PAPER

Visualization and Formalization of User Constraints for Tight Estimation of Worst-Case Execution Time

Jong-In LEE^{†a)}, *Member*, Ho-Jung BANG[†], Tai-Hyo KIM[†], and Sung-Deok CHA^{††}, *Nonmembers*

SUMMARY Automated static timing analysis methods provide a safe but usually overestimated worst-case execution time (WCET) due to infeasible execution paths. In this paper, we propose a visual language, User Constraint Language (UCL), to obtain a tight WCET estimation. UCL provides intuitive visual notations with which users can easily specify various levels of flow information to characterize valid execution paths of a program. The user constraints specified in UCL are translated into finite automata. The combined automaton, constructed by a cross-production of the automata for program and user constraints, reflects the static structure and possible dynamic behavior of the program. It contains only the execution paths satisfying user constraints. A case study using part of a software program for satellite flight demonstrates the effectiveness of UCL and our approach.

key words: *worst-case execution time, user constraint, control flow graph, finite automata*

1. Introduction

Correctness of real-time embedded software not only depends on the logical correctness of the computation but also upon the time at which the results are produced. A prerequisite for temporal validation is knowledge about the upper bounds of the execution time of all time-critical tasks in the system. Static analysis estimates WCET by examining all possible execution paths and calculating the worst execution time among these paths without executing software [1]–[5]. Current methods, however, have a limitation in that they usually overestimate the WCET since they cannot fully identify the infeasible execution paths. They may also overestimate WCET if unrealistic environmental assumptions (e.g., maximum input data rate, input data pattern, or exception handling) are not fully considered. As such information cannot be automatically extracted from the program code, users need to provide additional flow information such as loop bounds or information on infeasible paths by annotating program codes [6]–[8]. However, existing annotation languages are not expressive enough to fully specify infeasible paths of programs. Furthermore, they are difficult to understand.

This paper proposes a new flow representation lan-

guage called User Constraint Language (UCL). It is an automata-based graphical language that provides intuitive and visual notations to characterize the dynamic behavior of a program. It is expressive enough to specify complex-flow information which is difficult to specify in existing languages. User constraints specified in UCL are converted into corresponding finite automata. They are combined with the automaton representing a control flow graph (CFG) of the target program through cross production. It is also neutral to back-end calculation methods, so it can be applied to the path-based [4] or Implicit Path Enumeration Technique (IPET)-based static WCET calculation method [9].

The organization of this paper is as follows. Section 2 identifies the flow information used to exclude infeasible or impractical paths and surveys flow information representation languages. Section 3 describes notations of UCL with examples illustrating their usage followed by a comparison with other languages. Section 4 presents the formal syntax and semantics of UCL, and describes a translation scheme to convert UCL formulas into finite automata. Consistency checking of UCL specifications is also described. Section 5 briefly explains how the proposed approach can be applied to a static WCET analysis framework and presents the results of an experiment using satellite flight software to demonstrate the effectiveness of our method and UCL in obtaining a tight WCET estimation. Finally, Sect. 6 gives our conclusion and outlines our future research plan.

2. Flow Information and Representation Languages

When performing static WCET analysis, all possible execution paths have to be determined and the execution time of each basic block must be accurately estimated. The set of structurally possible execution paths may be infinite if loops can be taken an arbitrary number of times. Bounding all loops with some upper bounds makes it finite. Annotating flow information, such as dependency between statements, narrows the set of feasible paths [10]–[12]. Figure 1 shows a sample program and its CFG. However, there are types of flow information that must be provided by the user. An *impractical path* is the execution path that is semantically possible in a program code but actually unrealistic if the constraints on the environment under which the target system operates are fully considered. For example, the execution time of real-time embedded software depends on the amount or type of external input. Handling of an exception may require all the remaining input data be dis-

Manuscript received October 4, 2007.

Manuscript revised March 3, 2008.

[†]The authors are with the Dependable Software Lab, Division of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea.

^{††}Sung-Deok Cha, corresponding author, is with the Faculty of Dept. of Computer Science and Engineering, Korea University, Korea.

a) E-mail: jilee@dependable.kaist.ac.kr

DOI: 10.1587/transinf.E92.D.24

carded, which also influences execution paths. The constraints on impractical paths can be extracted from the system documents, and they need to be specified as additional flow information. Code annotation languages can be implemented as an extension to programming language syntax or as simple compiler directives [6], [13]. Park [14] introduced a flow information representation language called Information Description Language (IDL). In IDL, both program path and user information are represented as a regular expression of statement labels. The constraint that basic block B5 in Fig. 1 cannot be executed more than once can be specified in IDL as ‘*execute B5 [0, 1] times inside L*’. The constraint that bound of *while* loop *L* is 10 can be specified as ‘*loop L [1, 10] times*’. IDL has a limitation in specifying the constraint related to the iteration of loops or causal dependency between them.

