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

이종인 (Lee, Jong In)
School of Electrical Engineering and Computer Science
Division of Computer Science
KAIST
2009
Visualization and Formalization of User Constraints for a Tight Estimation of Worst-Case Execution Time
Visualization and Formalization of User Constraints for a Tight Estimation of Worst-Case Execution Time

Advisor: Professor Bae, Doo Hwan

by

Lee, Jong In

School of Electrical Engineering and Computer Science
Division of Computer Science
KAIST

A thesis submitted to the faculty of the KAIST in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the School of Electrical Engineering and Computer Science, Division of Computer Science

Daejeon, Korea
2008. 10. 27.

Approved by

__________________________
Professor Bae, Doo Hwan
Advisor
정확한 최장수행시간 예측을 위한 사용자 제약사항의 시각화 및 정형화

이 종인

위 논문은 한국과학기술원 박사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2008년 10월 27일

심사위원장 배 두환 (인)
심사위원 차성덕 (인)
심사위원 권오준 (인)
심사위원 송준화 (인)
심사위원 이윤준 (인)
Abstract

Timing analysis is an essential process for development of real-time embedded system and knowledge about the worst-case execution time (WCET) is crucial to validation of temporal correctness of implemented system. Research on automated static timing analysis is actively in progress to complement limitation of traditional timing measurement method. Automated static timing analysis methods provide safe but usually overestimated worst-case execution time (WCET). Overestimation is mainly due to the existence of execution paths which turn out to be infeasible or impractical if dynamic behaviors of a program or environmental assumptions are fully considered. Therefore, user annotation of additional flow information is required for static WCET analyzer to get a tighter WCET estimation.

In this thesis, we propose a new method and a visual language, User Constraint Language (UCL), to facilitate the specification of user constraints. In our method, both program and flow information are represented by single formalism—finite automata. UCL provides intuitive visual notations with which users can easily specify various levels of flow information to characterize execution paths of program. 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 reflects the static structure and possible dynamic behavior of the program, and does not contain infeasible or impractical execution paths that are the main cause of loose estimation of WCET.

We describe the visual notation and textual formula of UCL with examples which illustrate their usage. Then we define the formal syntax and semantics of UCL and propose a translation scheme to convert UCL formulas into finite automata. We also present a method to check the consistency of user-provided UCL constraints. UCL can
specify complex flow information to eliminate infeasible or impractical paths that are
difficult or impossible to specify in other flow representation languages. It is also neutral
to the back-end calculation methods so it can be applied to the path-based or IPET-based
static WCET calculation method. A case study using part of a software program for
satellite flight demonstrates the effectiveness of UCL and our approach.
Contents

Abstract .................................................................................................................. i
Contents .................................................................................................................... iii
List of Tables ............................................................................................................ v
List of Figures .......................................................................................................... vi

1 Introduction ......................................................................................................... 1

2 Static WCET Analysis .......................................................................................... 4
   2.1 Overview of Static WCET Analysis ................................................................. 4
   2.2 Program Flow Analysis .................................................................................. 5
   2.3 Low-Level Analysis ....................................................................................... 6
   2.4 Calculation .................................................................................................. 6
       2.4.1 Tree-based Calculation ....................................................................... 6
       2.4.2 Path-based Calculation ..................................................................... 6
       2.4.3 IPET Calculation ............................................................................... 7

3 Flow Information for WCET Estimation .............................................................. 9
   3.1 User Constraints ......................................................................................... 11
       3.1.1 Constraints on Input Data from Environment .................................... 11
       3.1.2 Constraints in Error/Exception Handling .......................................... 11
   3.2 Case Study of User Constraints to Static WCET Analysis ......................... 13
       3.2.1 Case Study 1: Command and Communication Interface Software ...... 14
       3.2.2 Case Study 2: Data Acquisition Software ......................................... 16

4 Flow Information Representation Languages ................................................... 18
   4.1 Information Description Language .............................................................. 19
   4.2 Flow Fact Language .................................................................................... 19
# User Constraint Language

5.1 UCL Notations ................................................................. 22

5.1.1 Causal dependency ....................................................... 23

5.1.2 Cardinality and iteration .............................................. 24

5.2 Expressiveness of UCL .................................................... 25

# Formal Syntax and Semantics of UCL

6.1 Control Flow Automaton and Execution Path ....................... 27

6.2 Constraint scope ............................................................ 28

6.3 Syntax of UCL Formula .................................................... 30

6.4 Semantics of UCL Formula .............................................. 31

6.5 Translation of UCL formulas ............................................ 32

6.6 Consistency check of UCL constraints ................................. 37

# Application of UCL into Static WCET Analysis Framework

7.1 Overview of our approach ................................................ 38

7.2 Prototype tool implementation ......................................... 41

# Experiment

8 .................................................................

# Conclusion and Future Work

9 Conclusion and Future Work ............................................. 48

Summary (in Korean) .......................................................... 49

References ................................................................. 51
List of Tables

5.1 Comparison of expressiveness with FFL and IDL . . . . . . . . . . . . . 26
8.1 The WCET estimation by TimeBounder2 with FFL . . . . . . . . . . . . . 45
8.2 WCET estimation by TimeBounder2 using FFL and UCL . . . . . . . . . 46
List of Figures

2.1 Approximate WCET .................................................. 4
2.2 Static WCET analysis phases ....................................... 5
2.3 Tree-based method ................................................... 7
2.4 Path-based method ................................................... 7
2.5 Structural constraints in IPET method .............................. 8
3.1 Relation between possible execution paths and flow information ........................................... 9
3.2 User constraints in static WCET analysis framework .................. 10
3.3 Example of input data processing task .......................... 12
3.4 Example of exception handling ....................................... 13
3.5 TimeBounder 2.0 ..................................................... 14
3.6 Summary of WCET estimation for CCI codes .................... 16
3.7 Summary of WCET estimation for DAQ codes .................... 17
4.1 Example of Information Description Language .................... 19
4.2 The syntax of IDL .................................................... 20
4.3 Example of Flow Fact Language ..................................... 21
4.4 Specification of Flow Fact Language .............................. 21
5.1 Notations of Use Constraints Language ........................... 23
5.2 Usage of causal dependencies ....................................... 24
5.3 Usage of cardinality and iteration ................................... 25
6.1 (a) Example code (b) CFA (c) Scoped CFA ..................... 28
6.2 Application ranges of UCL scopes ............................... 30
6.3 Syntax of UCL formula .............................................. 31
6.4 Finite automata for UCL terms ..................................... 33
6.5 Property automata for UCL formulas ............................. 34
7.1 Application of UCL to static WCET analysis framework . . . . . . 39
7.2 CFG with user constraint . . . . . . . . . . . . . . . . . . . . . . . 40
7.3 (a) Scoped CFA (b) PA (c) PCFA (d) WCET path . . . . . . . . . 41
7.4 CFG with WCET path . . . . . . . . . . . . . . . . . . . . . . . . . . 42
7.5 Prototype UCL tool . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
1. Introduction

Real-time embedded systems process external inputs, make appropriate decisions, and generate outputs necessary to control the peripherals connected to them. The correctness of system functionality 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. Underestimation of an upper bound can cause an overrun error endangering mission success, while overestimation causes a waste of valuable resources or degraded system performance. Traditional Worst-Case Execution Time (WCET) estimation method roughly estimates execution time by manually counting number of floating point operations or lines of codes, or runs program with proper input data in the target system and measures execution time using measurement tools such as oscilloscope, logic analyzer, or in-circuit emulator. But measurement is impractical for large programs with complex program execution paths because it is hard to execute software on all possible paths.

