for capturing complex and often subtle path constraints such as causalities. Manual annotation often introduces many disjunctions, and performance of IPET computation suffers significantly. This paper proposes a technique of encoding a subset of path constraints into flow facts. The technique has advantages over conventional approaches: (1) translation process is fully automated and (2) efficient IPET computation is possible because generated flow facts are compact in that they contain at most one disjunction. To demonstrate the effectiveness of our technique, a software tool was implemented to automatically generate flow facts for the subset of path constraints and case study has been conducted using public benchmark suites, GNU openSSH codes, and Korea multi-purpose satellite (KOMPSAT-1) software. Copyright © 2009 John Wiley & Sons, Ltd.

Received 18 December 2006; Revised 15 October 2008; Accepted 8 February 2009

KEY WORDS: real-time systems; worst-case execution time; WCET; static analysis; implicit path enumeration technique; infeasible paths

*Correspondence to: Sung Deok Cha, College of Information & Communication, Division of Computer Science and Communication Engineering, Korea University, Anam-dong, Seongbuk-Gu, Seoul, Republic of Korea.
†E-mail: scha@korea.ac.kr

Contract/grant sponsor: KOSEF
Contract/grant sponsor: Defense Acquisition Program Administration and Agency for Defense Development; contract/grant number: UD060048AD
Contract/grant sponsor: Ubiquitous Computing and Network (UCN) Project
Contract/grant sponsor: Ministry of Information and Communication (MIC) 21st Century Frontier R&D Program

1. INTRODUCTION

Accurate estimation of the worst-case execution time (WCET) is crucial when verifying the correctness of the real-time software and scheduling its tasks [1]. In fact, many safety-critical systems mandate reliability and safety demonstration, and WCET analysis is an important component in determining if timing requirements can be met. Traditional measurement-based techniques execute software with a set of test cases and choose the largest execution time as the WCET estimate. They, however, cannot guarantee the safeness of the estimate because testing of all the feasible paths is generally impossible.

Static methods approximate WCET by analyzing source codes without executing them. Examples include tree-based techniques [2–4], path-based techniques [5,6], and implicit path enumeration technique (IPET) [7–9]. Although static WCET methods are safe and automated, they usually overestimate the results because flow information derived statically is often incomplete and over-simplified [10]. If WCET estimates are overly pessimistic, valuable CPU resources will be wasted. Furthermore, to guarantee the fulfillment of real-time requirements, more expensive (or power consuming) systems than necessary might have to be deployed. To overcome such limitations, some approaches [11–14] relied on user annotation of additional path constraints, and techniques proposed in references [6,15–17] used data flow analysis to extract such constraints automatically.

Among the static methods, IPET uses *flow facts*, conjunctive linear constraints on the execution counts of basic blocks, to represent flow information. Integer linear programming (ILP) solvers then calculate the maximum execution time as satisfying all the flow facts. Thus, all additional path constraints should be presented in flow facts in order to be applied in IPET computation.

Encoding path constraints into flow facts involves two steps. First, path constraints are rewritten in a superset of flow facts, *flow predicates*, which may include disjunctions. Unfortunately, this step cannot be completely automated because complex path constraints are difficult to capture as constraints on the execution counts alone.

Next, disjunctions included in the flow predicates must be removed since generic ILP solvers accept only conjunctive constraints. Li and Malik [7] translated flow predicates into disjunctive sets of flow facts and computed all local maximums for the sets to determine the longest execution time. This approach, though feasible, is apparently inefficient because predicates for path constraints containing many disjunctions increase the number of sets exponentially.

This paper focuses on two types of path constraints, *positive* and *negative dependencies*, among basic blocks. The former indicates that the execution of a sequence of basic blocks triggers that of the other basic blocks. The latter refers to a sequence of basic blocks that cannot be simultaneously included in valid execution paths.

This paper reports that a subset of such dependencies can be automatically and effectively translated into flow predicates. The authors have developed an algorithm to statically determine whether a path constraint belongs to such a subset. Furthermore, the resulting predicates are guaranteed to contain no disjunctions when specifying positive dependencies and at most one disjunction for each negative dependency. Thus, the number of IPET calculations is significantly reduced because only the negative dependencies containing a disjunction need to be partitioned.

As an application of the proposed technique, a software tool was implemented to automatically generate functional flow facts from an input code. The tool uses abstract interpretation technique [16,18] to identify infeasible paths, performs path slicing technique [19] to translate the paths into dependencies, and generates additional flow facts for them. Experiments conducted on benchmark suites, OpenSSH, and satellite control software revealed that about 30% of the infeasible paths could be translated into flow facts fully automatically.

This paper is organized as follows. Sections 2 and 3 briefly describe research backgrounds and explain IPET. Section 4, describing the main contributions of this paper, explains how to encode a subset of path constraints into flow facts and Section 5 demonstrates how to automate the process. The paper presents experimental results in Section 6 and concludes in Section 7.

2. PRELIMINARIES

This paper uses a C language-like imperative language for the presentation. Figure 1 shows an example code and its control flow graph (CFG). A program P consists of basic blocks (*Blks*), maximal sequences of statements where all the statements are executed if the first statement is executed, and control flows among them ($E : Blks \times Blks$). In the figure, basic blocks $\mathtt{B}_1$ through $\mathtt{B}_8$ are labeled in CFG nodes. Each block has a sequence of statements (*Stmts*) and $\mathtt{stmt\_of(B,i)}$: $Blks \times \mathbb{N} \to Stmts$ refers to the $\mathtt{i}$th statement of basic block $\mathtt{B}$[‡].

2.1. Linear (time) structures for a program

An execution path (*or a run*) *of a program* P is a sequence of basic blocks that starts with an entry block (b_{entry}) and passes blocks along the control flows E. A finite path finishes at an exit block (B_{exit}) and an infinite path repeats loops infinitely. To avoid verbose formal notations, the remaining sections assume a program $P = (Stmts,\ Blks,\ E,\ b_{entry},\ B_{exit})$.

This paper treats all the execution paths as infinite ones uniformly; finite runs are transformed to semantically equivalent infinite ones by concatenating an infinite sequence of dummy blocks $\mathtt{B}_\varepsilon\mathtt{B}_\varepsilon\ldots$ at the ends of runs. For example, a finite run $\pi = \mathtt{B}_1\mathtt{B}_2\mathtt{B}_6\mathtt{B}_7\mathtt{B}_8$ is transformed to an infinite run $\pi^\omega = \mathtt{B}_1\mathtt{B}_2\mathtt{B}_6\mathtt{B}_7\mathtt{B}_8\mathtt{B}_\varepsilon\mathtt{B}_\varepsilon\ldots$. All execution paths of a program can be regarded as infinite paths, and they are termed as the language of a program P, $\mathscr{L}(P)$.

With this setup, the semantics of a program can be defined as linear time structures [20] as follows. An infinite execution path $\pi^\omega = b_0 b_1 \ldots$ of a program P can be represented as a linear time structure $M_{\pi^\omega} = (Q, \pi^\omega, L)$, where the states Q are basic blocks *Blks*. To specify the path constraints among basic blocks, this paper uses atomic propositions ap_{b_i}, which are true only at the corresponding block b_i as labels L. That is, atomic propositions AP_{Blks} are $\bigcup_{\mathtt{B_k} \in Blks} \{ap_{\mathtt{B_k}}\}$ and labeling function $L(\mathtt{B_i})$ returns $\{ap_{\mathtt{B_i}}\}$, e.g. $ap_{\mathtt{B}_2} \in L(\mathtt{B}_2)$ and $ap_{\mathtt{B}_3} \notin L(\mathtt{B}_2)$. For simplicity of presentation, we use b_i and ap_{b_i} interchangeably if apparent.

2.2. Path constraints and linear temporal logic (LTL)

Temporal logic provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time [20]. Temporal logic has been widely used in specifying and reasoning properties of reactive systems.

[‡]The set $\mathbb{N}$ is non-negative integers including zero. As a subset, $\mathbb{N}_{\le k}$ denotes non-negative integers less than or equal to k.

(a)

```
int foo(int a) {
  int i = 3;          // B1
  int b = 1;
  while(i>0) {        // B2
    i--;              // B3
    if(a>=0) {
      b=b*a;          // B4
    } else {
      b=b+a;          // B5
    }
  }
  if(b<0) {           // B6
    b= -1 * b;        // B7
  }
  return b;           // B8
}
```

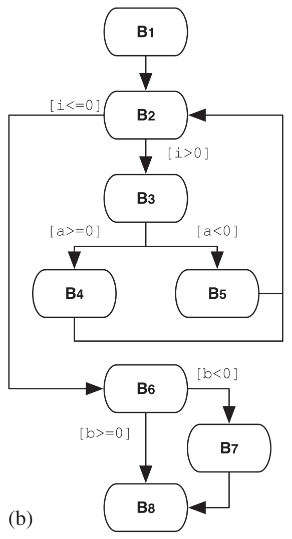

Figure 1. An example program: (a) an example code and (b) CFG of (a).

(a)
$$LTL ::= p \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\, \varphi$$
$$\mid \mathrm{F}\varphi \mid \mathrm{G}\varphi \mid \mathrm{X}\varphi \mid \psi\ \mathrm{U}\varphi$$

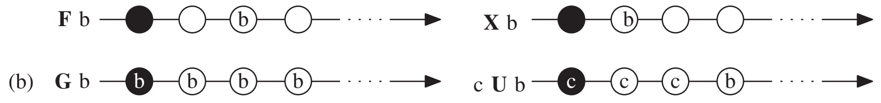

Figure 2. Linear temporal logic: (a) syntax of linear temporal logic. Symbol p is an atomic proposition and φ, ψ are LTL formulas and (b) temporal operators. Black dots are entry blocks.

This paper uses LTL to formally define *path constraints*, requisites for valid execution paths of P. In Figure 1(a), block $\mathtt{B}_7$ cannot be executed if $\mathtt{B}_4$ was executed earlier because variables a and b remain non-negative since then. This paper formally defines such constraints in terms of LTL formulas accepting only valid execution paths over linear structures for a program.