Engblom and Ermedahl [5] introduced a flow information representation language called Flow Fact Language (FFL) to specify flow information for the IPET-based calculation method. Users specify flow information in terms of execution counts of basic blocks. For example, the constraint that the basic block B5 cannot be executed more than once is specified in FFL as ‘*foo : [] : $x_{B5} \leq 1$* ’. The constraint that the bound of *while* loop is 10 can be specified as ‘*foo : [] : $x_{B2} \leq 11$* ’. A context specifier ‘[]’ means that the fact is considered as a sum over all iterations of the defining scope. FFL has clear and precise semantics to specify complicated user flow information. However, it is difficult to specify the path constraints on execution order among basic blocks in terms of execution counts of basic blocks. It also has a limitation in specifying a combined constraint such as ‘execution of basic block A or B must be followed by execution of basic block C’ because FFL does not directly support the Boolean operation OR.

Fig. 1 Example of infeasible path.

3. User Constraint Language

UCL is a formal and graphical language to specify user constraints. In designing UCL, we consider intuitiveness, expressiveness, and ease of analysis as criteria. For intuitiveness, UCL employs visual notations. For expressiveness, it specifies constraints of causal dependency, sequence, cardinality, and iteration. It supports a logical combination of them, too. Users can specify both static and dynamic flow information in UCL to eliminate infeasible or impractical paths. For ease of analysis, UCL is not closely connected to a particular back-end calculation method of static WCET analysis.

In Fig. 2, trigger block and triggered block correspond to the cause and effect parts of a dependency relation, which are located within the start- and end-of-constraint marks. Trigger block is not required for an unconditional constraint. OR-join, AND-join, OR-fork, or AND-fork notations are used to represent a logical combination among trigger and/or triggered blocks. Cardinality is the number of times a basic block is executed in a loop. Scope is a part of CFG whose basic blocks are connected, where user constraints are effective. Functions and loops are default scopes, while users can also define a scope. For explanation purposes, we also define several textual notations: causal relation using ‘ $\leadsto$ ’, execution sequence of basic blocks by ‘.’, and negation by ‘ $\neg$ ’. $A(n)$ denotes a basic block A with cardinality n ; $A(@_hi)$ means that the basic block A is executed at the i -th iteration of the enclosing loop whose header is h .

Figure 3 illustrates UCL notations to specify user constraints in a CFG, followed by a comparison with corresponding specifications in FFL and IDL. UCL constraints 1) and 2) show two causal dependencies that the execution of *D* triggers the execution of *G*, and that the execution of *E* should not be followed by the execution of *G*. UCL constraint 3) illustrates an OR-join causal dependency that if *L* or *R* is executed, then *X* should be executed later. UCL constraint 4) shows an AND-join causal dependency that when both *M* and *T* are executed, then *Y* should be executed later. UCL constraint 5) illustrates a sequence constraint that when two basic blocks *R* and *S* are executed in sequence, then *T* should be executed later. Note that the execution of *S* may not be followed immediately by the exe-

Fig. 2 Notations of User Constraint Language.

cution of T . UCL constraint 6) shows a cardinality constraint that the bound of loop $L1$ is $m - 1$. UCL constraint 7) shows an iteration constraint that L is executed at the first iteration of loop $L1$.

As shown in the table below diagram, FFL cannot express the causal dependency ' $L \vee R \leadsto X$ ' as ' $scope : \langle \rangle : x_L + x_R \leq x_X$ ' since both L and R can be executed more than once. The causal dependency ' $M \wedge T \leadsto Y$ ' cannot be specified as ' $scope : \langle \rangle : x_M \times x_T \leq x_Y$ ' because the execution counts of M and T may be greater than one. FFL cannot specify the sequence dependency ' $R \cdot S \leadsto T$ ', either. IDL cannot specify the iteration constraint 7). Note that FFL context specifier $[]$ means 'total' and $\langle \rangle$ means 'for each' iteration within a scope.