Research on static analysis is actively in progress to automatically find a tight and safe bound of WCET by analyzing program codes and modeling hardware characteristics of target processor [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Static analysis methods estimate WCET by examining all possible paths and calculating the worst execution time among these paths without executing software. Current methods, however, have a limitation in that they usually overestimate the WCET since they cannot fully identify the infeasible execution paths. Furthermore, static WCET analysis methods do not take the feasible scenario of system operation into consideration. They may overestimate WCET, as they cannot identify the unrealistic or impractical paths when environmental assumptions on input data (e.g., the maximum input data rate, the predefined input data pattern, or input data exception handling) are fully considered. This information cannot be automatically extracted from the program code itself since the impractical paths can be semantically valid. Therefore, it is necessary for user to provide additional flow information to static
WCET analyzer to obtain a tighter WCET estimation. Users can provide a loop bound or information about the infeasible or impractical paths to the static WCET analyzer by annotating program codes or by specifying them in a separate file [12, 13, 14].

However, currently available code annotation languages have simple constructs to bound the execution count of loops or calls to a subprogram, and lack enough expressiveness to specify infeasible or impractical paths of programs with a complicated control flow. Other program flow information representation languages such as the Information Description Language (IDL) [15] and the Flow Fact Language (FFL) [16] have been introduced to specify more complicated flow information. However, their notations are quite demanding for ordinary programmers to understand and use, and they have a limitation in specifying certain complex flow information.

This thesis proposes a new method for tight estimation of WCET and a flow information representation language called User Constraint Language (UCL). Our method uses finite automata to represent the static structure of a program and to specify its possible execution paths. It provides intuitive visual notations for users to easily specify various levels of flow information to characterize the dynamic behavior of a program. User constraints specified in UCL formulas 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. The combined automaton does not contain the execution paths that violate user constraints, from which a tight WCET is calculated. UCL can specify the complex flow information to eliminate infeasible or impractical paths that are difficult or impossible to specify in other flow representation languages. It is also neutral to back-end calculation methods, so it can be applied to the path-based [8] or Implicit Path Enumeration Technique (IPET)-based static WCET calculation method [17]. The consistency of a user-provided UCL specification is checked through the intersection of automata corresponding to UCL formulas.

The organization of this thesis is as follows. Chapter 2 presents an overview of static WCET analysis technique. Chapter 3 identifies the flow information to exclude infeasible or impractical paths and emphasizes their importance to WCET estimation. Chapter 4 surveys flow information representation languages. Chapter 5 describes notations of UCL with examples illustrating their usage followed by a comparison with
other languages. Chapter 6 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. Chapter 7 briefly explains how the proposed approach can be applied to a static WCET analysis framework. Chapter 8 presents the results of an experiment using satellite flight software to demonstrate the effectiveness of our method and UCL to obtain tight WCET estimation. Finally, Chapter 9 gives our conclusion and outlines future plan.
2. Static WCET Analysis

2.1 Overview of Static WCET Analysis

The execution time of a program depends on possible inputs and hardware system states on which the program runs. The worst-case execution time of executing tasks is a key parameter to give timing guarantee for real-time systems. Static Worst-Case Execution Time (WCET) analysis is to provide a priori knowledge about the worst-case execution time of a program without running it \[1\] \[2\] \[3\]. The WCET estimated by static WCET analysis methods should be safe and tight as shown in Fig. 2.1.

![Figure 2.1: Approximate WCET](image)

Static WCET analysis proceeds through the phases of program flow analysis, low-level analysis and calculation as shown in Fig. 2.2. The program flow analysis phase analyzes the code of the program and determines the possible program flows. The low-level analysis phase analyzes the object code and target hardware to determine the timing behavior for instructions running on the target hardware, giving the execution time for each atomic unit of flowbasic block. The calculation phase combines the results of the flow and low-level analyses to calculate a WCET estimate for the program.
Figure 2.2: Static WCET analysis phases

2.2 Program Flow Analysis

The purpose of program flow analysis is to obtain information about which functions are called, how many times loops iterate, if there are dependencies between if-statements, etc [18]. The flow information such as maximum loop iteration, branch constraints, and infeasible paths can be extracted from program code by automatic flow analysis methods or with manual annotations. It can be divided into three sub-phases: flow extraction for actually determining flow of code, flow representation for representing the information obtained in the analysis phase, and flow information conversion for processing the information to be useful for a particular calculation method. The automatic flow analysis is still limited to the well-structured programs which do not use pointers, dynamic data structures, or recursion. In this case, the automatic flow analysis needs to be complemented with manual annotations providing additional flow information such as bound of loop iteration and description of flow dependencies. Further research on automatic flow analysis to get tighter WCET is in progress by detecting loop bounds or infeasible paths automatically instead of the manual annotations which may be error prone [7, 9, 19].
2.3 Low-Level Analysis

The low-level analysis is to determine the execution time of basic blocks given the architectural feature of the target hardware. It determines the timing effect of machine-dependent factors that need to be modeled such as cache, branch predictor and pipeline. As modern processors utilize various performance-optimizing features to enhance performance, modeling of processors’s timing behavior becomes more complex and hard to predict. Research on modeling complex processor’s timing behavior as well as validation of the hardware model is in progress [6, 20, 21, 22, 23, 24].

2.4 Calculation

The calculation phase is to calculate the WCET estimate for the program, given the program flow and low-level analysis results. There are three categories of calculation methods: tree-based, path-based, and IPET-based calculation.

2.4.1 Tree-based Calculation

In tree-based calculation, the WCET estimate is generated by a bottom-up traversal of a program syntax tree [4, 25]. The program syntax tree is a representation of program whose non-leaf nodes correspond to structure of the program (e.g., sequences, loops, conditionals) and whose leaf-nodes represent basic blocks as shown in Fig. 2.3. It shows that the WCET calculated is 754 cycles. This method is simple and computationally cheap, while it has the drawback that the computation is local within a single program statement and cannot consider dependencies between statements.

2.4.2 Path-based Calculation

Path-based calculation is to find the longest execution time path using graph search algorithm after converting the source codes into a control flow graph [8, 25]. This method explicitly represents possible execution paths but has problems with handling flow information of loop-nesting levels. Figure 2.4 shows an example code and its CFG.
extern int k; /* k >= 0 */
foo ()
{
  int ok, j, r;
  ok = FALSE;          // B1
  while (k < 10) {      // B2
    if (ok)   ... ( k < 10)
      j++; B5 j = 0; k = j;
    else {
      B3 if (ok) B4 j++; // B3
      B5 j = 0; k = j;
      B6 k++; // B6
    }
    B7 r = j; // B7
    k++;                     // B6
  }
  r = j;             // B7
}

Figure 2.3: Tree-based method

extern int k; /* k >= 0 */
foo ()
{
  int ok, j, r;
  ok = FALSE;          // B1
  while (k < 10) {      // B2
    if (ok)   ... ( k < 10)
      j++; B5 j = 0; k = j;
    else {
      B3 if (ok) B4 j++; // B3
      B5 j = 0; k = j;
      B6 k++; // B6
    }
    B7 r = j; // B7
    k++;                     // B6
  }
  r = j;             // B7
}

(a) Program code (b) Control flow graph

Figure 2.4: Path-based method

2.4.3 IPET Calculation

The IPET (Implicit Path Enumeration Technique) calculates WCET by solving the objective function which satisfies structural or functional constraints extracted from the program CFG or provided by user \[17\] [16].
\[ WCET = \text{MAX}(\sum_{i}^{N} t_{i} x_{i}) \]

where \( x_{i} \) and \( t_{i} \) are the execution count and time of the basic block \( B_{i} \), respectively.

The result of IPET calculation is a WCET and the worst-case counts for all basic blocks in the CFG. It does not show the worst-case execution path explicitly. The IPET constraints systems can be solved using a constraint-solver or ILP (Integer Linear Programming) technique. IPET-based approach can handle more complex flow information compared to other calculation methods. Comparison with the path-based calculation is performed in [26]. Figure 2.5 illustrates how structural constraints are generated from the program CFG. For example, the constraint that execution count of basic block \( B_{3} \) is equal to that of incoming edge \( d_{3} \) and the sum of outgoing edges \( d_{4} \) and \( d_{5} \) is extracted from the structure of the CFG.