An LTL formula for a path constraint is built up with atomic propositions AP_{Blks}, boolean connectives ($\wedge$, $\vee$, $\neg$), and temporal operators $\mathbf{F}p$ (sometimes p), $\mathbf{G}p$ (always p), $\mathbf{X}p$ (next time p), and $p\mathbf{U}q$ (p until q) as presented in Figure 2(a). By mixing the boolean and temporal operators in nested sub-formulas, LTL provides enough expressive power to capture the various path constraints precisely. Figure 2(b) illustrates the meaning of temporal operators briefly: temporal operator $\mathbf{F}b$ accepts execution paths where block b occurs at a future moment and $\mathbf{G}b$ accepts paths only repeating b forever. An execution path whose second block is b satisfies $\mathbf{X}b$ and a path repeating block c until b is executed satisfies $c\mathbf{U}b$.

The semantics of an LTL formula φ can be formally defined as linear structures satisfying it. The satisfaction relation $\pi \vDash \varphi$ holds if a linear structure $M=(Q,\pi^{\omega},L)$ (or an execution path π^{ω}) satisfies φ. Definition 1 inductively formalizes $\vDash$ relation. In the definition, π^{ω}_k is the kth state, i.e. $\pi^{\omega}_k=\pi^{\omega}(k)$ and $post_k(\pi^{\omega})$ is the postfix of π^{ω} starting from the kth state, i.e. $\pi^{\omega}_k\pi^{\omega}_{k+1}\ldots$.

Definition 1 (*Linear temporal logic*). let p be an atomic proposition, φ, ψ be LTL formulas, and $M=(Q,\pi^{\omega},L)$ be a linear time structure.

- $\pi^{\omega}\vDash p$ *iff* $p\in L(\pi^{\omega}_0)$
- $\pi^{\omega}\vDash \varphi\vee\psi$ *iff* $\pi^{\omega}\vDash\varphi$ or $\pi^{\omega}\vDash\psi$
- $\pi^{\omega}\vDash \varphi\wedge\psi$ *iff* $\pi^{\omega}\vDash\varphi$ and $\pi^{\omega}\vDash\psi$
- $\pi^{\omega}\vDash \neg\varphi$ *iff* $\pi^{\omega}\nvDash\varphi$
- $\pi^{\omega}\vDash \mathbf{F}\varphi$ *iff* $\pi^{\omega}\vDash\varphi$ or $post_1(\pi^{\omega})\vDash\mathbf{F}\varphi$
- $\pi^{\omega}\vDash \mathbf{G}\varphi$ *iff* $\pi^{\omega}\vDash\varphi$ and $post_1(\pi^{\omega})\vDash\mathbf{G}\varphi$
- $\pi^{\omega}\vDash \mathbf{X}\varphi$ *iff* $post_1(\pi^{\omega})\vDash\varphi$
- $\pi^{\omega}\vDash \psi\mathbf{U}\varphi$ *iff* $\exists j: post_i(\pi^{\omega})\vDash\varphi\wedge\{\forall k<j: post_k(\pi^{\omega})\vDash\psi\}$

In Figure 1, $\mathtt{B}_1\mathtt{B}_2\ldots\mathtt{B}_6\mathtt{B}_7\mathtt{B}_8\vDash\mathbf{F}\mathtt{B}_8$ holds because $\mathtt{B}_8$ is eventually executed in the all path[§]. Likewise, a formula $\varphi=\mathbf{G}(\mathtt{B}_4\rightarrow\mathbf{G}\neg\mathtt{B}_7)$ denotes that $\mathtt{B}_7$ is not executed if $\mathtt{B}_4$ is executed earlier and, therefore, $\mathtt{B}_1\mathtt{B}_2\mathtt{B}_3\mathtt{B}_4\mathtt{B}_2\mathtt{B}_6\mathtt{B}_7\mathtt{B}_8\nvDash\varphi$.

3. IMPLICIT PATH ENUMERATION TECHNIQUE

IPET-based WCET estimation consists of three major phases: *program flow analysis*, *low-level analysis*, and *calculation*. The program flow analysis extracts flow information on possible execution paths and encodes them into flow facts. The low-level analysis determines the number of CPU cycles that each basic block takes to execute. Both the static technique and measurement are applicable for this phase [21,22]. In this phase, hardware-dependent characteristics (e.g. pipeline [23], cache [24,25], and branch prediction [26–28]) must be taken into consideration to obtain accurate results. For example, the execution time of a block would be less if referenced variables remain in the cache, and incorrect branch prediction would delay execution. Finally, an ILP solver computes a WCET estimate using flow facts and timing information.

In IPET, *structural flow facts* denote control flow information. Let χ_i and f^k_j denote the number of times a basic block $\mathtt{B_i}$ is executed and a control flow from $\mathtt{B_j}$ to $\mathtt{B_k}$ is taken, respectively. In Figure 1, block $\mathtt{B}_2$ has three incoming edges (from $\mathtt{B}_1$, $\mathtt{B}_4$, and $\mathtt{B}_5$) and two outgoing edges (to $\mathtt{B}_3$ and $\mathtt{B}_6$). Thus, the execution count of $\mathtt{B}_2$ (χ_2) must be equal to the sum of the execution counts of all the incoming edges ($f^2_1+f^2_4+f^2_5$) as well as that of the outgoing edges ($f^3_2+f^6_2$) as shown in Equation (2) of Figure 3(a).

Structural flow facts are, unfortunately, insufficient to compute a WCET estimate. They contain no information on loop bounds. In the example code, $\mathtt{B}_1$ is executed once in the function and the loop is executed three times. Flow facts (9)–(11) represent such flow information. Note that

[§]$\mathbf{F}\mathtt{B}_8$ abbreviates $\mathbf{F}ap_{\mathtt{B}_8}$ as mentioned.

$$\begin{array}{ll}
\chi_1 = f_1^2 & (1)\\
\chi_2 = f_4^2 + f_5^2 + f_1^2 = f_2^3 + f_2^6 & (2)\\
\chi_3 = f_2^3 = f_3^4 + f_3^5 & (3)\\
\chi_4 = f_3^4 = f_4^2 & (4)\\
\chi_5 = f_3^5 = f_5^2 & (5)\\
\chi_6 = f_2^6 = f_6^7 + f_6^8 & (6)\\
\chi_7 = f_6^7 = f_7^8 & (7)\\
\chi_8 = f_7^8 + f_6^8 & (8)
\end{array}$$

(a)

$$\begin{array}{rl}
\chi_1 \leq 1 & (9)\\
\chi_2 \leq 4 & (10)\\
\chi_4 + \chi_5 \leq 3 & (11)
\end{array}$$

(b)

Figure 3. Flow facts for finiteness: (a) structural flow facts and (b) flow facts for finiteness.

Table I. The result of ILP solver.

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	WCET
$\tau(B_i)$	7	3	3	7	3	3	3	10	
$(\mathscr{S}, \mathscr{W})$	1	4	3	3	0	1	**1**	1	**65**
$(\mathscr{S}', \mathscr{W}')$	1	4	3	3	0	1	**0**	1	**62**

flow fact (10) set the boundary of B_2's execution count to 4 to include the exit transition to B_6. This paper assumes that such *finiteness flow facts* have already been derived using the existing techniques [13,29,30].

Given the above assumptions, the proposed technique is applied to a bounded program $\overline{P} = (\mathit{Stmts}, \mathit{Blks}, E, b_{entry}, B_{exit}, \mathscr{V}: \mathit{Blks} \rightarrow \mathbb{N})$ whose execution counts of nodes b are bounded to finite numbers $\mathscr{V}(b)$. Execution paths of a bounded program can be represented as flow facts on structural information and finiteness.

A bounded program $\overline{P}$ defines only finite execution paths from the initial node (b_{entry}) to a final node in B_{exit} because all execution counts of basic blocks are bounded. In a run $\pi = b_0 b_1 b_2 \ldots b_n$ of $\overline{P}$, a block b may occur at most $\mathscr{V}(b)$ times. That is, $|\pi|_b \leq \mathscr{V}(b)$ where $|\pi|_{b \in Blks}$ denotes the occurrence number of the node b in the sequence π. Note that the semantics of a bounded program $\overline{P}$ can also be given in linear structures using B_ε analogous to a program P.

If transition times between the basic blocks are ignored, the total execution time of a path is the sum of time necessary for executing each basic block b, $\tau(b)$, as many times as its occurrence in the sequence, χ_b. The objective function $\mathscr{T}(\overline{P})$, therefore, becomes

$$\mathscr{T}(\overline{P}) = \sum_{b \in Blks} \tau(b) \times \chi_b$$

WCET is the maximum value of this objective function satisfying all the given flow facts, namely, $\mathit{WCET}(\overline{P}) = \mathit{max}(\ \mathscr{T}(\overline{P})\ )$. ILP solves this maximization problem and returns a WCET estimate $\mathscr{W}$ and a solution $\mathscr{S}$ indicating how many times each block is executed. In Table I, the result

$$(\chi_4 = 0 \vee \chi_5 = 0) \qquad (12)$$
$$(\chi_4 \geq 1 \rightarrow \chi_7 = 0) \equiv (\chi_4 = 0 \vee \chi_7 = 0) \qquad (13)$$
$$(\chi_5 \geq 1 \rightarrow \chi_7 \geq 1) \equiv (\chi_5 = 0 \vee \chi_7 \geq 1) \qquad (14)$$

Figure 4. Functional flow information.

$(\mathscr{S}, \mathscr{W})$ illustrates the WCET estimate, 65 cycles, derived using only structural and finiteness flow facts (1)–(11), while assuming that execution of $\mathtt{B_i}$ takes $\tau(\mathtt{B_i})$ CPU cycles.