UCL is intuitive and user-friendly compared to source code annotation as it uses visual notations to specify user

No.	Dependency	FFL	IDL
1	Positive ($D \leadsto G$)	$scope : \langle \rangle : x_D \leq x_G$	$(D) \text{ imply } (G)$
2	Negative ($E \leadsto \neg G$)	$scope : \langle \rangle : x_E + x_G \leq 1$	$\text{nopath}(E, G)$
3	OR-join ($L \vee R \leadsto X$)	<i>Not supported</i>	$(L) \text{ or } (R) \text{ imply } (X)$
4	AND-join ($M \wedge T \leadsto Y$)	<i>Not supported</i>	$(M) \text{ and } (T) \text{ imply } (Y)$
5	Sequence ($R \cdot S \leadsto T$)	<i>Not supported</i>	$(R) \text{ imply } (S) \text{ imply } (T)$
6	Cardinality ($J[\leq m]$)	$scope : [] : x_J \leq m$	$\text{loop } L1 [1, m-1] \text{ times}$
7	Iteration ($L[1]$)	$L1 : \langle 1, 1 \rangle : x_L = 1$	<i>Not supported</i>

Fig. 3 Usage of UCL notations.

Table 1 Comparison of expressiveness with FFL and IDL.

Category	Description	FFL	IDL	UCL
Causal Dependency	Dependency between basic blocks	O	O	O
positive, negative		X	O	O
sequence		X	O	O
OR-join, AND-join		X	O	O
OR-fork, AND-fork		O	O	O
Cardinality	Execution count of a basic block or loop bound	O	O	O
single block or loop		Δ	X	O
dependency between blocks or loops				
Iteration	Iteration number at which a basic block is executed in a loop	O	X	O
Inter-scope constraints	Dependency between basic blocks in different scopes	Δ	Δ	O

O: supported, Δ: partially supported, X: not supported

constraints in the CFG. It also extends the expressiveness of existing languages as summarized in Table 1. FFL cannot support the sequence, OR-join, and AND-join constraints. IDL cannot specify the iteration constraints or the dependency between execution counts of basic blocks. Both FFL and IDL cannot fully support the causal dependencies between basic blocks in different scopes. We will exemplify the necessity of the constraints through a case study using industry software in Sect. 5.

4. Formal Syntax and Semantics of UCL

4.1 Control Flow Automaton and Execution Path

In this paper, we represent a program as a control flow finite automaton whose nodes correspond to the program locations and whose edges are labeled with the basic blocks of the control flow graph.

Definition 1 (Control Flow Automaton) A *Control Flow Automaton (CFA)* is a 5-tuple $G = (Q, B, \delta, q_0, F)$, where

- Q is a finite set of states (program locations),
- B is a finite set of basic blocks,
- $\delta : Q \times B \times Q$ is a transition relation, and $q_i \xrightarrow{b} q_j$ abbreviates $(q_i, b, q_j) \in \delta$,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is a set of final states.

For instance, Fig. 4 (b) depicts a control flow automaton for the example code in Fig. 4 (a). The automaton has a final state q_{11} . Therefore, this CFA accepts a finite sequence of basic blocks $w = B_1 B_2 (B_3 (B_4 + B_5) B_6 B_2)^k B_7 (B_8 + B_9) B_{10}$, where k is less than or equal to the loop bound. We assume that the number of occurrences of a basic block in the sequence w is finite.

Definition 2 (Execution path) An *execution path (or run)*

Fig. 4 (a) An example code. (b) CFA of (a). (c) Scoped CFA of (a).

of CFA $G = (Q, B, \delta, q_0, F)$ is a finite sequence of basic blocks $w = b_0 b_1 b_2 \dots b_n, \forall i \in \mathbb{N}_0 \wedge i \leq n: (q_i, b_i, q_{i+1}) \in \delta$, that starts at q_0 and finishes at a final state $q_{n+1} \in F$. Language of G , designated $L(G)$, is the set of all execution paths of G .

4.2 Constraint Scope

The meaning of a constraint can vary according to its application range. For example, suppose the constraint that ‘ B_1 is executed only once’ in Fig. 4. This constraint is ambiguous in that the constraint is violated if the function $foo()$ is called several times in the whole program; whereas the constraint is satisfied when $foo()$ alone is considered. Therefore, to express the user constraint precisely, UCL supports constraint scope. The type of constraint scope can be structurally defined as a program scope, function scope, or loop scope. Users can also define a scope. UCL generalizes the notion of a scope as a set of basic blocks connected in a CFG. For execution path w , scope β identifies several maximal subpaths of w , where basic blocks only in β appear. The following examples show possible scopes for the code in Fig. 4.