![Figure 2.5: Structural constraints in IPET method](image-url)
3. Flow Information for WCET Estimation

Static WCET analysis is a technique to estimate WCET by analyzing software codes without execution. For this, all possible execution paths have to be determined, and the execution time for each basic block in the path has to be accurately estimated. The set of structurally possible paths regardless of the semantics of the code is infinite if loops in the code can be executed an arbitrary number of times. Bounding all loops with some upper bounds makes it finite as shown in Fig. 3.1 [16, 18]. Adding more flow information, such as dependency between if-statements, allows the set to be narrowed down further to the set of statically feasible paths. This information can be provided by users or automatically obtained through static flow analysis. Research on static WCET analysis to obtain a tight WCET by eliminating infeasible paths and finding loop bounds has been actively performed [7, 9, 19, 27, 28].

Figure 3.1: Relation between possible execution paths and flow information

For example, a tight WCET can be obtained for the code shown in Fig. 2.5(a) by bound-
ing the \textit{while} loop to 10 and constraining the execution of basic block $B_5$ to once at most. This flow information is used to eliminate infeasible path, such as $B_1 - B_2 - B_3 - B_5 - B_6 - B_2 - B_3 - B_5 - B_6 - \cdots - B_2 - B_7$, which causes an overestimation of WCET.

However, there are other types of flow information that are not extractable from the program code and must be provided by the user to obtain a tight WCET estimation. \textit{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. The characteristics of input data from environment may constrain the set of possible execution paths in program, and occurrence of an exception or error in real-time embedded system may cause a special processing for safe operation of system: abortion of operation or discard of input data which imposes restriction on execution paths \cite{29}. As these constraints depend on the target system for which a WCET is estimated, users need to extract them from the system documents such as system specification, equipment specification and interface control document, or from the engineers with domain knowledge. Identified constraints need to be specified as additional flow information applicable to static WCET analysis techniques. Figure 3.2 shows the phases of a static WCET analysis method where user-specified constraints are applied.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure3.2.png}
\caption{User constraints in static WCET analysis framework}
\end{figure}
3.1 User Constraints

The program flow information to eliminate impractical paths should be provided by users. The constraints on input data from environment and the constraints in error/exception handling are examples which restrict the set of possible execution paths. Execution time of an input data processing program in the system may depend on the amount or type of input data from environment. Handling of exception depends on the characteristics of the target system or on the assumption made for the system operation. Users are required to consider the assumption of the environment under which target system operates and specify them in a flow information representation language for static WCET analyzers to get a tighter WCET estimation. These constraints can be extracted from the system documents or system engineers with domain knowledge.

3.1.1 Constraints on Input Data from Environment

Figure 3.3 shows an input data processing task of a hypothetical real-time embedded system. Input data from environment is acquired and stored in the input data buffer for processing by the input data processing task. If the number of input data to be processed by the task \( \text{input data processing} \) is much smaller than the \( \text{MAX} \), which is defined large enough to prevent overflow of the buffer with sufficient margin, or if the input data has always predefined pattern of, for example, \( \text{Type 1} \cdot \text{Type 2} \cdot \text{Type 3} \cdot \text{Type 4} \cdots \text{Type N} \) (\( N < \text{MAX} \)), then a static WCET analyzer will overestimate the WCET since it will take the path to process \( \text{Type 1 MAX} \) times as the WCET path regardless of actual size or pattern of the data in the buffer.

3.1.2 Constraints in Error/Exception Handling

Error checking and logging is important to detect, isolate, and recover from a fault in the real-time embedded system. If an error or exception occurs, the information such as error code, time of occurrence, and description of error, is logged in the error table for diagnosis and required exception handling is performed. A static WCET analyzer usually chooses the path to invoke error handling as WCET path since the execution time to handle error is usually larger than that of normal operation. However, the occurrence
Figure 3.3: Example of input data processing task

of faults all the time is very unlikely and the input data stream in the buffer is usually aborted by an exception handler until the next valid input data sequence starts. If a static WCET analyzer considers the subsequent input data as error without discarding them, it will overestimate the WCET.

Figure 3.4 shows another example of the exception handling of a data acquisition task which gathers hardware sensor data. The data acquisition software, which polls the status of hardware after writing request to it, waits for readiness of data up to the predefined timeout duration. In the most of nominal cases, hardware data is available before the timeout which is conservatively set with a margin. However, static WCET analysis methods, which calculate WCET theoretically, take the path always passing
the *timeout* as the WCET path. They produce an unrealistically overestimated WCET because the probability that all the acquisitions of data will result in timeout is near to zero in practice considering the conservatively defined margin of *timeout* and the fault detection and isolation scheme prepared. (Health of the mission-critical input device is periodically monitored and redundant backup device is available.)

![Diagram of exception handling](image)

Figure 3.4: Example of exception handling

### 3.2 Case Study of User Constraints to Static WCET Analysis

We performed a case study to estimate the WCET of satellite command and communication software and data acquisition software. The WCET estimated by an IPET-based static WCET analysis tool, TimeBounder which was developed in KAIST [30], is compared to the WCET acquired using a traditional dynamic method. Then we show that a tighter WCET could be obtained by specifying user constraints. TimeBounder takes C source code and user defined flow constraints as its input, and returns a WCET with the execution counts of basic blocks. Figure 3.5 shows a screenshot of the tool. Current
3.2.1 Case Study 1: Command and Communication Interface Software

The Command and Communication Interface (CCI) software receives and processes telecommands from ground. Telecommands from ground are in the CLTU (Command Link Transmission Unit) format of the CCSDS (Consultative Committee for Space Data Systems) recommendation. The functions performed by CCI are:

- Process uplinked ground commands,
- Conduct CCSDS frame validation and command validation,
• Execute the uplinked commands

CCI consists of 17 software units and the total C-source line of code is about 1,000 lines long. Figure 3.6 shows a summary of the experiment with CCI codes. Column Measured WCET shows measured WCETs of CCI codes by domain engineers using a target processor simulator (VisualProbe x86 simulator) with test cases identified as the worst case. Columns Ratio-1 and Ratio-2 represent percentage of TimeBounder estimation with and without user constraints to the measurement (TimeBounder estimation / Measurement), respectively. The WCETs obtained using TimeBounder in the row number 1 to 7, 10, 12, 13, and 15 are overestimated compared to the WCET measured by domain engineers. It is largely due to the library functions whose execution times depend on the size of input argument. A library function can be invoked several places in a function or from several functions with different input arguments. As the source codes of run-time library functions (e.g. sprintf, strcpy) in the operating system were not available and TimeBounder did not support object codes, the execution time (cycles) of the library functions had to be provided by user. For the TimeBounder estimation, we chose the maximum execution time of a library function among the whole instances it was called, while an actual input string argument was used in the invocation to the library function for the measured WCET. We also found that the wrong execution paths were chosen for some measured WCETs. For example, they ignored the table overflow case in the exception handler which logs error information in the software error table. The WCETs obtained without user constraints in the row number 8 and 9 are overestimated due to the repeated calls to an exception handler, which is semantically not possible. The maximum input data rate was not considered in the row number 11 and 14 resulting in overestimated WCETs. The large overestimation in the row number 16 and 17 was caused by disregarding the predefined pattern of input data. The WCET obtained with user constraints in the row number 11 was underestimated due to code optimization performed by TimeBounder tool. The code which contains a nested loop was optimized by TimeBounder (One jump instruction shorter compared to the code used for measurement). The WCET estimated using user constraints in the row 8 to 10, 14, 16, and 17 are tighter than those estimated without user constraints.
### WCET Estimated Using Timebounder 1.0