Unfortunately, the estimate $\mathscr{W}$ shown in Table I is not tight enough since the chosen WCET path, where $\mathtt{B_4}$ and $\mathtt{B_7}$ are supposed to be executed thrice and once, respectively, is infeasible: block $\mathtt{B_7}$ is executed only when variable `b` has a negative value, but `b` remains non-negative if $\mathtt{B_4}$ is executed. Such an infeasible path problem [31] occurred because static analysis failed to fully capture semantic dependencies. Had path constraint in flow predicate (13) in Figure 4 showing that '$\mathtt{B_7}$ is never executed if $\mathtt{B_4}$ were executed at least once' been included¶, ILP solver would not have picked such a path as the WCET solution.

Furthermore, as the value of input parameter `a` is never updated, the truth of loop condition at line 6, ($\mathtt{a} \geq \mathtt{0}$), never changes. Thus, valid execution paths must never include $\mathtt{B_4}$ and $\mathtt{B_5}$ simultaneously as flow predicate (12) in Figure 4 indicates. More subtle dependencies involve $\mathtt{B_5}$ and $\mathtt{B_7}$. Flow predicate (14) in Figure 4 denotes that if block $\mathtt{B_5}$ had been executed at least once, $\mathtt{B_7}$ must also be included in valid execution paths.

Inclusion of the additional functional flow information reduces the WCET estimate from 65 to 62 cycles as $(\mathscr{S}', \mathscr{W}')$ in Table I. The more the functional flow information available, the more the infeasible paths can be eliminated and the tighter the WCET estimate becomes.

Extracting functional flow information and capturing them as equivalent flow predicates are neither trivial nor always possible. Data flow analysis using abstract interpretation [18] with respect to signs of values may help elicit the above functional flow facts. However, not all of such information can be derived automatically, and manual annotation is often needed as a supplement.

Not all of the functional flow information elicited can be translated into flow predicates, either. Constraints on execution counts are not expressive enough to capture ordering relation among basic blocks in dependency. For example, predicate $(\chi_3 \geq 1 \rightarrow \chi_4 \geq 1)$ does not accurately capture the constraint that the execution of $\mathtt{B_3}$ must always be followed by that of $\mathtt{B_4}$ because an execution path in which $\mathtt{B_4}$ occurs prior to $\mathtt{B_3}$ would also satisfy this predicate.

In addition, disjunctions contained in flow predicates decrease the performance of IPET significantly. Because generic ILP solvers accept only conjunctive linear equations, the separation technique [7] splits flow predicates into disjunctive sets of flow facts. These sets of flow facts are calculated separately to identify local maximums, and the longest execution time among them is selected as the WCET. For example, flow facts (12)–(14) can be divided into eight sets of flow facts including a flow facts set $\{(1), (2), \ldots, (12), \chi_4 = 0, \chi_5 = 0\}$ where the three left disjuncts of the equations is contained.

¶Predicate $(\chi_4 \geq 1 \rightarrow \chi_7 = 0)$ can be rewritten as an equivalent equation $(\chi_4 = 0 \vee \chi_7 = 0)$ including a disjunction since all execution counts are non-negative integers.

It is apparently inefficient to apply the separation technique to predicates containing many disjunctions because the number of combinations increases exponentially. Knowledge about the program behavior may simplify flow predicates. For example, a predicate ($\chi_4=3 \wedge \chi_7=0$), requiring only one ILP calculation, effectively replaces predicates (12)–(14), and the same WCET estimate, 62 cycles, would be returned.

4. ENCODING OF PATH CONSTRAINTS FOR IPET

4.1. Path constraints

This paper focuses on two representative types of path constraints: *positive* and *negative dependencies* among basic blocks. These dependencies are well-known and popular patterns of path constraints [32]. Moreover, they are theoretically primitive in that all the other constraints can be represented as a set of infeasible paths, which are always transformable into negative dependencies.

Functional flow information (13) shown in Figure 4 is an example of a negative dependency since inclusion of $\mathtt{B}_4$ in any valid execution path eliminates $\mathtt{B}_7$. The positive dependency, shown in (14) in Figure 4, means that execution of $\mathtt{B}_5$ should be followed by that of $\mathtt{B}_7$ later. This paper denotes these constraints as $\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$ and $\mathtt{B}_5 \rightsquigarrow \mathtt{B}_7$, respectively. More rigorously, both types of constraints consist of triggering blocks and a consequence block, represented as '$t_1 \circ t_2 \circ \cdots \circ t_n \rightsquigarrow c$' or '$t_1 \circ t_2 \circ \cdots \circ t_n \not\rightsquigarrow c$'. Concatenation operators '$\circ$' are omitted when apparent.

The semantics of a dependency in $\overline{P}$ can be expressed in an LTL formula accepting only runs satisfying the dependency. Let $\mathscr{L}(\overline{P})$ be the language of a program. Given an LTL formula ψ, *formula preserving paths* (*or formula paths*) $\mathscr{L}_\psi(\overline{P})$ is a set of execution paths that satisfies the formula ψ, namely $\{\pi^\omega \in \mathscr{L} \mid \pi^\omega \vDash \psi\}$. The formal semantics of positive and negative dependencies are defined as follows.

Definition 2 (*Positive dependency*). *A positive dependency* $t_1 t_2 \ldots t_i \rightsquigarrow c$ of $\overline{P}$ is a set of formula paths $\mathscr{L}_\varphi(\overline{P})$, where $\varphi = \mathbf{G}\ \{t_1 \wedge \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots(t_{i-1} \wedge \mathbf{F}t_i))) \rightarrow \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots(t_{i-1} \wedge \mathbf{F}(t_i \wedge \mathbf{F}c))))\}$[‖].

Definition 3 (*Negative dependency*). *A negative dependency* $t_1 t_2 \ldots t_i \not\rightsquigarrow c$ is a set of formula paths $\mathscr{L}_\varphi(\overline{P})$, where $\varphi = \mathbf{G}\ \{t_1 \wedge \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots(t_{i-1} \wedge \mathbf{F}t_i))) \rightarrow \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots(t_{i-1} \wedge \mathbf{F}(t_i \wedge \mathbf{G}(\neg c)))))\}$.

The only difference between the above definitions is the terms $\mathbf{F}c$ and $\mathbf{G}\neg c$ included in the formulas. The former requires that node c must occur later in the sequence, while the latter denotes that node c must not. The remaining terms specify the order among triggers.

4.2. Translation of simple path constraints

A dependency can be rewritten in a flow predicate if nodes included in the constraint are always executed in a certain order. For example, formula path $\mathbf{G}(\chi_1 \geq 1 \rightarrow \mathbf{F}\chi_3 \geq 1)$[**] denotes a positive

[‖]The formula corresponds to the response chain pattern presented by Dwyer *et al.* [32].

[**]$\mathtt{B}_1$ in the dependency $\mathtt{B}_1 \rightsquigarrow \mathtt{B}_3$ means an occurrence of the block $\mathtt{B}_1$, and it therefore can be represented as $\chi_1 \geq 1$.

dependency $\mathtt{B}_1 \rightsquigarrow \mathtt{B}_3$ including only one triggering basic block. Temporal operator $\mathbf{F}$ may be removed from the formula because $\mathtt{B}_1$ is always executed prior to $\mathtt{B}_3$ in all valid execution paths of the program as shown in Figure 1. Temporal operator $\mathbf{G}$ can also be omitted because $(\chi_1 \geq 1 \rightarrow \chi_3 \geq 1)$ is an invariant in which execution counts do not vary once an execution path is selected. Therefore, flow predicate $(\chi_1 \geq 1 \rightarrow \chi_3 \geq 1)$ is sufficient. This example illustrates that possibility of translating path constraints into flow predicates depends on the structure of $\overline{P}$.

Definition 4 (*Prior node*). Node p **precedes** q if $p \rightarrow^\star q \wedge q \not\rightarrow^\star p$, where $p \rightarrow^\star q \iff (p,q) \in E^\star$.

Definition 4 is essential when enforcing an ordering relationship between two nodes. Node p precedes q if p never occurs after q in all the valid execution paths of $\overline{P}$. In Figure 1, as $\mathtt{B}_1$ precedes $\mathtt{B}_4$, flow predicates on the path constraints between $\mathtt{B}_1$ and $\mathtt{B}_4$ need not explicitly specify the order between them. On the contrary, $\mathtt{B}_4$ does not precede $\mathtt{B}_5$, and flow predicates on $\mathtt{B}_4$ and $\mathtt{B}_5$ must include the ordering information. Note that the notion of prior node does not necessarily require the occurrence of nodes; that is, $\mathtt{B}_4$ still precedes $\mathtt{B}_7$ even though some paths may not contain either or both of the nodes.

Lemma 4.1 defines the condition under which temporal operators $\mathbf{G}$ and $\mathbf{F}$ may be removed from the LTL formula corresponding to a positive dependency. Assume that there is a simple path constraint $\mathbf{G}(t \rightarrow \mathbf{F}c)$ and t precedes c. If an execution path w contains both nodes, the order between them can be statically determined since t precedes c. Temporal operator $\mathbf{F}$, therefore, can be removed from the formula safely. Predicate $\chi_t \geq k$ is an invariant because the total execution counts, χ_t, do not vary in an execution path. Therefore, the operator $\mathbf{G}$ may be omitted when applied to execution counts. Similarly, Lemma 4.2 is for a negative dependency.

Lemma 4.1. *If node t precedes c,* $w \vDash \mathbf{G}(t \rightarrow \mathbf{F}c)$ *if and only if* $|w|_t \geq 1 \rightarrow |w|_c \geq 1$.

Proof of Lemma 4.1. Appendix A.1. □

Lemma 4.2. *If node t precedes c,* $w \vDash \mathbf{G}(t \rightarrow \mathbf{G}\neg c)$ *if and only if* $|w|_t \geq 1 \rightarrow |w|_c = 0$.