1. Function scope: scope $\{B_1, B_2, B_3, \dots, B_{10}\}$ denotes a function scope from the start to the end of $foo()$.
2. Loop scope (global): scope $\beta = \{B_2, B_3, B_4, B_5, B_6\}$ is a global loop scope from the start to the end of the loop.
3. Loop scope (each iteration): scope $\{B_3, B_4, B_5, B_6\}$ is a loop scope from the start to the end of the loop body. In this case, because B_2 is not included in the scope, the execution escapes from the scope whenever it repeats.
4. User-defined scope: scope $\{B_1\}$ denotes the first basic block executed when $foo()$ is called.

Figure 4(c) shows a scoped CFA of Fig. 4(b), where entry and exit of the scopes are denoted by special symbols, β_B , β_E , and α_E which are described later (Sect. 4.4).

4.3 Syntax of UCL Formula

A UCL constraint can be in one of the two forms, ‘ $\alpha[\varphi]$ ’ and ‘ $\alpha[\beta[\varphi_1] \leadsto \varphi_2]$ ’, as shown in Fig. 5, where α and β are the *constraint* and *cause* scopes, respectively, and φ, φ_1 , and φ_2 are the UCL terms. The symbol ‘ $\leadsto$ ’ represents a

$UCL\ formula ::=$	$\alpha[\varphi]$	Invariant
	$\alpha[\beta[\varphi_1] \leadsto \varphi_2]$	Causality
$UCL\ term\ \varphi ::=$	$b[\otimes n]$	Cardinality & iteration
	sequence	Sequence
	$\varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$	Logical combinations
	$\neg \varphi$	Negation
sequence	$b \mid b \cdot sequence$	

• Symbol b is a basic block and $n \in \mathbb{N}$.
 • Symbols α and β are *constraint* and *cause* scopes for UCL formula such that $\beta \subset \alpha$
 • Cardinality and iteration operators: $\otimes \in \{<, \leq, =, >, \geq, @_h\}$

Fig. 5 Syntax of UCL formula.

causal dependency relation between the cause φ_1 and the effect φ_2 . The constraint scope α determines the application range of a constraint specified; whereas β determines that of the cause within constraint scope α . A UCL term may be a basic block, a sequence of blocks, cardinality of a block, or the logical combination of them. For example, the followings are valid UCL formulas.

1. ‘ $\alpha[a[=2]]$ ’ means that basic block a must be executed exactly twice whenever scope α is entered.
2. ‘ $\alpha[b[@_h3]]$ ’ specifies that basic block b must be executed at the third iteration of the enclosing loop whose header block is h whenever scope α is entered. Note that b may or may not be executed at the other iterations.
3. ‘ $\alpha[\beta[a \vee b] \leadsto c[>3]]$ ’ indicates that basic block c must be executed more than three times in scope α if basic blocks a or b are executed within scope β . Note that β is the subset of α .

For an execution path $w = b_0 b_1 b_2 \dots b_n$ and a scope γ , we denote the **scope subpaths** w^γ is the set of maximal subpaths $b_j b_{j+1} b_{j+2} \dots b_k$ of w such that $j \leq \forall i \leq k \leq n: b_i \in \gamma$. Figure 6 depicts possible subpaths and the range of scopes that are applied to UCL terms φ, φ_1 , and φ_2 . For example, Fig. 6(a) shows the scope subpaths $w^\alpha = \{w_1^\alpha, w_2^\alpha\}$ for scope α , and Fig. 6(b) shows the scope subpaths $w^\alpha = \{w_1^\alpha, w_2^\alpha\}$ and $w^\beta = \{w_1^\beta, w_2^\beta\}$ for scopes α and β .

4.4 Semantics of UCL Formula

We define the semantics of UCL terms and formulas in terms of accepting finite paths using a satisfaction relation $\models$. We write $w \models \varphi$ to denote that an execution path w satisfies the term φ . For a given execution path $w = b_0 b_1 b_2 \dots b_n$, the satisfaction relation $\models$ is defined as follows. In the definition, $pre(w, i)$ and $post(w, i)$ represent the prefix of w of length i and the postfix of w starting from b_i . Let w' be a prefix of w , $pre(w, i)$. Then $w - w'$ is defined to be $post(w, i)$ (i.e., $= b_i b_{i+1} \dots b_n$). And $|w|_b$ is defined to be the number of occurrences of b in w .