<table>
<thead>
<tr>
<th>No.</th>
<th>Module name</th>
<th>File Name</th>
<th>File Size (SLOC)</th>
<th>Measured WCET (cycles)</th>
<th>WCET Estimated Using Timebounder 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Without User Constraints (cycles)</td>
<td>With User Constraints (cycles)</td>
</tr>
<tr>
<td>1</td>
<td>cci_validate_command_number</td>
<td>CCI_C005.C</td>
<td>19</td>
<td>212</td>
<td>222</td>
</tr>
<tr>
<td>2</td>
<td>cci_validate_memory_space</td>
<td>CCI_C004.C</td>
<td>5</td>
<td>169</td>
<td>171</td>
</tr>
<tr>
<td>3</td>
<td>cci_cmd_frame_err_handler</td>
<td>CCI_C015.C</td>
<td>5</td>
<td>11,486</td>
<td>13,921</td>
</tr>
<tr>
<td>4</td>
<td>cci_reject_command_type</td>
<td>CCI_C011.C</td>
<td>5</td>
<td>9,385</td>
<td>13,922</td>
</tr>
<tr>
<td>5</td>
<td>cci_command_uploads</td>
<td>CCI_C004.C</td>
<td>22</td>
<td>9,551</td>
<td>14,061</td>
</tr>
<tr>
<td>6</td>
<td>cci_memory_upload</td>
<td>CCI_C009.C</td>
<td>52</td>
<td>12,704</td>
<td>14,061</td>
</tr>
<tr>
<td>7</td>
<td>cci_validate_atc_time_tags</td>
<td>CCI_C013.C</td>
<td>30</td>
<td>10,347</td>
<td>16,440</td>
</tr>
<tr>
<td>8</td>
<td>cci_atc_command_types</td>
<td>CCI_C001.C</td>
<td>45</td>
<td>12,988</td>
<td>30,753</td>
</tr>
<tr>
<td>9</td>
<td>cci_code_and_data_uploads</td>
<td>CCI_C002.C</td>
<td>42</td>
<td>12,903</td>
<td>28,278</td>
</tr>
<tr>
<td>10</td>
<td>cci_kpd_load</td>
<td>CCI_C006.C</td>
<td>28</td>
<td>9,835</td>
<td>28,365</td>
</tr>
<tr>
<td>11</td>
<td>cci_manager_cib</td>
<td>CCI_C008.C</td>
<td>159</td>
<td>312,123</td>
<td>591,216</td>
</tr>
<tr>
<td>12</td>
<td>cci_real_time_command</td>
<td>CCI_C010.C</td>
<td>39</td>
<td>12,957</td>
<td>14,359</td>
</tr>
<tr>
<td>13</td>
<td>cci_rxs_load</td>
<td>CCI_C012.C</td>
<td>54</td>
<td>9,952</td>
<td>14,679</td>
</tr>
<tr>
<td>14</td>
<td>cci_command_processing</td>
<td>CCI_C003.C</td>
<td>124</td>
<td>313,339</td>
<td>608,113</td>
</tr>
<tr>
<td>15</td>
<td>cci_main_init</td>
<td>CCI_C007.C</td>
<td>85</td>
<td>1,231</td>
<td>1,284</td>
</tr>
<tr>
<td>16</td>
<td>cci_csds_frm_processing</td>
<td>CCI_C016.C</td>
<td>254</td>
<td>315,340</td>
<td>1,905,220</td>
</tr>
<tr>
<td>17</td>
<td>cci_main</td>
<td>CCI_MAIN.C</td>
<td>11,1</td>
<td>315,430</td>
<td>1,905,340</td>
</tr>
</tbody>
</table>

1. Overestimation by TimeBounder due to library functions whose execution times depend on input argument.
2. Overestimation by TimeBounder (without user constraints) due to repeated calls to exception handler.
3. Overestimation by TimeBounder (without user constraints) due to ignoring maximum input data rate.
4. Overestimation by TimeBounder (without user constraints) due to disregarding a predefined input data pattern.
5. Underestimation by TimeBounder (with user constraints) due to the code optimization of nested loops.

Figure 3.6: Summary of WCET estimation for CCI codes

### 3.2.2 Case Study 2: Data Acquisition Software

The Data Acquisition software (DAQ) acquires telemetry data from the serial, analog, or parallel ports, and formats them according to a predefined telemetry format table for downlink to groundstation. The `daq_read_hardware_data` is invoked periodically to acquire sensor data for use by application software of various satellite subsystems. Acquisition of analog telemetry data takes longer time than those of other types and goes through following steps:

- Outputs request command (channel number, gain, offset),
• Waits for data ready (until timeout).

• If data is ready before timeout, then reads data. If timeout occurs, it returns error status to log the event in the error table. 

Figure 3.7 shows summary of experiment with DAQ codes. The WCETs estimated by TimeBounder in row 1, 2, and 5 are much overestimated since it did not consider the constraints on input data from environment or the constraints on exception handling. For example, the WCET estimated without user constraints in the row number 1 was overestimated since it did not consider the maximum data rate and types of input data. The TimeBounder without user constraints chose the WCET path which processes 64 analog input data all causing timeout errors as shown in Figure 3.4, while there is a constraint that at most twenty input data, which consist of ten analog data and ten other types of input data, are available.

<table>
<thead>
<tr>
<th>No.</th>
<th>Module Name</th>
<th>File Name</th>
<th>File Size (SLOC)</th>
<th>Measured WCET (cycles)</th>
<th>WCET Estimated Using TimeBounder 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Without User Constraints (cycles)</td>
</tr>
<tr>
<td>1</td>
<td>daq_read_hardware_data</td>
<td>daq_rhidw.c</td>
<td>61</td>
<td>46,092</td>
<td>2,085,590</td>
</tr>
<tr>
<td>2</td>
<td>daq_telemetry_processing</td>
<td>daq_tlmp.c</td>
<td>29</td>
<td>147,336</td>
<td>2,503,500</td>
</tr>
<tr>
<td>3</td>
<td>daq_fmt_tlm_header</td>
<td>daq_head.c</td>
<td>9</td>
<td>839</td>
<td>898</td>
</tr>
<tr>
<td>4</td>
<td>daq_read_port_tlm</td>
<td>daq_rtlm.c</td>
<td>72</td>
<td>77,148</td>
<td>717,250</td>
</tr>
<tr>
<td>5</td>
<td>daq_create_minor_frame</td>
<td>daq_cmfr.c</td>
<td>60</td>
<td>41,460</td>
<td>1,048,560</td>
</tr>
<tr>
<td>6</td>
<td>daq_dump_data_to_minor_frame</td>
<td>daq_mfdp.c</td>
<td>53</td>
<td>5,401</td>
<td>7,797</td>
</tr>
</tbody>
</table>

Figure 3.7: Summary of WCET estimation for DAQ codes
4. Flow Information Representation Languages

Identified constraints need to be specified as additional flow information applicable to a static WCET analysis technique as shown in Fig. 3.2. Users can provide the information for a static WCET analyzer by annotating program codes or by specifying them in a separate file [12]. Code annotation language can be implemented as an extension to programming language syntax, or as simple compiler directives by modifying a compiler. Or it can be an independent flow information language which can be processed by static WCET analysis tools [13, 14]. Currently available code annotation languages have simple constructs to bound the execution count of loops or calls to subprograms, and lack enough expressiveness to specify the flow information for infeasible or impractical paths of programs with complicated control flow. The desired properties of a flow representation language to facilitate the specification of flow information by users are:

- Intuitiveness
  Visual notation is much more intuitive for users to specify the required flow information than the textual notation such as source code annotation or formulas of execution counts of basic blocks.

- Expressiveness
  It should be expressive enough to specify the static and dynamic flow information to exclude infeasible or impractical paths for a tight estimation of WCET.

- Easy to analyze
  The constraints specified in a flow representation language should be automatically analyzed by a static WCET analysis tool.