Proof of Lemma 4.2. Appendix A.2. □

For instance, path constraints $\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$ and $\mathtt{B}_5 \rightsquigarrow \mathtt{B}_7$ of Figure 1 are equivalent to predicates $(\chi_4 \geq 1 \rightarrow \chi_7 = 0)$ and $(\chi_5 \geq 1 \rightarrow \chi_7 \geq 1)$, respectively, since $\mathtt{B}_4$ and $\mathtt{B}_5$ are prior nodes to $\mathtt{B}_7$. On the other hand, $\mathtt{B}_4 \rightsquigarrow \mathtt{B}_5$ is not equivalent to $(\chi_4 \geq 1 \rightarrow \chi_5 \geq 1)$ since $\mathtt{B}_4$ is not a prior node to $\mathtt{B}_5$: this predicate is satisfied by an execution path in which $\mathtt{B}_5$ never occurs after $\mathtt{B}_4$.

Unfortunately, predicates defined in Lemmas 4.1 and 4.2 each contain a disjunction as a form of implication. They, therefore, cannot be used directly in the IPET calculation without applying the separation technique [7]. Nevertheless, the disjunction for a positive dependency can be eliminated, and a single flow fact is sufficient to represent the dependency.

Figure 5 illustrates the basic idea. Black and white dots depict solutions to $(\chi_5 \geq 1 \rightarrow \chi_7 \geq 1)$. All the other points, non-solutions, are marked $\times$. However, information on finiteness, $\mathscr{V}(\mathtt{B}_5) = 3$ and $\mathscr{V}(\mathtt{B}_7) = 1$, limits the solution space only to the black dots. The shaded convex area in Figure 5 contains only and all the solutions while containing none of the $\times$ marks. Thus, the union of three linear constraints $(\chi_5 \leq 3)$, $(\chi_7 \leq 1)$, and $(\chi_5 \leq 3 \cdot \chi_7)$ enclosing the area is equivalent to $(\chi_5 \geq 1 \rightarrow \chi_7 \geq 1)$, in that both have the same solution space (χ_5, χ_7). Note that this holds since χ_i

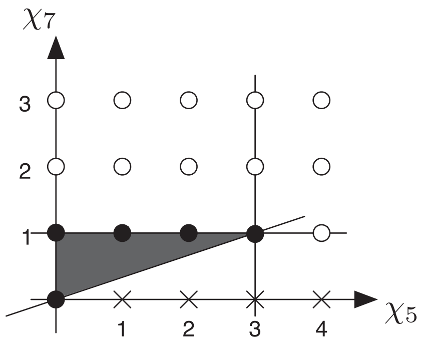

Figure 5. Solution space for $\mathtt{B}_5 \rightsquigarrow \mathtt{B}_7$: $\chi_5 \leq 3 \cdot \chi_7$.

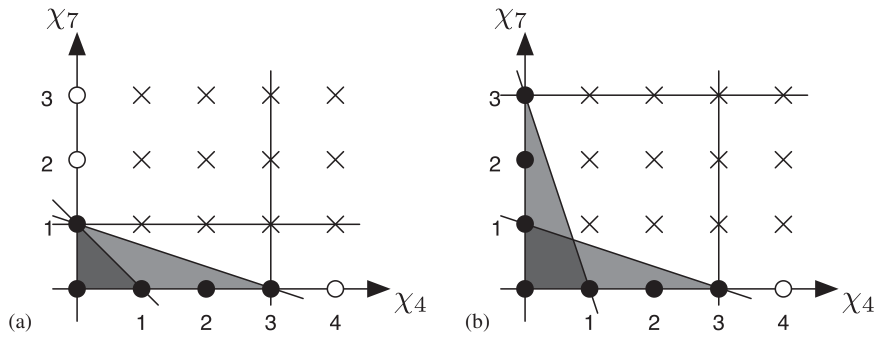

Figure 6. Solution spaces of negative dependencies: (a) $\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$: $3-3\cdot\chi_7 \geq \chi_4$ and (b) $\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$: $3-3\cdot\chi_7 \geq \chi_4 \vee 3-3\cdot\chi_4 \geq \chi_7$.

are non-negative integers. Because the first two constraints correspond to the finiteness property of $\mathtt{B}_5$ and $\mathtt{B}_7$, $(\chi_5 \leq 3\cdot\chi_7)$ is the only flow fact to be added to encode the path constraint $\mathtt{B}_5 \rightsquigarrow \mathtt{B}_7$.

This idea can be generalized in that positive dependency $t \rightsquigarrow c$ can be translated into a flow fact $(\chi_t \leq \mathscr{V}(t)\cdot\chi_c)$ if t precedes c, because a predicate for a positive dependency always has a convex solution space. Note that a concave region requires disjunctions since it is the union of convex areas.

Figure 6(a) illustrates how flow facts corresponding to negative dependencies can be derived. Negative dependency $\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$ can be rewritten as $(\chi_4 \geq 1 \rightarrow \chi_7 = 0)$ or $(\chi_4 = 0 \vee \chi_7 = 0)$. The solution space is shown in black dots when $\mathscr{V}(\mathtt{B}_4) = 3$ and $\mathscr{V}(\mathtt{B}_7) = 1$, and it is covered by a convex area enclosed by constraints $(\chi_4 \leq 3)$, $(\chi_7 \leq 1)$, and $(3-\chi_4 \geq 3\cdot\chi_7)$.

Unfortunately, this idea cannot be generalized to all negative dependencies since solution space for a negative dependency is not guaranteed to be a convex; it may be a concave. For example, if $\mathscr{V}(\mathtt{B}_4) = \mathscr{V}(\mathtt{B}_7) = 3$, the solution space becomes a concave region as shown in Figure 6(b). This region is the union of two convex areas each of which is represented by the following constraints: (i) $\{(\chi_7 \leq 3-3\cdot\chi_4), (0 \leq \chi_4 \leq 1), (0 \leq \chi_7 \leq 3)\}$ and (ii) $\{(\chi_4 \leq 3-3\cdot\chi_7), (0 \leq \chi_4 \leq 3), (0 \leq \chi_7 \leq 1)\}$. Thus, two disjunctive linear constraints $(\chi_4 \leq -3\cdot\chi_7 + 3)$ and $(3-3\cdot\chi_4 \geq \chi_7)$ must be added.

In general, a negative dependency $t \not\rightsquigarrow c$ is equivalent to $(\chi_t \le \mathscr{V}(t) \cdot (1-\chi_c)) \vee (\chi_c \le \mathscr{V}(c) \cdot (1-t))$, if node t precedes c. If either $\mathscr{V}(c)=1$ or $\mathscr{V}(t)=1$ holds as Figure 6(a), the solution space for a negative dependency becomes a convex area because one area corresponding to $(c \le \mathscr{V}(c) \cdot (1-t))$ includes the other area $(t \le \mathscr{V}(t) \cdot (1-c))$ or vice versa; that is, the predicate can be simplified as a flow fact $(\chi_t \le \mathscr{V}(t) \cdot (1-\chi_c))$ if $\mathscr{V}(c)=1$, or $(\chi_c \le \mathscr{V}(c) \cdot (1-\chi_t))$ if $\mathscr{V}(t)=1$. The example in Figure 6(a) is the case requiring only one flow fact $(\chi_4 \le 3 \cdot (1-\chi_7))$.

Note that the solutions of a negative dependency $t \not\rightsquigarrow c$ can be represented as an alternative predicate $(\chi_t = 0 \vee \chi_c = 0)$. This predicate containing one disjunction cannot be reduced to a single flow fact even though the same condition $\mathscr{V}(t)=1 \vee \mathscr{V}(c)=1$ holds.

The following theorems define the simplified versions of flow predicates corresponding to the Lemmas 4.1 and 4.2, respectively.

Theorem 4.1. *A positive dependency $t \rightsquigarrow c$ is equivalent to $(\chi_t \le \mathscr{V}(t) \cdot \chi_c)$ if node t precedes c.*

Proof of Theorem 4.1. Appendix A.3. □

Theorem 4.2. *A negative dependency $t \not\rightsquigarrow c$ is equivalent to $(\chi_t \le \mathscr{V}(t) \cdot (1-\chi_c)) \vee (\chi_c \le \mathscr{V}(c) \cdot (1-t))$, if node t precedes c.*

Proof of Theorem 4.2. Appendix A.4. □

4.3. Encoding of complex path constraints

The proposed technique on the translation of simple path constraints can be generalized with multiple triggers, namely, $t_1 t_2 \ldots t_n \rightsquigarrow c$. As all the triggering blocks must be executed in specific order, the notion of *dominance*, used in compilers and data flow analysis [33,34], is applicable. Occurrence of one node guarantees the prior occurrence of the other node if there exists a dominance relationship between them.

Definition 5 (*Dominance*). Let a bounded program be $\overline{P} = (Stmts, Blks, E, b_{entry}, B_{exit}, \mathscr{V})$. Node $q_i \in Blks$ **dominates** $q_j \in Blks$ if and only if every prefix of execution paths of $\overline{P}$ from q_0 to q_j contains q_i.

If all the consecutive pairs of triggers satisfy dominance relation in the specified order, a sequence of triggers is replaced with the last trigger. That is, $t_1 \circ t_2 \circ \cdots \circ t_n \rightsquigarrow c$ is reduced to $t_n \rightsquigarrow c$ if t_1 dominates t_2, t_2 dominates t_3, and so on. Because the resulting path constraint has a single trigger, the predicate for simple path constraints is sufficient to represent it. Lemma 4.3 presents this idea.

Lemma 4.3. *Path constraints $t_1 \circ t_2 \circ \cdots \circ t_n \rightsquigarrow c$ and $t_1 \circ t_2 \circ \cdots \circ t_n \not\rightsquigarrow c$ are equivalent to the path constraint $t_n \rightsquigarrow c$ and $t_n \not\rightsquigarrow c$, respectively, if $\forall i \wedge (1 \le i \le n-1)$: t_i dominates t_{i+1}.*

Proof of Lemma 4.3. Appendix A.5. □

Lemma 4.3 allows the conversion of complex path constraints containing multiple triggers into flow facts containing at most one disjunction. Theorems 4.3 and 4.4 generalize the Theorems 4.1 and 4.2 with respect to Lemma 4.3. Using these theorems, $\mathtt{B}_1\mathtt{B}_5 \rightsquigarrow \mathtt{B}_7$ can be translated into $(\chi_5 \le 3 \cdot \chi_7)$ since $\mathtt{B}_1$ dominates $\mathtt{B}_5$ and $\mathtt{B}_5$ precedes $\mathtt{B}_7$. Similarly, $\mathtt{B}_1\mathtt{B}_4 \not\rightsquigarrow \mathtt{B}_7$ is encoded as $(3 - 3 \cdot \chi_7 \ge \chi_4)$.