Definition 3 (Semantics of UCL) Let $a, b, c \in B$ be basic blocks, and φ, φ_1 , and φ_2 be UCL terms. A finite sequence of basic blocks $w = b_0 b_1 b_2 \dots b_n$ satisfies UCL terms and formulas according to the following rules.

1. $w \models b$ iff $\exists i \in \mathbb{N} \wedge i \leq n: b_i = b$
2. $w \models b[\otimes k]$ iff $|w|_b \otimes k$, where $\otimes \in \{<, \leq, =, >, \geq\}$
3. $w \models b[@_h k]$ iff $\exists i, m, j \in \mathbb{N} \wedge i < m < j < n: |pre(w, i)|_h =$

Fig. 6 Application range of UCL scopes.

$k \wedge (|pre(w, j)|_h = k+1 \vee |post(w, i)|_h = 0) \wedge b_m = b$, where 'h' denotes header of the loop enclosing b

4. $w \models a \cdot c$ iff $\exists i \in \mathbb{N} \wedge i \leq n: pre(w, i) \models a \wedge post(w, i) \models c$
5. $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$
6. $w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$
7. $w \models \neg \varphi$ iff $w \not\models \varphi$
8. $w \models \alpha[\varphi]$ iff $\forall w' \in w^\alpha: w' \models \varphi$
9. $w \models \alpha[\beta[\varphi_1] \leadsto \varphi_2]$ iff $\forall w' \in w^\alpha: w' \models \beta[\varphi_1] \leadsto \varphi_2$
10. $w \models \beta[\varphi_1] \leadsto \varphi_2$ iff $\forall w' \in w^\beta: w' \models \varphi_1$ imply $w - w' \models \varphi_2$

Rules 1 to 4 define the UCL terms, and rules 5 to 7 show that the UCL terms are closed under $\vee$, $\wedge$, and $\neg$. Rule 8 defines the UCL formula $\alpha[\varphi]$. Rules 9 and 10 define the UCL formula $\alpha[\beta[\varphi_1] \leadsto \varphi_2]$.

Definition 4 (Property automaton) A *Property Automaton (PA)* of a property φ is a finite automaton P over basic blocks ($\Sigma = B$) such that $\forall w \in \Sigma^*: w \in L(P)$ iff $w \models \varphi$.

Our aim is to construct the property-preserving CFA by production of the CFA and PA. We first generate PAs for the UCL formulas representing user constraints. Figure 7 depicts how finite automata are constructed from the UCL terms. The automaton for the UCL term 1) only accepts the path w that contains at least one b ($|w|_b \geq 1$), and the automaton for the UCL term 2) only accept the path w that contains exactly n times of b ($|w|_b = n$). The automata for UCL terms $b[> n]$ and $b[< n]$ are generated similarly. The automaton for the UCL term 3) only accepts the path w that contains b after exactly n times of h . The automaton for the UCL term 4) is constructed by concatenating two automata for b and S as shown in [15]. The automaton for the UCL term 5) is constructed through the complementation and intersection of the automata for UCL terms φ_1 and φ_2 as described in [15], [16].

Figure 8 shows the PAs for the two types of UCL formulas, $\alpha[\varphi]$ and $\alpha[\beta[\varphi_1] \leadsto \varphi_2]$. These PAs are constructed based on the automata for UCL terms φ , φ_1 , and φ_2 . To handle scopes α and β , we introduce special transitions labeled with α_B , α_E , β_B and β_E to denote the entry and exit of the scopes. The state q_x denotes a non-accepting

Fig. 7 Finite automata for UCL terms.

state that is entered when the property is not satisfied. The automaton for the UCL formula $\alpha[\varphi]$ in Fig. 8 (a) accepts a sequence of basic blocks w where all subpaths in w^α satisfy the UCL term φ . It also accepts w where all basic blocks are not in scope α . The automaton shown in Fig. 8 (b) accepts w where all subpaths in w^α satisfy φ_1 in scope β and satisfy φ_2 in scope $\alpha - \beta$. It also accepts w where all subpaths in w^α fail to satisfy φ_1 in scope β , or w where all basic blocks are not in scope α . However, the automaton does not accept w , which fails to satisfy the UCL formula.