- Neutral to back-end calculation methods
  The language should not be closely connected to a particular back-end calculation method of static WCET analysis.
4.1 Information Description Language

Park [15] introduced a flow information representation language called Information Description Language (IDL). Both program path and user information are represented as a regular expression of statement labels in IDL. For example, the program path in Fig. 2.4 is represented as $B_1 B_2 (B_3 (B_4 + B_5) B_6 B_2)^* B_7$ and a user information that the while loop is executed 10 times is represented as $\_ (B_3 \_)^{10}$. Note that the wild card symbol ‘?’ means any string of statements not containing adjacent statements. Figure 4.1 shows an example of IDL specification. The constraint that bound of while loop $L$ is 10 can be specified in IDL as ‘loop$L$[1,10]times’. The constraint that statement $A$ cannot be executed more than once can be specified in IDL as ‘execute$A$[0,1]timesinside$L$’. The syntax of IDL is shown in Fig. 4.2. IDL has a limitation in specifying the constraint related to the iteration of loops or causal dependency between them.

![Figure 4.1: Example of Information Description Language](image)

4.2 Flow Fact Language

Engblom and Ermedahl [16] 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 execution of basic block $B$ should be followed by
the execution of basic block $E$ in Fig. 4.3 is specified in FFL as ‘$\text{scope} : [] : x_B \leq x_E$’. The constraint that the bound of loop $L_1$ is 3 can be specified as ‘$\text{scope} : [] : x_F \leq 4$’.

A context specifier ‘[ ]’ means that the fact is considered as a sum over all iterations of the defining scope. Figure 4.4 shows summary of flow fact language specification. 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. For example, if we have a constraint in Fig. 3.3 that $Type-3$ comes only after $Type-1$ and the $Type-2$ data come in sequence, it cannot be specified in FFL. It also has a limitation in specifying a combined constraints 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.
if (cond)                         //A
  x = true;                     //B
else
  x = false;                    //C
if (x)                        //D
  z = 0;                        //E
while (i < 3) {
  if (x) {
    y = k++;           //H
    x = false;         //I
  } else
    y = k--;                   //I
  i++;               //J
}                    //J
z = z + k;               //K

Figure 4.3: Example of Flow Fact Language

Fact Æ Scope : ContextSpec : Constraint
ContextSpec Æ < >
|    [ ]
|    <RangeList>
|    [RangeList]
RangeList Æ ...   
all
all
range
range
foreach
total
foreach
total
< >
[ ]
< Ranges …>
[Ranges…]
IterationsTypeOperator

Operator    Type     Iterations
< >           foreach  all
[ ]           total    all
< Ranges …>   foreach  range
[Ranges…]     total    range

Figure 4.4: Specification of Flow Fact Language
5. User Constraint Language

UCL is a formal and graphical language to specify user constraints for tight estimation of WCET [31]. 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. We introduce notations of UCL with examples illustrating their usage and compare them with those of FFL and IDL to demonstrate their expressiveness. The usefulness of the notations will be shown later with a real-world example.

5.1 UCL Notations

UCL specifies user constraints on CFG using graphic notations as shown in Fig. 5.1. A UCL constraint begins with a start-of-constraint mark and ends with an end-of-constraint mark. 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. Sequence notation is used to specify execution sequence of trigger or triggered basic blocks. 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 user can also define a scope.

For explanation purpose, we also define several textual notations: causal relation using ‘→’, execution sequence of basic blocks by ‘•’ and negation by ‘¬’. 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 block is $h$.

### 5.1.1 Causal dependency

Figure 5.2 illustrates UCL notations to specify the causal dependency between basic blocks in CFG, followed by comparison with corresponding specifications in FFL and IDL. Figure 5.2(a) shows two causal dependencies that the execution of $C$ triggers the execution of $F$ and the execution of $D$ should not be followed by the execution of $G$. Figure 5.2(b) illustrates two OR causal dependencies that if $C$ or $G$ is executed then $I$ should be executed later, and when $D$ is executed then $F$ or $J$ should be executed later. Figure 5.2(c) shows two AND causal dependency that when both $C$ and $G$ are executed then $I$ should be executed later, and when $D$ is executed then both $F$ and $J$ should be executed later. Figure 5.2(d) specifies a sequence constraint that when two basic blocks $D$ and $E$ are executed in sequence, then $F$ should be executed later. Note that the execution of $D$ may not be followed immediately by the execution of $E$.

As shown in the table below diagram, FFL cannot express the causal dependency `$C \lor G \leadsto I$’ as ‘$\text{scope} : <>: x_C + x_G \leq x_I$’ since both $C$ and $G$ can be executed more than once. The causal dependency `$C \land G \leadsto I$’ cannot be specified as ‘$\text{scope} : <>: x_C \times x_G \leq x_I$’ because the execution counts of $C$ and $G$ may be greater than one. FFL cannot specify the sequence dependency `$D \cdot E \leadsto F$’ either.
Figure 5.2: Usage of causal dependencies

5.1.2 Cardinality and iteration

Figure 5.3 shows the causal dependencies related to the execution counts of basic blocks. It illustrates six cardinality constraints and one iteration constraint: 1) $B$ is executed unconditionally, 2) $H$ is never executed when the loop scope $L1$ is entered, 3) bound of loop $L1$ is $m$, 4) $S$ must be executed exactly $m$-times when the scope $L2$ is entered, 5) $N$ must be executed after $G$ is executed more than three times, 6) $F$ must be executed at the first iteration of loop $L1$, and 7) the execution of $K$ exactly $n$-times must be followed by the execution of $R$ exactly $n$-times.

Both FFL and IDL have difficulties in specifying the fifth and seventh constraints. IDL cannot specify the fourth constraint. Note that the FFL context specifier $<>$ means ‘for each’ iteration within a scope.
5.2 Expressiveness of UCL

UCL is intuitive and user-friendly compared to source code annotation as it uses visual notations to specify user constraints in the CFG. It also extends the expressiveness of existing languages as summarized in Table 5.1. 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 Chapter 8.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>FFL</th>
<th>IDL</th>
<th>UCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causal Dependency</td>
<td>Dependency between basic blocks</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>positive</td>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>negative</td>
<td></td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>sequence</td>
<td></td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>OR-join</td>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>OR-fork</td>
<td></td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>AND-join</td>
<td></td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>AND-fork</td>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>Cardinality</td>
<td>Execution count of a basic block or loop bound</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>single block or loop</td>
<td></td>
<td>△</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>dependency between</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocks or loops</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration</td>
<td>Iteration number at which a basic block is executed in a loop</td>
<td>O</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>Inter-scope constraints</td>
<td>Dependency between basic blocks in different scopes</td>
<td>△</td>
<td>△</td>
<td>O</td>
</tr>
</tbody>
</table>

O: supported, △: partially supported, X: not supported
6. Formal Syntax and Semantics of UCL

The constraints specified by users using visual notations of UCL are converted to textual notations - UCL formulas. We describe the formal syntax and semantics of the UCL formulas and a translation scheme to convert them into corresponding automata. A method to check the consistency of user-provided UCL specifications is also proposed.

6.1 Control Flow Automaton and Execution Path

In this thesis, 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 CFG.

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

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

For instance, Fig. 6.1(b) depicts a control flow automaton for the example code in Fig. 6.1(a). The automaton has a final state $q_{11}$. Therefore, this CFA accepts a finite sequence of basic blocks $w = B_1B_2(B_3(B_4 + B_5)B_6B_7)^kB_8(B_9 + B_{10})B_{10}$, 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) of CFA $G = (Q, \Sigma, \delta, q_0, F)$ is a finite sequence of basic blocks $w = b_0b_1b_2\cdots b_n \forall i \in \mathbb{N}_0 \wedge i \leq n : (q_i, b_i, q_{i+1}) \in \delta$, as abbreviated to $q_i \xrightarrow{b_i} q_{i+1}$, 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 $G$.