Theorem 4.3. *A positive dependency $t_1 \ldots t_n \rightsquigarrow c$ is equivalent to $(\chi_{t_n} \leq \mathscr{V}(t_n) \cdot \chi_c)$, if the node t is a prior node to the node c and $\forall i \wedge (1 \leq i \leq n-1): t_i$ dominates t_{i+1}.*

Proof of Theorem 4.3. Trivially true by Lemma 4.3 and Theorem 4.1. □

Theorem 4.4. *A negative dependency $t_1 \cdots t_n \not\rightsquigarrow c$ is equivalent to $(\chi_{t_n} \leq \mathscr{V}(t_n) \cdot (1-\chi_c)) \vee (\chi_c \leq \mathscr{V}(c) \cdot (1-\chi_{t_n}))$, if the node t_n is a prior node of c and $\forall i \wedge (1 \leq i \leq n-1): t_i$ dominates t_{i+1}.*

Proof of Theorem 4.4. Trivially true by Lemma 4.3 and Theorem 4.2. □

5. AUTOMATIC ELICITATION OF FUNCTIONAL FLOW FACTS

Tight integration of the proposed technique with IPET requires two steps: (i) identification of positive and negative dependencies and (ii) conversion of the resulting dependencies to flow facts. This section presents the techniques with algorithms applicable to these steps and tool implementation issues.

5.1. Automated dependency identification

Traditionally, identification of dependency relied on time-consuming and potentially error-prone user annotation. Automatic identification of a complete set of dependency is theoretically undecidable. The proposed technique automatically derives partial information, which is still useful to make estimation more accurate and to alleviate annotation efforts.

Figure 7 shows an overview of the proposed approach. In the figure, the automatic dependency identifier utilizes the information on infeasible paths to identify negative dependencies. Gustafsson and Ermedahl [16] proposed a static analysis technique to automatically identify loop bounds. They unfolded loops and performed flow- and context- sensitive data flow analysis based on abstract interpretation [18]. This technique is still applicable to identify infeasible paths with minor modification.

Unfortunately, the resulting infeasible paths tend to be lengthy and verbose, which are inadequate for identifying the dependency underlying it. Thus, Path Slicer eliminates irrelevant statements (or

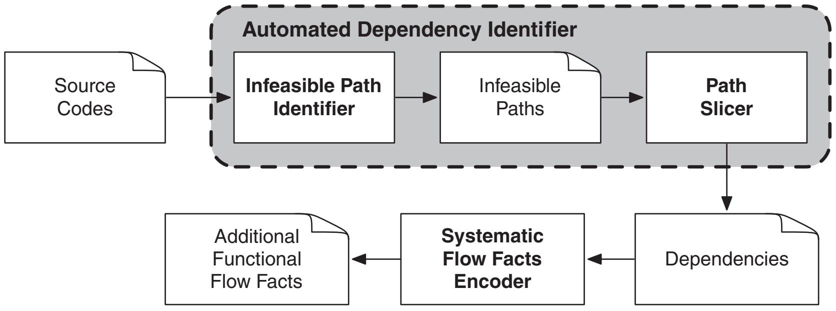
Automated Dependency Identifier
Source Codes
Infeasible Path Identifier
Infeasible Paths
Path Slicer
Additional Functional Flow Facts
Systematic Flow Facts Encoder
Dependencies

Figure 7. Approach on elicitation of functional flow facts.

blocks) to make the identification of dependency more effective. The Path Slicer, based on research reported in [19], summarizes infeasible paths into abstract ones using sets of post-dominants and reachability relations for each statement.

An abstract infeasible path is equivalent to a negative dependency if the remaining statements of a path are grouped into basic blocks. For instance, an abstract infeasible path passing through basic blocks $\mathtt{B}_1\mathtt{B}_2\mathtt{B}_3\ldots\mathtt{B}_\mathtt{n}$ corresponds to the negative dependency $\mathtt{B}_1\mathtt{B}_2\mathtt{B}_3\ldots\not\rightsquigarrow\mathtt{B}_\mathtt{n}$ because basic blocks in an infeasible path cannot be executed simultaneously.

5.2. Systematic encoding of functional flow facts

Systematic flow facts encoder implements Theorem 4.4 by checking whether the translatability conditions hold and generating the corresponding flow facts. While testing the translatability, it uses information on dominance (DOM) and reachability relations (REACH). Information on reachability relation generated by Path Slicer is used to avoid redundant computation.

Algorithm 1 Algorithm for flow fact encoder for negative dependency

```
Input $\overline{P}$=($Stmts$, $Blks$, $E$, $b_{entry}$, $B_{exit}$, $\mathscr{V}$)
Output A flow predicate for a negative dependency
1: procedure FLOWFACTENCODERNEG($tg, c$)
   // $tg$ : list of triggering blocks $t_0t_1\cdots t_n$,
   // $c$ : a consequence block.
2:     $\pi \leftarrow t_0t_1\ldots t_nc$
3:     $\mathscr{B} \leftarrow$ a set of all the involving blocks in the list $\pi$.
4:     if ISDOMINATE(tg) = true $\wedge$ $t_n \notin$ REACH(c) then
           // if lemma 4.3 and theorem 4.2 is satisfied
5:         if $\mathscr{V}(t_n) \leq 1$ then
6:             return $\left(\chi_c \leq \mathscr{V}(c)\cdot(1-\chi_{t_n})\right)$
7:         else if $\mathscr{V}(c) \leq 1$ then
8:             return $\left(\chi_{t_n} \leq \mathscr{V}(t_n)\cdot(1-\chi_c)\right)$
9:         else
10:            return $\left(\chi_{t_n} \leq \mathscr{V}(t_n)\cdot(1-\chi_c)\right) \vee \left(\chi_c \leq \mathscr{V}(c)\cdot(1-\chi_{t_n})\right)$
11:        end if
12:    else
           // ISDOMINATE(tg) $\neq$ true or $t_n$ does not precede $c$
13:        return null// naive encoding of the negative dependency
14:    end if
15: end procedure
```

Algorithm 1 shows an implementation of Flow Fact Encoder for a negative dependency. The algorithm tests the translatability of a given dependency at line 4. If the test is passed, it generates flow facts for the dependency according to the bound of blocks at lines 6, 8, and 10. Algorithm 1 utilizes function ISDOMINATE() in Algorithm 2 to determine the translatability. In the algorithms,

Algorithm 2 Algorithm for IsDominate(tg)

```
Input  Dom: Blks→$2^{Blks}$ // Dominant sets
1:  procedure IsDominate($tg$)
    // a list of triggering blocks $tg = t_0 t_1 \ldots t_n$
2:      $old \leftarrow t_0$, $i \leftarrow 1$
3:      while $i \leq n$ do
4:          if $old \notin$ Dom($t_n$) then
5:              return false
6:          end if
7:          $old \leftarrow t_i$, $i \leftarrow i+1$
8:      end while
9:      return true
10: end procedure
```

the two sets Dom and Reach are fix-point solutions of the following equations, where $preds(b)$ and $succs(b)$ are predecessors and successors of block b.

$$\textsc{Dom}(b_0) = \{b_0\} \quad \forall b \in Blks.\textsc{Reach}(t) = \{t\}$$

$$\textsc{Dom}(b) = \left(\bigcap_{p \in preds(b)} \textsc{Dom}(p) \right) \cup \{b\}, \quad \textsc{Reach}(n) = \left(\bigcup_{p \in succs(n)} \textsc{Reach}(p) \right)$$

These equations can be computed using generic worklist algorithms [35].

5.3. An example: automatic generation of flow facts

Figures 8 shows an example code and an concrete infeasible path $s_2 s_3 s_7 \ldots s_{19} s_{20}$ generated by Infeasible Path Identifier, where s_i is the statement at the ith line in the code. In the infeasible path, the condition at line 19 is never satisfied. This infeasible path corresponds to negative dependency $\psi : \mathtt{B}_1\mathtt{B}_3 \ldots \mathtt{B}_8 \not\rightsquigarrow \mathtt{B}_9$ because $\mathtt{stmt\_of}(\mathtt{B}_1, 0) = s_2$, $\mathtt{stmt\_of}(\mathtt{B}_1, 1) = s_3$, $\mathtt{stmt\_of}(\mathtt{B}_3, 0) = s_7, \ldots$. For the dependency to be encoded into flow facts, it must satisfy the following conditions (Theorem 4.4): (1) blocks $\mathtt{B}_1$–$\mathtt{B}_8$ satisfy the dominant relation and (2) $\mathtt{B}_8$ precedes $\mathtt{B}_9$. Unfortunately, this is not the case because the first condition does not hold.

However, some statements included in the path are irrelevant with respect to the condition ($\mathtt{c} > 0$) at line 19. Because the loop, lines 10–18, does not change the value of variable c, statements s_{10}–s_{18} can be safely removed from the path, thereby resulting in a negative dependency $\psi' : \mathtt{B}_1\mathtt{B}_3\mathtt{B}_8 \not\rightsquigarrow \mathtt{B}_9$.

To generate more compact negative dependencies automatically, Path Slicer generates an abstract path from a concrete one while preserving its property on feasibility; that is, a concrete path is (in)feasible if its sliced counterpart is (in)feasible. While traversing the infeasible path in Figure 8(b), *Path Slicer* removed the shaded nodes from the path and returned even more compact negative dependency $\psi' : \mathtt{B}_1\mathtt{B}_3 \not\rightsquigarrow \mathtt{B}_9$. It satisfies the translatability conditions and, therefore, functional flow fact $\chi_9 \leq (1 - \chi_3)$ is generated.