Definition 5 (Property-preserving CFA) Let $C = (Q_C, B_C, \delta_C, q_C^0, F_C)$ be a CFA and $P = (Q_P, B_P, \delta_P, q_P^0, F_P)$ be a PA of a UCL constraint. The property-preserving CFA (PCFA) $C_P = (Q, B, \delta, q_0, F)$ is the Cartesian product of C and P such that

- $Q : Q_C \times Q_P$,
- $B = B_C = B_P$,
- $\delta : (Q_C \times Q_P) \times B \times (Q_C \times Q_P) = \{((q_C \in Q_C, q_P \in Q_P), b \in B, (q'_C \in Q_C, q'_P \in Q_P)) \mid q_C \xrightarrow{b} q'_C \wedge q_P \xrightarrow{b} q'_P\}$,
- $q_0 = (q_C^0, q_P^0)$,
- $F : F_C \times F_P$.

The PCFA $C_{P_0, P_1, \dots, P_n}$ for a CFA C and multiple PAs $P_0, P_1, \dots, P_n$ is constructed from the Cartesian product of the $C, P_0, P_1, \dots, P_n$.

User-specified UCL constraints may not be consistent with each other or can conflict with the structural or functional flow information of the target program. For example, a user constraint of $'D \leadsto G \wedge H'$ for the CFG shown in Fig. 3 is inconsistent with the structure of the program. The consistency of UCL constraints can be verified through a language emptiness check for the PCFA. If the intersection of the set of reachable states from the initial state and that of the final states of an automaton is an empty set, then the language of the automaton is considered empty. This is well known in the area of model checking [16], so we do not describe it here.

5. Experimental Evaluation

UCL can be accommodated into the existing framework of

Fig. 8 Property automata for UCL formulas.

the static WCET analysis as shown in Fig. 9. The shaded blocks indicate the required parts to integrate the UCL into the static WCET framework. The user specifies UCL constraints in the CFG generated by the static WCET analysis tool through an interactive GUI. Note that each basic block of the CFG is annotated with source codes for users to easily specify the constraints.

The visual UCL constraints are transformed into the UCL formulas. Users can also directly specify their constraints as UCL formulas in a separate text file and load it later. These UCL formulas need to be translated into the proper forms for use in the calculation phase of a static WCET analysis technique. We use the deterministic finite automata to represent both the CFG of the target program and the user-provided UCL constraints. Our tool converts a CFG into a CFA and transforms the UCL formulas into PAs with the user-provided scope information.

The PCFA is constructed through the cross-product of the CFA and PA. The inconsistency of user-provided constraints is fed back to the users by checking the emptiness of the PCFA. The longest execution time path of the PCFA becomes a candidate of the WCET path as all the user constraints are contained in the structure itself. It can be traversed to find the longest path in a path-based method, or can be used to generate the flow facts and solve the integer linear program (ILP) problem in the IPET method.

Figure 10 illustrates how a user-provided UCL specification is processed by our tool. Figure 10(a) shows the CFG of the code in Fig. 4 (a) with a user-specified constraint ' $B_4 \leadsto B_9$ ' to eliminate the infeasible paths that contain both B_4 and B_8 . Figure 10(b) shows the CFA converted from the CFG of Fig. 10(a). Note that the numbers that identify the basic blocks of the CFG become the labels of the CFA. The labels '30' (α_B), '31' (β_B), '32' (β_E), and '33' (α_E) that represent the entry and exit of the scopes are also properly placed in the CFA from the user-designated scopes α and β . The UCL formula ' $\alpha[\beta[11] \leadsto 6)]$ ' is generated from ' $B_4 \leadsto B_9$ ', where $\alpha = \{4, 5, 6, \dots, 13\}$ and $\beta = \{8, 9, 10, \dots, 13\}$. It is then transformed into the property automaton shown in Fig. 10(c) (Note that all self-loops are omitted). It accepts only the paths that satisfy the UCL formula. Figure 10(d) is the PCFA generated through a cross-product of the CFA in Fig. 10(b) and the PA in Fig. 10(c). Automata

intersection, determinization, and minimization utilities are used in the generation of the PCFA.

The WCET is calculated from this PCFA in the same way as it is calculated from a CFG in the usual static WCET methods. Figure 10(e) depicts the WCET path produced by our tool from the PCFA. The calculated WCET is 884 cycles, which is tighter than the WCET (912 cycles) obtained

Fig. 9 Application of UCL to static WCET analysis.

Fig. 10 (a) CFG with user constraint. (b) Scoped CFA. (c) PA (11~6). (d) PCFA. (e) WCET path.