### 6.2 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. 6.1. 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 \( \beta \) identifies several maximal subpaths of \( w \), where basic blocks only in \( \beta \) appear. The following examples show possible scopes for the code in Fig. 6.1.

1. Function scope: scope \( \{B_1, B_2, B_3, \cdots B_{10}\} \) denotes a function scope from the start to the end of \( \text{foo()} \).

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 \( \text{foo()} \) is called.

Figure 6.1(c) shows a scoped CFA of Fig. 6.1(b), where entry and exit of the scopes are denoted by special symbols, \( \beta_B, \beta_E, \) and \( \alpha_E \) which are described later (Section 6.5).

The following shows our formal definition of scope. In the definition, we use \( w^\beta \) to denote the set of subpaths that a scope \( \beta \) identifies.

**Definition 3 (Scope)** A scope \( \beta \in 2^\Sigma \) is a set of basic blocks which are connected. For an execution path \( w = b_0 b_1 b_2 \cdots b_n \) and a scope \( \beta \), **scope subpaths** \( w^\beta \) is the set of maximal subpaths \( b_j b_{j+1} b_{j+2} \cdots b_k \) of \( w \) such that \( j \leq \forall i \leq k \leq n : b_i \in \beta \).

For example, Fig. 6.2(a) shows the scope subpaths \( w^\alpha = \{w_1^\alpha, w_2^\alpha\} \) for scope \( \alpha \), and Fig. 6.2(b) shows the scope subpaths \( w^\alpha = \{w_1^\alpha, w_2^\alpha\} \) and \( w^\beta = \{w_1^\beta, w_2^\beta\} \) for scopes \( \alpha \) and \( \beta \).
6.3 Syntax of UCL Formula

A UCL constraint can be in one of the two forms, ‘α[φ]’ and ‘α[β[φ₁] \(\rightarrow\) φ₂]’, as shown in Fig. 6.3 where α and β are the constraint and cause scopes, respectively, and φ, φ₁, and φ₂ are the UCL terms. The symbol ‘\(\rightarrow\)’ represents a causal dependency relation between the cause φ₁ and the effect φ₂. 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. ‘α[a[= 2]]’ means that basic block a must be executed exactly twice whenever scope α is entered.

2. ‘α[b[@h,3]]’ 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. ‘α[β[a \lor b] \(\rightarrow\) 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.
6.4 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 $\varphi$. For a given execution path $w = b_0 b_1 b_2 \cdots b_n$, the satisfaction relation $\models$ is defined as follows. In the definition, $\text{pre}(w, i)$ and $\text{post}(w, i)$ represent the prefix of $w$ of length $i$ and the postfix of $w$ starting from $b_i$. Let $w'$ be a prefix of $w$, $\text{pre}(w, i)$. Then $w - w'$ is defined to be $\text{post}(w, i)$ (i.e., $= b_i b_{i+1} \cdots b_n$). And $|w|_b$ is defined to be the number of occurrences of $b$ in $w$.

**Definition 4 (Semantics of UCL)** Let $a, b, c \in \Sigma$ be basic blocks, and $\varphi, \varphi_1$, and $\varphi_2$ be UCL terms. A finite sequence of basic blocks $w = b_0 b_1 b_2 \cdots b_n$ satisfies UCL terms and formulas according to the following rules.

1. $w \models b$ iff $\exists i \in \mathbb{N} \land i \leq n : b_i = b$

2. $w \models \neg b$ iff $w \not\models b$

3. $w \models b[\diamond k]$ iff $|w|_b \diamond k$, where $\diamond \in \{<, \leq, =, >, \geq\}$

4. $w \models b[@hk]$ iff $\exists i, m, j \in \mathbb{N} \land i < m < j < n : |\text{pre}(w, i)|_h = k \land (|\text{pre}(w, j)|_h = k+1 \lor |\text{post}(w, i)|_h = 0) \land b_m = b$, where ‘$h$’ denotes header of the loop enclosing $b$
5. $w |\models a \cdot c$ iff $\exists i \in \mathbb{N} \land i \leq n: \text{pre}(w, i) |\models a \land \text{post}(w, i) |\models c$

6. $w |\models \varphi_1 \lor \varphi_2$ iff $w |\models \varphi_1$ or $w |\models \varphi_2$

7. $w |\models \varphi_1 \land \varphi_2$ iff $w |\models \varphi_1$ and $w |\models \varphi_2$

8. $w |\models \neg \varphi$ iff $w |\not\models \varphi$

9. $w |\models \alpha[\varphi]$ iff $\forall w' \in \omega^* : w' |\models \varphi$

10. $w |\models \alpha[\beta[\varphi_1] \rightarrow \varphi_2]$ iff $\forall w' \in \omega^* : w' |\models \beta[\varphi_1] \rightarrow \varphi_2$

11. $w |\models \beta[\varphi_1] \rightarrow \varphi_2$ iff $\forall w' \in \omega^* : w' |\models \varphi_1$ imply $w - w' |\models \varphi_2$

Rules 1 to 5 define the UCL terms, and rules 6 to 8 show that the UCL terms are closed under $\lor$, $\land$, and $\neg$. Rule 9 defines the UCL formula $'\alpha[\varphi]'$. Rules 10 and 11 define the UCL formula $'\alpha[\beta[\varphi_1] \rightarrow \varphi_2]'$.

**Definition 5 (Property automaton)** A Property Automaton (PA) of a property $\varphi$ is a finite automaton $P$ over basic blocks ($\Sigma = B$) such that $\forall w \in \Sigma^* : w \in L(P) \iff w |\models \varphi$.

### 6.5 Translation of UCL formulas

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 6.4 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 accepts the path $w$ that does not contain $b$ ($|w|_b = 0$). The automaton for the UCL term 3) only accepts the path $w$ that contains exactly $n$ times of $b$ ($|w|_b = n$). The automata for UCL terms 4) and 5) are generated similarly. The automaton for the UCL term 6) only accepts the path $w$ that contains $b$ after exactly $n$ times of $h$. The automaton for the UCL term 7) is constructed by concatenating two automata for $b$ and $S$ as shown in [32]. The automaton for the UCL
term 8) is constructed through the complementation and intersection of the automata for UCL terms \( \varphi_1 \) and \( \varphi_2 \) as described in [32], [33].

Figure 6.4 shows the PAs for the two types of UCL formulas, ‘\( \alpha[\varphi] \)’ and ‘\( \alpha[\beta[\varphi_1]] \)’.
φ2]. These PAs are constructed based on the automata for UCL terms φ, φ1, and φ2. To handle scopes α and β, we introduce special transitions labeled with αB, αE, βB and βE to denote the entry and exit of the scopes. The state qs denotes a non-accepting state that is entered when the property is not satisfied. The automaton for the UCL formula ‘α[φ]’ in Fig. 6.5(a) accepts a sequence of basic blocks w where all subpaths in wα satisfy the UCL term φ. It also accepts w where all basic blocks are not in scope α. The automaton shown in Fig. 6.5(b) accepts w where all subpaths in wβ satisfy φ1 in scope β and satisfy φ2 in scope α − β. It also accepts w where all subpaths in wα fail to satisfy φ1 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.

![Property automata for UCL formulas](attachment:image.png)

**Theorem 1 (Conversion of α[φ] to PA)** Let A = (QA, ΣA, δA, q0A, F_A) be the PA of property φ over basic blocks. (ΣA = the set of basic blocks of the corresponding CFG).
Then, the PA $P = (Q, \Sigma, \alpha, q_0, F)$ of property $\alpha[\varphi]$ is constructed as follows:

- $Q = \{q_0\} \cup \Sigma \cup \{q_X\}$.
- $\Sigma = \Sigma \cup \{\alpha_B, \alpha_E\}$.
- $F = \{q_0\} \cup F_A$.
- $\delta : Q \times \Sigma \times Q = \delta_A \cup \{(q_0, \alpha_B, q_0^A)\} \cup \{(q_0, \sigma', q_0) | \forall \sigma' \in \Sigma - \{\alpha_B\}\} \cup \{(q_0', \alpha_E, q_X) | \forall q_0' \in Q_A - F_A\} \cup \{(q_0''', \sigma'', q_0''') | \forall q_0''' \in F_A \land \forall \sigma'' \in \Sigma - \{\alpha_E\}\} \cup \{(q_0''', \alpha_E, q_0) | \forall q_0''' \in F_A\}$.

where $\alpha_B$ and $\alpha_E$ are virtual symbols indicating the entry and exit of scope $\alpha$, respectively.