(a)

```
 1  foo() {
 2    int b=0, c=0, a=-1; // B1
 3    if(a>0) {
 4      c=1;              // B2
 5      a=100;
 6    } else {
 7      c=-1;             // B3
 8      a=-100;
 9    }
10    while(a!=0) {       // B4
11      if(a>0) {         // B5
12        b+=a;           // B6
13        a=a-1;
14      } else {          // B7
15        b-=a;
16        a=a+1;
17      }
18    }
19    if(c>0) {           // B8
20      a=b;              // B9
21    }
22    return a;           // B10
23  }
```

(b)

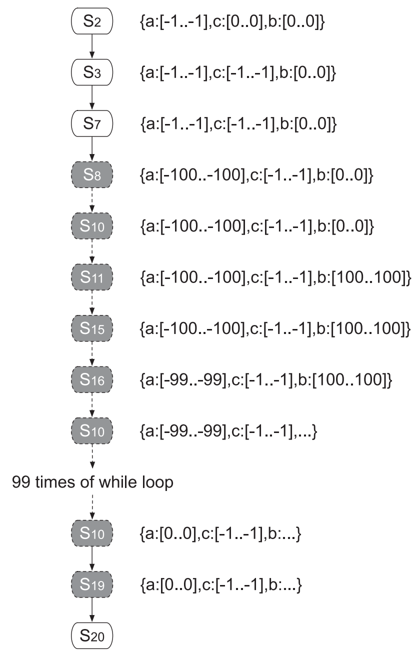
S2 {a:[-1..-1],c:[0..0],b:[0..0]}
S3 {a:[-1..-1],c:[-1..-1],b:[0..0]}
S7 {a:[-1..-1],c:[-1..-1],b:[0..0]}
S8 {a:[-100..-100],c:[-1..-1],b:[0..0]}
S10 {a:[-100..-100],c:[-1..-1],b:[0..0]}
S11 {a:[-100..-100],c:[-1..-1],b:[100..100]}
S15 {a:[-100..-100],c:[-1..-1],b:[100..100]}
S16 {a:[-99..-99],c:[-1..-1],b:[100..100]}
S10 {a:[-99..-99],c:[-1..-1],...}
99 times of while loop
S10 {a:[0..0],c:[-1..-1],b:...}
S19 {a:[0..0],c:[-1..-1],b:...}
S20

Figure 8. Infeasible path and path slicing: (a) example 2 and (b) an infeasible path.

Not all of the sliced negative dependencies can be encoded into flow facts, and user annotations are unavoidable in such cases. While the separation technique may be applied, it is impractical; our experiment revealed that up to 10^{93} ILP computations are needed to compute all the possible combinations of about 84 negative dependencies, which contain 13 basic blocks on average. Nevertheless, sliced paths are easy to understand in that they contain only essential information on causality, and they are useful when annotating flow facts.

6. EXPERIMENTAL RESULTS

Four benchmark suites[††] GNU OpenSSH[‡‡], and Command and Communication Interface (CCI) software were used to measure the effectiveness of the proposed technique. Four benchmark suites are commonly used in the researches on WCET estimation. GNU OpenSSH consists of network connectivity tools using SSH protocol. CCI software, deployed in Korea multi-purpose satellite (KOMPSAT-1) was developed by Korea Aerospace Research Institute (KARI) to receive and process tele-commands periodically.

[††] http://www.c-lab.de/index.php?id=462&L=3.
[‡‡] http://www.openssh.org.

Table II. Experimental results.

Benchmarks	LOC	Infeasible paths	Neg. dependencies	Functional flow facts	Coverage* (%)
Gothenburg	2299	497	329	81	25
FloridaSt	1456	8531	938	239	25
Uppsala	6494	1434	1373	51	4
Seoul National University	3465	361	313	184	59
CCI	5394	20	16	10	63
OpenSSH	65 530	1888	1362	80	6
Average	14 106.3	2121.8	721.8	107.5	30.2

*Coverage: ratio of infeasible paths covered by functional flow facts.

Table II shows the experimental results. Note that infeasible paths are concrete paths and negative dependencies are abstract ones, as mentioned previously. Because one abstract path may represent multiple concrete paths, the number of negative dependencies is smaller than that of infeasible paths.

Among the infeasible paths generated, nearly 30% of them were automatically converted into functional flow facts using the proposed technique. In benchmarks Seoul National University and CCI, functional flow facts covering more than 50% of the infeasible paths were derived automatically. These flow facts increase the tightness of WCET estimate without expensive computational cost; in that the number of IPET computation doubles for every disjunction included in flow facts.

Unfortunately, benchmarks Uppsala and OpenSSH demonstrated poor coverage of less than 10%. Although the performance of IPET computation could still be improved by these functional flow facts and still reduce the penalties caused by disjunctions, uncovered cases contained many simple dependencies that are convertible to flow facts.

Detailed inspection revealed that the infeasible path generation technique is still not advanced enough to identify infeasible paths in complex control and data flows. More accurate and scalable emerging techniques for software model checkers and static program analyzers [36–38] could increase the performance of the generation.

On the other hand, effectiveness of the path slicing technique seriously depended on the characteristics of infeasible paths identified. In some case, the slicing failed to simplify some infeasible paths into compact dependencies that the proposed technique can be applied.

For example, in the case of *FloridaSt* benchmark, path slicing dramatically reduced the number of infeasible paths by nearly 90%; that is, 938 negative dependencies captured 8531 concrete infeasible paths. The slicing technique was particularly effective on this program because it had a sequence of loops and there were numerous routes by which a trivially infeasible statement could be reached.

However, the static path slicing technique is unable to eliminate semantically irrelevant statements. In the below code, for instance, the first `if`-statement (lines 1–5) updates the value of variable `a` on both branches. Therefore, the static slicing algorithm is unable to reduce either of the two infeasible paths $s_0s_1s_2s_6s_7$ and $s_0s_1s_4s_6s_7$. Semantic analysis, however, clearly reveals that s_7 is never executed due to the initialization of variable `a` at line 0. Had a forbidden path $s_0 \rightsquigarrow s_7$ been derived, flow facts would have been automatically generated.

Table III. Number of ILP calculation required for Gothenburg benchmark suite.

Functions	Identified dependencies	Separation technique (# of ILP solving)	Proposed technique Encoded	Proposed technique Failed	Proposed technique # of ILP solving
f1	1	2	1	0	1
f2	6	180	4	2	30
f3	8	2520	8	0	1
f4	12	1.04E+06	12	0	1
f5	68	2.11E+41	24	44	3.93E+33
f6	211	1.36E+277	9	202	3.34E+271
f7	11	967 680	11	0	1
f8	3	6	3	0	1
f9	3	15	3	0	1
f10	3	6	3	0	1
f11	3	6	3	0	1

```
a = −100;
if(b>0) {
  a++;
} else {
  a=a+2;
}
if(a>0) {
  ...
}
```

To alleviate this problem, the Path Slicer must be able to identify the fundamental cause–effect relationship with respect to infeasible paths. Research on summarization technique such as shortest counterexample generation [39] and error explanation [40] could help to achieve shorter and more simple path constraints and increase the encoding ratio. This adaptation still remains as a further work.

Table III shows the number of ILP calculation required for the Gothenburg benchmark in detail. The separation technique required more than 100 calculations for six functions. However, our technique requires only one computation for eight functions. For example, WCET of $f2$ can be obtained roughly 6 times faster than the separation method when using our technique. Unfortunately, two functions $f5$ and $f6$ still introduced many disjunctions because few of their path constraints can be encoded in our technique and the separation technique applied to.

7. CONCLUSION

This paper proposed a systematic encoding technique for path constraints, positive and negative dependencies. If all the triggers satisfy the dominance relation and the last trigger precedes the consequence node, a positive dependency can be translated into a flow fact. A negative dependency can also be encoded into a flow predicate with at most one disjunction. Such translation has been automated, and the amount of ILP computation can be reduced significantly.

While not all the path constraints can be automatically translated into flow facts, the contribution of this paper is significant and of practical importance to those who must perform IPET analysis on embedded real-time software. One less disjunction in flow facts reduces the number of ILP calculation into half.

Experimental results revealed that the effectiveness of the proposed technique mainly relied on the quality of dependency generation. While path slicing technique could produce compact dependencies efficiently sometimes, syntactic methods had limitations on the dependency identification. Because improvement of effectiveness of the proposed work requires more subtle techniques on dependency generation, adaption of non-syntactic techniques including error explanation and shortest counterexample technique remains as further work.

APPENDIX A: PROOFS OF LEMMAS AND THEOREMS

A.1. Proof of Lemma 4.1

Lemma 4.1. *If node t precedes c, $w \vDash \mathbf{G}(t \to \mathbf{F}c)$ if and only if $|w|_t \geq 1 \to |w|_c \geq 1$.*

Proof of Lemma 4.1 in two directions.
$\Rightarrow$ **direction** If $w \vDash \mathbf{G}(t \to \mathbf{F}c)$, two types of execution paths w are possible.

(i) if $|w|_t = 0$, $|w|_t \geq 1 \to |w|_c \geq 1$ is trivially true.
(ii) if $|w|_t \geq 1$, w contains at least one occurrence of node c and t should be followed by c by the semantics of LTL formula. Therefore, $|w|_t \geq 1 \to |w|_c \geq 1$ is true.