Table 2 The WCET estimation by TimeBouncer2 using FFL and UCL.

No.	Measured WCET	Estimated WCET (no constraints)		Estimated WCET (with FFL constraints)			Estimated WCET (with UCL constraints)		
	(cycles)	(cycles)	(ratio)	(cycles)	(FFL constraints)	(ratio)	(cycles)	(UCL constraints)	(ratio)
1	46,092	2,085,590	4,524.8%	48,771	loop bound, impractical paths (pre-defined data type, exception)	105.8%	48,771	cardinality constraints	105.8%
2	147,336	2,503,500	1,699.2%	211,966	infeasible paths	143.9%	211,230	causal dependency constraints: 1) $F1_N3 \leadsto F1_N13$, 2) $F1_N5 \leadsto F1_N15$, 3) $F1_N7 \leadsto F1_N17$	135.7%
3	839	898	107.1%	898	(no user constraints required)	107.1%	898	(no user constraints required)	107.1%
4	77,148	717,250	929.7%	78,370	loop bound, impractical paths (no exception)	101.6%	78,370	cardinality constraints	101.6%
5	41,460	1,048,560	2,529.1%	66,274	loop bounds, infeasible paths, impractical paths (no exception)	159.9%	65,906	causal dependency constraints: 1) $F1_N3 \leadsto \neg F1_N5 \wedge F1_N10$, 2) $F1_N5 \leadsto \neg F1_N10 \wedge F1_N20$, 3) $F1_N10 \leadsto \neg F1_N20$, cardinality constraint: 4) $F1_L1_N8[= 0]$	145.4%
6	5,401	7,797	144.4%	7,605	infeasible paths	140.8%	7,237	causal dependency constraints: 1) $F1_N3 \leadsto \neg(F1_L1_N7 \vee F1_L1_N15 \vee F1_L1_N21) \wedge F1_L1_N28$, 2) $F1_L1_N26 \leadsto F1_L1_N3$, 3) $(F1_L1_N8 \vee F1_L1_N10) \leadsto F1_L1_N11$	128.1%
7	31,512	37,322	118.4%	32,137	loop bound, infeasible paths	102.0%	31,512	causal dependency constraints: 1) $F1_L1_L1_N4 \leadsto F1_L1_L2_N4$, 2) $(F1_L1_L1_N5 \vee F1_L1_L3_N4) \leadsto F1_N5$ 3) $F1_L1_N7[@1]$	100%

directly from the Fig. 10(a) using the IPET method, since the latter contains infeasible path containing both B_4 and B_8 . Table 2 summarizes the experiment with the data acquisition and command modules of the KOMPSAT-2 flight software to evaluate the expressiveness of UCL and to check the validity of our approach. It shows the estimated WCETs with the measured ones. The measurement was performed by domain engineers through the time-consuming process of running possible paths and measuring the execution time in the electrical testbed of the satellite using the In-Circuit Emulator of an 80386DX target processor.

As shown in the ratio to measurement, the WCETs estimated without user constraints are overestimated because TimeBouncer, a static WCET tool developed at KAIST, could not yet identify the infeasible or impractical paths. For example, the WCET estimation without user constraints in row 1 was overestimated since it did not consider the maximum data rate and types of input data. The WCET estimation with user constraints is tight due to the additional flow information on the possible loop bounds and impractical paths specified in FFL. Even tighter estimations of the WCET were obtained as shown in rows 2, 5, 6, and 7 by specifying the flow information in UCL, which could not be fully specified in FFL. Note that the UCL constraints in the table are not in complete UCL formulas, and the loop bound information presented in the FFL constraints is omitted in the UCL constraints.

The experiment demonstrates that UCL has enough expressiveness to specify complex constraints such as causal dependencies between if-statements in different loops or a

logical combination of causal dependencies which are difficult or impossible to specify in FFL. UCL is also evaluated to be user-friendly since users can specify constraints using visual notations through a graphic user interface instead of specifying them by equations of execution counts. However, the user constraints related to the cardinality constraints, such as the execution count of basic blocks within a scope and simple loop bound, can be expressed and processed more efficiently in FFL since the nodes are repeated or the loop is unfolded at the PCFA in our approach. This problem can be alleviated by turning over the simple cardinality constraints to FFL. We can directly apply the simple cardinality constraints that can be specified in FFL at the calculation phase of the IPET method.