**Proof sketch:** We will show that the set of execution path of the CFA constrained by formula $\alpha[\varphi]$ is equivalent to the set of words accepted by the corresponding PA $P$, i.e., $w \models \alpha[\varphi] \Leftrightarrow w \in L(P)$.

($\Rightarrow$) We know that PA $A$ for the UCL term $\varphi$, constructed as shown in Fig. 6.4, accepts all $w \models \varphi$. Let $\hat{\delta} : Q \times \Sigma \times Q$ be a new transition relation such that $(q, w, q') \in \hat{\delta}$ if there is a path in $P$ from $q$ to $q'$, labeled by $w$. Note that $(q, \varepsilon, q) \in \hat{\delta}$, and $(q, wa, q') \in \hat{\delta}$ if $(q, w, r) \in \hat{\delta}$ and $(r, a, q') \in \hat{\delta}$ for $r \in Q$. Let $w = w_1 w_2^a w_2^3 \cdots w_k^a$, where $w_i$ is a sequence of basic blocks which do not belong to $\alpha$, $w_i^A \in L(A)$ and $w_i^A \models \varphi \forall i : 1 \leq i \leq k$. By definition of $w \models \alpha[\varphi]$, it is clear that $(q_0, w, q') \in \hat{\delta}$ for a $q' \in F_A \cup \{q_0\}$ since $w_i^A \in L(A)$. If $w$ does not contain the basic blocks in scope $\alpha$, then $(q_0, w, q_0) \in \hat{\delta}$. So $w \in L(P)$.

($\Leftarrow$) Let $w = w_1 w_2$, where $w_1$ and $w_2$ are the sequences of basic blocks which do not belong to $\alpha$, and $w'$ consists of the basic blocks which belong to $\alpha$. If we assume $w' \notin L(A)$ then $(q_0, w, q_X) \in \hat{\delta}$ since $(q_0, w_1, q_0) \in \hat{\delta}$, $(q_0, w', q') \in \hat{\delta}$ for a $q' \in Q_A - F_A$, and $(q', w_2, q_X) \in \hat{\delta}$, which contradicts to $w \in L(P)$. Therefore $w \models \alpha[\varphi]$ as $w' \in L(A)$ and hence $w' \models \varphi$.

**Theorem 2 (Conversion of $\alpha[\beta[\varphi_1] \leadsto \varphi_2]$ to PA)** Let $A = (Q_A, \Sigma_A, \delta_A, q_0^A, F_A)$ and $B = (Q_B, \Sigma_B, \delta_B, q_0^B, F_B)$ be the PAs of properties $\varphi_1$ and $\varphi_2$ over the basic blocks, respectively ($\Sigma_A = \Sigma_B = \{\text{the set of basic blocks of the corresponding CFG}\}$). Then, the PA $P = (Q, \Sigma, \alpha, q_0, F)$ of property $\alpha[\beta[\varphi_1] \leadsto \varphi_2]$ is constructed as follows:
\[ Q = \{q_0\} \cup Q_A \cup Q_B \cup \{q_x\}. \]

\[ \Sigma = \Sigma_A \cup \{\alpha_E, \beta_B, \beta_E\}. \]

\[ F = \{q_0\} \cup F_B. \]

\[ \delta : Q \times \Sigma \times Q = \delta_A \cup \delta_B \cup \{(q_0, \beta_B, q_0^0)\} \cup \{(q_0, \sigma', q_0) \mid \forall \sigma' \in \Sigma - \{\beta_B\}\} \cup \{(q_0^0, \beta_E, q_0) \mid \forall q_0^0 \in Q_A - F_A \} \cup \{(q_0^0, \beta_E, q_0) \mid \forall q_0^0 \in F_A \} \cup \{(q_b^0, \alpha_E, q_X) \mid \forall q_b^0 \in Q_B - F_B \} \cup \{(q_b^0, \sigma', q_b^0) \mid \forall q_b^0 \in F_B \land \forall \sigma' \in \Sigma - \{\alpha_E\}\} \cup \{(q_b^0, \alpha_E, q_0) \mid \forall q_b^0 \in F_B\}, \]

where \( \beta_B \) and \( \beta_E \) are virtual symbols indicating the entry and exit of the scope \( \beta \), and \( \alpha_E \) is the exit of the scope \( \alpha \).

**Proof sketch:** We will show that the set of execution path of a CFA constrained by the formula \( \alpha[\beta[\varphi_1] \leadsto \varphi_2] \) is equivalent to the set of words accepted by the corresponding PA \( P \), i.e., \( w \models \alpha[\beta[\varphi_1] \leadsto \varphi_2] \iff w \in L(P) \).

(\( \Rightarrow \)) Let \( w = w^1_{a} w_{a}^y \) as shown in Fig. 6.2(b), where \( w^1_{a} \) and \( w_{a}^y \) are the subpaths consisting of basic blocks not in scope \( \alpha \). We introduce virtual symbols indicating the entry and exit of scopes to have \( w = w^1_{a} \beta_B \mu_{a}^\beta \beta_E w_{a}^y \alpha_{E} w_{a}^y \). Our aim is to show that \( w \in L(P) \). We know that \( w^1_{a} \in L(A) \) and \( w_{a}^y \in L(B) \) from the Rule 11 of Definition 4 and Theorem 1. \( (q_0, w, q_f) \in \hat{\delta} \) for a \( q_f \in F \) since \( (q_0, w^1_{a}, q_0) \in \hat{\delta}, (q_0, \beta_B, q_0^0) \in \hat{\delta}, (q_0^0, w^y_{a}, q_0^0) \in \hat{\delta} \). \( (q_0^0, w^y_{a}, q_0^0) \in \hat{\delta} \) for a \( q_{a} \in F_A \). \( (q_{a} \beta_E, q_0^0) \in \hat{\delta}, (q_0^0, w^y_{a}, q_{a} \beta_E) \in \hat{\delta} \) for a \( q_{a} \in F_B \). Similarly, we can show that \( w \in L(P) \) for \( w = w^1_{a} w_{a}^y \). \( \Leftarrow \) Let \( w = w^1_{a} \beta_B w_{a}^y \) where \( w^1_{a} \) and \( w_{a}^y \) are the subpaths consisting of basic blocks not in the scope \( \alpha \), and \( w^1_{a} \in L(A), w_{a}^y \in L(B) \). Since \( w^1_{a} \models \varphi_1 \) and \( w_{a}^y \models \varphi_2 \) by Theorem 1 and the meaning of virtual symbols, we have \( w \models \alpha[\beta[\varphi_1] \leadsto \varphi_2] \). Let \( w = w^1_{a} \beta_B \mu_{a}^\beta \beta_E w_{a}^y \), where \( w^1_{a} \in L(A) \) and \( w_{a}^y \in L(B) \). Then \( w \in L(P) \), since \( (q_0, w^1_{a}, q_0) \in \hat{\delta} \), \( (q_0, \beta_B, q_0^0) \in \hat{\delta}, (q_0^0, w^y_{a}, q_0^0) \in \hat{\delta} \) for some \( q_{a} \in Q_A - F_A \), \( (q_{a} \beta_E, q_0^0) \in \hat{\delta} \), and \( (q_0, \beta_B, q_0^0) \in \hat{\delta} \). Since \( w^1_{a} \models \varphi_1 \) and \( \beta_B \mu_{a}^\beta \beta_E \models \beta[\varphi_1] \leadsto \varphi_2 \), we have \( w \models \alpha[\beta[\varphi_1] \leadsto \varphi_2] \) by the Rules 10 and 11 of Definition 4.