$\Leftarrow$ **direction** If $|w|_t \geq 1 \to |w|_c \geq 1$ is true, the following possibilities exist:

(i) if $|w|_t = 0$, path w does not contain t and trivially $w \vDash \mathbf{G}(t \to \mathbf{F}c)$.
(ii) if $|w|_t \geq 1 \wedge |w|_c \geq 1$, let us assume an execution path w such that $|w|_t \geq 1 \wedge |w|_c \geq 1$ and $w \nvDash \mathbf{G}(t \to \mathbf{F}c)$. This path $w = q_0 q_1 \ldots q_n$ contains $q_i = t$, and c does not occur after t by the definition of prior node.
Therefore, there must exist $q_j = c$ such that $j < i$ since $|w|_c \geq 1$. This contradicts the assumption that the node t precedes c. □

A.2. Proof of Lemma 4.2

Lemma 4.2. *If node t precedes c, $w \vDash \mathbf{G}(t \to \mathbf{G}\neg c)$ if and only if $|w|_t \geq 1 \to |w|_c = 0$.*

Proof of Lemma 4.2 in two directions.
$\Rightarrow$ **direction** If $w \vDash \mathbf{G}(t \to \mathbf{G}\neg c)$, two types of execution paths w are possible.

(i) if $|w|_t = 0$, $|w|_t \geq 1 \to |w|_c \geq 0$ is trivially true.
(ii) if $|w|_t \geq 1$, $w = q_0 q_1 \ldots q_n$ contains $q_i = t$ such that $\forall j (i < j \leq n). q_j \neq c$. The following shows the possible cases of $|w|_c$.

(a) if $|w|_c = 0$, $|w|_t \geq 1 \rightarrow |w|_c = 0$ is trivially true.
(b) if $|w|_c \geq 1$, the path w contains q_j such that $j < i \wedge q_j = c$. This contradicts the assumption that the node t precedes c.

Thus, $|w|_c = 0$ if $|w|_t \geq 1$.
By (i) and (ii), $|w|_t \geq 1 \rightarrow |w|_c \geq 0$ is true.

$\Leftarrow$ **direction** If $|w|_t \geq 1 \rightarrow |w|_c = 0$ is true, two cases are possible.

(i) if $|w|_t = 0$, $w \vDash \mathbf{G}(t \rightarrow \mathbf{G}\neg c)$ is trivially true.
(ii) if $|w|_t \geq 1 \wedge |w|_c = 0$, w contains only t. Therefore, $w \vDash \mathbf{G}(t \rightarrow \mathbf{G}\neg c)$. □

A.3. Proof of Theorem 4.1

Theorem 4.1. *A positive dependency $t \rightsquigarrow c$ is equivalent to $(\chi_t \leq \mathscr{V}(t) \cdot \chi_c)$ if node t precedes c.*

Proof of Theorem 4.1. Directly derived by Lemma 4.1 and the following Lemma A.1 □

Lemma A.1. *The convex area of $(t \leq \mathscr{V}(t) \cdot c) \wedge (0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq \mathscr{V}(c))$ contains only those (t, c) such that $(t \geq 1 \rightarrow c \geq 1)$, if t and c have integer values.*

Proof of Lemma A.1. The following are equivalent to each other, when t and c are non-negative integers:

(i) $(t \geq 1 \rightarrow c \geq 1)$
(ii) $(t = 0 \vee c \geq 1)$
(iii) $(t \geq 0 \wedge c \geq 1) \vee (t = 0 \wedge c = 0)$

Figure A1 shows the convex area enclosed by $(t \leq \mathscr{V}(t) \cdot c) \wedge (0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq \mathscr{V}(c))$. In this figure, it is apparent that the rectangle A contains only those (t, c) such that $(t \geq 0 \wedge c \geq 1)$ where $(0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq \mathscr{V}(c))$. On the other hand, triangle B includes only those (t, c) such that $(t = 0 \wedge c = 0) \vee (0 \leq t \leq \mathscr{V}(t) \wedge c = 1)$, since points (t, c) satisfying the condition $(0 \leq t \leq \mathscr{V}(t) \wedge c = 1)$ are already included in the region A, and the regions A and B together include only those (t, c) satisfying condition (iii). □

A.4. Proof of Theorem 4.2

Theorem 4.2. *A negative dependency $t \not\rightsquigarrow c$ is equivalent to $(\chi_t \leq \mathscr{V}(t) \cdot (1 - \chi_c)) \vee (\chi_c \leq \mathscr{V}(c) \cdot (1 - t))$, if node t precedes c.*

Proof of Theorem 4.1. Directly derived by Lemma 4.2 and the following Lemma A.2. □

Lemma A.2. *The union of two convex areas of $\{(c \leq \mathscr{V}(c) \cdot (1 - t)) \wedge (0 \leq t \leq 1) \wedge (0 \leq c \leq \mathscr{V}(c))\}$ and $\{(t \leq \mathscr{V}(t) \cdot (1 - c)) \wedge (0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq 1)\}$ contains only those (t, c) such that $(t \geq 1 \rightarrow c = 0)$, where t and c are integers and $(0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq \mathscr{V}(c))$.*

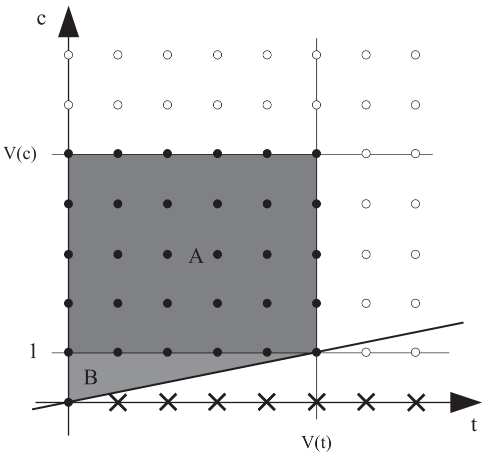

Figure A1. The convex area for a positive dependency : $t \leq \mathscr{V}(t) \cdot c$.

Proof of Lemma A.2. The followings are equivalent to each other if t and c are non-negative integers:

(i) $(t \geq 1 \rightarrow c = 0)$
(ii) $(t = 0 \wedge c \geq 0) \vee (t \geq 0 \wedge c = 0)$

Figure A2 shows the two convex areas A and B of $\{(c \leq \mathscr{V}(c) \cdot (1-t)) \wedge (0 \leq t \leq 1) \wedge (0 \leq c \leq \mathscr{V}(c))\}$ and $\{(t \leq \mathscr{V}(t) \cdot (1-c)) \wedge (0 \leq t \leq \mathscr{V}(t)) \wedge (0 \leq c \leq 1)\}$, respectively. Area A includes points (t, c) satisfying $(t = 0 \wedge c \geq 0) \vee \underline{(t = 1 \wedge c = 0)}$. Similarly, the condition $(t \geq 0 \wedge c = 0) \vee \overline{(t = 0 \wedge c = 1)}$ characterizes the area B. The $\overline{\text{union of the}}$ two areas, therefore, includes only (t, c) such that $(t \geq 1 \rightarrow c = 0)$ and the two areas together include (t, c) such that it satisfies each conjunctive term of (ii). □

A.5. Proof of Lemma 4.3

Lemma 4.3. *Path constraints $t_1 \circ t_2 \circ \cdots \circ t_n \rightsquigarrow c$ and $t_1 \circ t_2 \circ \cdots \circ t_n \not\rightsquigarrow c$ are equivalent to the path constraints $t_n \rightsquigarrow c$ and $t_n \not\rightsquigarrow c$, respectively, if $\forall i \wedge (1 \leq i \leq n-1)$: t_i dominates t_{i+1}.*

Proof of Theorem 4.3. Trivially derived by the following Lemma A.3. □

Lemma A.3. *Let a bounded program be $\overline{P}$=(Stmts, Blks, E, b_{entry}, B_{exit}, $\mathscr{V}$) and $\forall i \in \mathbb{N} \wedge (1 \leq i \leq n) : t_i \in$ Blks. If $\forall i \wedge (1 \leq i \leq n-1)$: t_i dominates t_{i+1}, $w \vDash \{t_1 \wedge \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots (t_{n-1} \wedge \mathbf{F} t_n)))\}$ if and only if $|w|_{t_n} \geq 1$.*

Proof of Lemma A.3. in two directions:
$\Rightarrow$ **direction** if $w \vDash \{t_1 \wedge \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots (t_{n-1} \wedge \mathbf{F} t_n)))\}$, $|w|_{t_n} \geq 1$ is trivially true by the semantics of the LTL formula.
$\Leftarrow$ **direction** Proof by induction on the number of triggers.

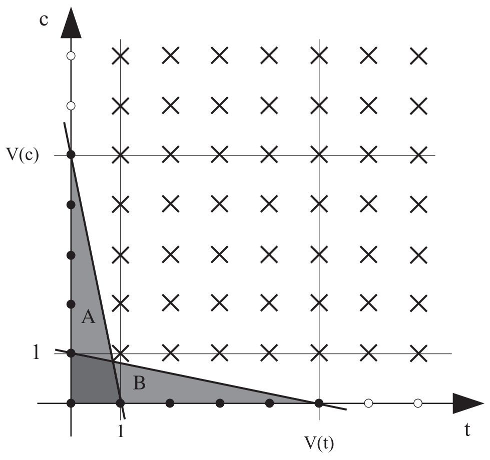

Figure A2. The two convex areas for a negative dependency : $(c \leq \mathcal{V}(c)\cdot(1-t))$ and $(t \leq \mathcal{V}(t)\cdot(1-c))$.

(i) basis$(n=1)$: if trigger is a node t, the lemma is trivially true,
(ii) induction hypothesis$(n=k-1)$: let $w \vDash \{t_1 \wedge \mathbf{F}(t_2 \wedge \cdots \mathbf{F}t_{n-1})\}$ if $|w|_{t_{n-1}} \geq 1$,
(iii) induction$(n=k)$: If $|w|_{t_k} \geq 1$ and t_{k-1} dominates t_k, t_{k-1} occurs between q_0 and t_k in w. Therefore, $|w|_{k-1} \geq 1$. By induction hypothesis, $w \vDash \{t_1 \wedge \mathbf{F}(t_2 \wedge \cdots \mathbf{F}t_{n-1})\}$. Therefore, $w \vDash \{t_1 \wedge \mathbf{F}(t_2 \wedge \mathbf{F}(\cdots(t_{n-1} \wedge \mathbf{F}t_n)))\}$ since t_{n-1} is followed by t_n by dominance relation. $\square$

ACKNOWLEDGEMENTS

This research was partially supported by the KOSEF through the AITrc and by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD060048AD. This research was partially supported by the ubiquitous Computing and Network (UCN) Project, the Ministry of Information and Communication (MIC) 21st Century Frontier R&D Program in Korea.