6. Conclusion

The WCET provided by static WCET analysis methods needs to be safe and tight, but it is usually overestimated due to loops and infeasible paths in program codes. The WCET overestimation can also occur in a real-time embedded system that has constraints on input data or hardware dependency that are not analyzable from the code. This implies that additional flow information is a prerequisite for tight WCET estimation by static analysis tools.

In this paper, we proposed a new flow representation language, UCL, to facilitate the specification of user constraints. User constraints specified in UCL formulas are converted into corresponding finite automata. They are combined with the automaton representing the control flow

graph of a target program through cross production. The combined automaton does not contain infeasible or impractical execution paths, which are the main causes of loose estimation of WCET.

Our approach is neutral to back-end calculation methods. The PCFA generated in our method can be applied to the calculation phase of IPET-based or path-based static WCET analysis techniques. A case study with satellite flight software using an IPET-based static WCET analysis tool demonstrated the feasibility and usefulness of UCL and our approach.

For future work, we plan to perform further case studies in various domains to evaluate the usefulness of UCL. We also are studying ways to reduce the large state space caused by the intersection of automata as the number of nodes in the PCFA increases according to the UCL constraints.

Acknowledgments

This research was supported by Foundation of Ubiquitous Computing and Networking (UCN) project, the Ministry of Knowledge Economy (MKE) 21st Century Frontier R&D Program in Korea and a result of subproject UCN 08B3-S2-10M.

References

- [1] A. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Softw. Eng.*, vol.15, no.7, pp.875–889, 1989.
- [2] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol.1, no.2, pp.159–176, Sept. 1989.
- [3] S. Lim, Y.H. Bae, C.T. Jang, B.D. Rhee, S.L. Min, C.Y. Park, H. Shin, K. Park, and C.S. Ki, "An accurate worst-case timing analysis for risc processors," *IEEE Trans. Softw. Eng.*, vol.21, no.7, pp.593–604, 1995.
- [4] F. Stappert and P. Altenbernd, "Complete worst-case execution time analysis of straight-line hard real-time programs," *J. Syst. Archit.*, vol.46, no.4, pp.339–355, 2000.
- [5] J. Engblom and A. Ermedahl, "Modeling complex flows for worst-case execution time analysis," *Proc. 21th IEEE Real-Time Systems Symposium*, 2000.
- [6] R. Kirner and P. Puschner, "Classification of code annotations and discussion of compiler-support for worst-case execution time analysis," *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, 2005.
- [7] AbsInt, aiT Worst-Case Execution Time Analyzers. <http://www.absint.com/ait/>
- [8] TIDORUM, Bound-T execution time analyzer. <http://www.tidorum.fi/bound-t/>
- [9] Y. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *Proc. 32nd ACM/IEEE Design Automation Conference*, 1995.
- [10] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen, "Supporting timing analysis by automatic bounding of loop iterations," *Real-Time Systems*, vol.18, no.2-3, pp.129–156, 2000.
- [11] C. Healy and D. Whalley, "Automatic detection and exploitation of branch constraints for timing analysis," *IEEE Trans. Softw. Eng.*, vol.28, no.8, pp.763–781, 2002.
- [12] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," *Proc. 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pp.57–66, Dec. 2006.
- [13] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel, "Wcet analysis: The annotation language challenge," *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, July 2007.
- [14] C. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems Journal*, vol.5, no.1, pp.31–62, 1993.
- [15] J. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.
- [16] E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

Jong-In Lee received a B.S. degree in computer science from Seoul National University in 1983 and an M.S. degree in computer science from KAIST in 1985. He has participated in the development of KOMPSAT-1&2 flight software at Korea Aerospace Research Institute. He is currently a Ph.D. candidate at KAIST. His research interests include formal method and WCET analysis.

Ho-Jung Bang received a B.S. degree in economics from Seoul National University in 1996. He received an M.S. degree in computer science from KAIST in 2003. He is currently a Ph.D. candidate at KAIST. His research interests include software engineering, formal method, and worst-case execution time analysis.

Tai-Hyo Kim is a CEO of Formal Works, Inc., received B.S., M.S. and Ph.D. degree in computer science from KAIST in 1998, 2000 and 2007. His research interests include formal method, worst-case execution time analysis, and model checking.

Sung-Deok Cha is a professor in the Computer Science and Engineering Department at Korea University. He received a Ph.D. degree in information and computer science at the University of California, Irvine, in 1991. From September 1994 through February 2008, he was a faculty at the Division of Computer Science, KAIST. His research interests include software safety, computer security, and formal method.