REFERENCES

1. Puschner P, Koza C. Calculating the maximum execution time of real-time programs. *Real-time Systems* 1989; **1**(2): 159–176. DOI: 10.1007/BF00571421.
2. Park CY, Shaw AC. Experiments with a program timing tool based on a source-level timing schema. *Proceedings of the 11th IEEE Real-time Systems Symposium* (*RTSS'90*), Lake Buena Vista, FL, U.S.A., December 1990; 72–81. DOI: 10.1109/REAL.1990.128731.
3. Pospischil G, Puschner P, Vrchoticky A, Zainlinger R. Developing real-time tasks with predictable timing. *IEEE Software* 1992; **9**(5):35–44. DOI: 10.1109/52.156895.
4. Bate I, Bernat G, Murphy G, Puschner P. Low-level analysis of a portable Java byte code WCET analysis framework. *Proceedings of the Seventh International Conference on Real-time Computing Systems and Applications* (*RTCSA'00*). IEEE Computer Society: Silver Spring, MD, December 2000; 39–46. DOI: 10.1109/RTCSA.2000.896369.
5. Healy CA, Arnold RD, Mueller F, Whalley DB, Harmon MG. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers* 1999; **48**(1):53–70. DOI: 10.1109/12.743411.

6. Stappert F, Altenbernd P. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture* 2000; **46**(4):339–355. DOI: 10.1016/S1383-7621(99)00010-7.
7. Li Y-TS, Malik S. Performance analysis of embedded software using implicit path enumeration. *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (*DAC'95*), San Francisco, CA, U.S.A., 1995; 456–461. DOI: 10.1145/217474.217570.
8. Puschner P, Schedl AV. Computing maximum task execution times with linear programming techniques. *Technical Report*, Technische Universität Wien, Institut fur Technische Informatik, April 1995.
9. Engblom J, Ermedahl A. Modeling complex flows for worst-case execution time analysis. *Proceedings of 21st IEEE Real-time Systems Symposium*, Orlando, FL, U.S.A. IEEE Press: New York, November 2000; 163–174. DOI: 10.1109/REAL.2000.896006.
10. Tan L. The worst-case execution time tool challenge 2006. *International Journal on Software Tools for Technology Transfer* (*STTT*) 2009; **11**(2):133–152. DOI: 10.1007/s10009-008-0095-9.
11. Park CY. Predicting program execution times by analyzing static and dynamic program paths. *Real-time Systems* 1993; **5**(1):31–62. DOI: 10.1007/BF01088696.
12. Ferdinand C, Martin F, Wilhelm R. Applying compiler techniques to cache behavior prediction. *ACM SIGPLAN Workshop on Languages*, *Compilers and Tools for Real-time Systems* (*LCT-RTS'97*), Las Vegas, Nevada, U.S.A., 1997; 37–46.
13. Holsti N, Långbacka T, Saarinen S. Worst-case execution-time analysis for digital signal processors. *Proceedings of the 10th European Signal Processing Conference* (*EUSIPCO 2000*), Tampere, Finland, September 2000; 2469–2472.
14. Colin A, Bernat G. Scope-Tree: A program representation for symbolic worst-case execution time analysis. *Proceedings of the 14th Euromicro Conference on Real-time Systems* (*ECRTS'02*). IEEE Computer Society: Silver Spring, MD, 2002; 50–59. DOI: 10.1109/EMRTS.2002.1019185.
15. Chapman R, Burns A, Wellings A. Integrated program proof and worst-case timing analysis of SPARK Ada. *ACM SIGPLAN Workshop on Languages*, *Compilers and Tools for Real-time Systems* (*LCT-RTS'94*), Washington, DC, U.S.A., June 1994.
16. Gustafsson J, Ermedahl A. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Proceedings of 1997 Joint Workshop on Parallel and Distributed Real-time Systems* (*WPDRTS/OORTS'97*), Geneva, Switzerland, 1997; 257–262. DOI: 10.1109/WPDRTS.1997.637989.
17. Lundqvist T, Stenström P. Integrating path and timing analysis using instruction-level simulation techniques. *Proceedings of the ACM SIGPLAN Workshop on Languages*, *Compilers*, *and Tools for Embedded Systems* (*LCTES'98*) (*Lecture Notes in Computer Science*, vol. 1474), Montreal, Canada. Springer: Berlin, June 1998; 1–15. DOI: 10.1007/BFb0057775.
18. Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the Fourth Annual SIGPLAN-SIGACT Symposium on Principles of Program Languages* (*POPL*'77), Los Angeles, CA, ACM, 1977; 238–252. DOI: 10.1145/512950.512973.
19. Jhala R, Majumdar R. Path slicing. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'05*), Chicago, IL, U.S.A., ACM, June 2005; 38–47. DOI: 10.1145/1065010.1065016.
20. Allen Emerson E. Temporal and modal logic. *Formal Models and Semantics* (*Handbook of Theoretical Computer Science*, vol. B), van Leeuwen J (ed.). Elsevier Science: Amsterdam, 1990.
21. Stewart DB. Measuring execution time and real-time performance. *Proceedings of Embedded Systems Conference 2001*, San Francisco, U.S.A., April 2001.
22. Bernat G, Colin A, Petters SM. WCET analysis of probabilistic hard real-time systems. *Proceedings of the 23rd IEEE Real-Time Systems Symposium* (*RTSS'02*). IEEE Computer Society: Silver Spring, 2002; 279–288. DOI: 10.1109/REAL.2002.1181582.
23. Ottosson G, Sjödin M. Worst-case execution time analysis for modern hardware architectures. *Proceedings of ACM SIGPLAN Workshop on Languages*, *Compilers and Tools for Real-time Systems* (*LCT-RTS'97*), Las Vegas, NV, U.S.A., June 1997.
24. Li Y-TS, Malik S, Wolfe A. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems* (*TODAES*) 1999; **4**(3):257–279. DOI: 10.1145/315773.315778.
25. Sebek F. Cache memories in real-time systems. *Technical Report 01/37*, Mälardalen Real-time Research Centre, Department of Computer Engineering, Mälardalen University, Sweden, 2001.
26. Colin A, Puaut I. Worst case execution time analysis for a processor with branch prediction. *Real-time Systems* 2000; **18**(2–3):249–274. DOI: 10.1023/A:1008149332687.
27. Bate I, Reutemann R. Worst-case execution time analysis for dynamic branch predictors. *Proceedings of 16th Euromicro Conference on Real-time Systems* (*ECRTS'04*). IEEE Computer Society: Silver Spring, 2004; 215–222. DOI: 10.1109/EMRTS.2004.1311023.
28. Burguiere C, Rochange C. A contribution to branch prediction modeling in WCET analysis. *Proceedings of Design*, *Automation and Test in Europe* (*DATE'05*). IEEE Computer Society: Silver Spring, 2005; 612–617. DOI: 10.1109/DATE.2005.7.

29. Ermedahl A, Gustafsson J. Deriving annotations for tight calculation of execution time. *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing* (*Euro-Par'97*) (*Lecture Notes in Computer Science*, vol. 1300). Springer: Berlin, Heidelberg, 1997; 1298–1307. DOI: 10.1007/BFb0002886.
30. Healy C, Sjödin M, Rustagi V, Whalley D. Bounding loop iterations for timing analysis. *Proceedings of the Fourth IEEE Real-time Technology and Applications Symposium* (*RTAS'98*). IEEE Computer Society: Silver Spring, 1998; 12–21. DOI: 10.1109/RTTAS.1998.683183.
31. Altenbernd P. On the false path problem in hard real-time programs. *Proceedings of the Eighth Euromicro Workshop on Real Time Systems*. Nova Science Publishers, Inc., June 1996; 102–107. DOI: 10.1109/EMWRTS.1996.557827.
32. Dwyer MB, Avrunin GS, Corbett JC. Property specification patterns for finite-state verification. *Proceedings of the 21st International Conference on Software Engineering* (*ICSE'99*), Los Angeles, CA, U.S.A. IEEE Computer Society: Silver Spring, 1999; 411–420.
33. Hecht MS. *Flow Analysis of Computer Programs*, Programming Language Series. Elsevier North-Holland: Amsterdam, 1977.
34. Aho AV, Sethi R, Ullman JD. *Compilers*: *Principles*, *Techniques*, *and Tools*. Addison-Wesley: Reading, MA, 1986.
35. Nielson F, Riis Nielson H, Hankin C. *Principles of Programming Analysis*. Springer: Berlin, 1999.
36. Ball T, Rajamani SK. Automatically validating temporal safety properties of interfaces. *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software* (*SPIN'01*) (*Lecture Notes in Computer Science*, vol. 2057), Toronto, ON, Canada. Springer: New York, May 2001; 103–122.
37. Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. *Proceedings of the 29th Annual Symposium on Principles of Programming Languages* (*POPL'02*), Portland, OR, U.S.A. ACM, 2002; 58–70. DOI: 10.1145/503272.503279.
38. Chaki S, Clarke E, Groce A, Jha S, Veith H. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 2004; **30**(6):388–402. DOI: 10.1109/TSE.2004.22.
39. Schuppan V, Biere A. Shortest counterexamples for symbolic model checking of LTL with past. *Proceedings of the 11th Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'05*) (*Lecture Notes in Computer Science*, vol. 3440), Edinburgh, U.K. Springer: Berlin, Heidelberg, April 2005; 493–509. DOI: 10.1007/b107194.
40. Groce A. Error explanation with distance metrics. *Proceedings of the 10th Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'04*) (*Lecture Notes in Computer Science*, vol. 2988), Barcelona, Spain. Springer: Berlin, Heidelberg, March 2004; 108–122. DOI: 10.1007/b96393.