We now combine the CFA with PAs to construct the property-preserving CFA from which a WCET is calculated using the path-based or IPET-based calculation method.
Definition 6 (Property-preserving CFA) Let $C = (Q_C, \Sigma_C, \delta_C, q^0_C, F_C)$ be a CFA and $P = (Q_P, \Sigma_P, \delta_P, q^0_P, F_P)$ be a PA of a UCL constraint. The property-preserving CFA (PCFA) $C_P = (Q, \Sigma, \delta, q^0, F)$ is the Cartesian product of $C$ and $P$ such that

- $Q : Q_C \times Q_P$,
- $\Sigma = \Sigma_C = \Sigma_P$,
- $\delta : (Q_C \times Q_P) \times \Sigma \times (Q_C \times Q_P) = \{((q_C \in Q_C, q_P \in Q_P), \sigma \in \Sigma, (q'_C \in Q_C, q'_P \in Q_P)) \mid q_C \xrightarrow{\sigma} q'_C \land q_P \xrightarrow{\sigma} q'_P\}$,
- $q^0 = (q^0_C, q^0_P)$,
- $F : F_C \times F_P$.

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

6.6 Consistency check of UCL constraints

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 ‘$C \lor D \sim F \land G$’ for the CFG shown in Fig. 5.2(a) 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 [33, 34], so we do not describe it here.

Definition 7 (Consistency of UCL constraints) Let $C$ be a CFA and $P_0, P_1, \ldots, P_n$ be a set of the property automata for UCL constraints. The UCL constraints are consistent if and only if the language of the property-preserving CFA $C_{P_0, P_1, \ldots, P_n}$, $L(C_{P_0, P_1, \ldots, P_n}) \neq \phi$; otherwise, they are inconsistent.
7. Application of UCL into Static WCET Analysis Framework

7.1 Overview of our approach

UCL can be accommodated into the existing framework of the static WCET analysis as shown in Fig. 7.1. 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 the user to easily specify the constraints. The visual UCL constraints are transformed into the UCL formulas by our tool for subsequent processing. 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 7.2 to Fig. 7.4 illustrate how a user-provided UCL specification is processed by our tool. Figure 7.2 shows the CFG of the code in Fig. 6.1(a) with a user-specified constraint ‘B₄→B₉’ to eliminate the infeasible paths that contain both B₄ and B₈. Figure 7.3(a) shows the CFA converted from the CFG of Fig. 7.2. Note that the numbers...
that identify the basic blocks of the CFG become the labels of the CFA. The labels ‘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 (Note that α_B is the same as β_B). The UCL formula ‘α[β[11]~6]’ is generated from ‘B4~B9’, where α = {4, 5, 6, · · · , 13} and β = {8, 9, 10, · · · , 13}. It is then transformed into the property automaton shown in Fig. 7.3(b) (Note that all self-loops are omitted). It accepts only the paths that satisfy the UCL formula. Figure 7.3(c) is the PCFA generated through a cross-product of the CFA in Fig. 7.3(a) and the PA in Fig. 7.3(b). Automata intersection, determinization, and minimization utilities are used in the generation of the PCFA. The WCET is calculated from this PCFA with the execution time for each label in the same way as it is calculated from a CFG in the usual static WCET methods. The loop bound constraint for the original CFG (‘F1.L1.NHeader ≤ 11 × F1.N1’) should be correspondingly reflected in the calculation of the WCET from the PCFA (‘F1.L1.NHeader_0 + F1.L1.NHeader_1 ≤ 11 × F1.N1’). Figure 7.3(d) depicts the WCET path produced by our tool from the PCFA using the IPET technique. The worst-case execution time path obtained by the IPET calculation can be disconnected due to loops in the PCFA. Puschner and Schedl [25] proposed a method to prevent the disconnected WCET path. The two constraints ‘F1.L1.NHeader_1 ≤ 11 × F1.N1’ and
‘F1.L1._NHeader.0 ≤ 11 × F1.L1._N3.0’ were required for the two loops in the PCFA. Note that the automata shown in Fig. 7.3 are only for the internal use inside our tool and are not presented to the user. The edges in the WCET path are automatically mapped back into the original CFG for presentation to users as shown in Fig. 7.4. The calculated WCET is 884 cycles, which is tighter than the WCET (912 cycles) obtained directly from the Fig. 7.2 using the IPET method, since the latter was calculated from the infeasible path containing both ‘F1.L1._N3’ (B4) and ‘F1._N4’ (B8).

Figure 7.2: CFG with user constraint
7.2 Prototype tool implementation

We implemented UCL into an IPET-based static WCET analysis tool called TimeBounder. TimeBounder takes a C source code with user-defined flow constraints specified in FFL as input, and outputs the calculated WCET with the execution counts of all basic blocks. Figure 7.3 shows a screenshot of the tool. The current version of TimeBounder supports only an Intel 80386 processor and runs on an MS-Windows platform. We put em-
phasis on the minimum changes in the existing tool when implementing UCL. Modules
to convert CFG into CFA with user-provided scopes and to transform user constraints
in UCL formula into PA were added to the existing tool. Automata intersection, de
terminization, and minimization utilities to generate the PCFA and a function to check the consistency of UCL formulas were also implemented. An interactive GUI module to provide users with the menu of UCL notations for specifying user constraints in the CFG and a module to convert them into the UCL formulas are under construction. Figure 7.5 shows a snapshot of prototype tool which calculates the WCET from PCFA with user-provided loop bound information.

Figure 7.5: Prototype UCL tool
8. Experiment

We performed an experiment with a part of satellite flight software to evaluate the expressiveness of UCL and to check the validity of the proposed approach. Table 8.1 summarizes the experiment with the data acquisition and command modules of the KOMPSAT-2 flight software. It shows the estimated WCETs with the measured WCET. 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 TimeBounder could not 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 using user constraints specified in FFL is tight due to the additional flow information on the possible loop bounds and impractical paths specified in FFL. In Table 8.2, even tighter estimations of the WCET were obtained as 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 is omitted.

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 the user 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 di-
rectly apply the simple cardinality constraints that can be specified in FFL at the calculation phase of the IPET method.

Table 8.1: The WCET estimation by TimeBounder2 with FFL

<table>
<thead>
<tr>
<th>No.</th>
<th>Measured WCET (cycles)</th>
<th>Estimated WCET (no user constraints) (cycles)</th>
<th>Estimated WCET (FFL constraints) (cycles)</th>
<th>Estimated WCET (FFL constraints) (ratio )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46,092</td>
<td>2,085,590</td>
<td>48,771</td>
<td>loop bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1) (F_{1,L1,N} \leq 20),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2) (F_{2,L1,N} \leq 4),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>impractical paths</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(pre-defined data type, exception):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3) (F_{1,L1,N8} = 10),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4) (F_{1,L1,N14} \leq 2),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5) (F_{1,L1,N15} \leq 9),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6) (F_{1,L1,N20} = 0),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7) (F_{1,L1,N23} = 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>infeasible paths:</td>
</tr>
<tr>
<td>2</td>
<td>147,336</td>
<td>2,503,500</td>
<td>211,966</td>
<td>loop bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1) (F_{1} \leq F_{1,N13}),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2) (F_{1,N5} \leq F_{1,N15}),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3) (F_{1,N7} \leq F_{1,N17})</td>
</tr>
<tr>
<td>3</td>
<td>839</td>
<td>898</td>
<td>107.1%</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>77,148</td>
<td>717,250</td>
<td>78,370</td>
<td>loop bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1) (F_{1,L1,N} \leq 23),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2) (F_{2,L1,N} \leq 5),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>impractical paths (no exception):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3) (F_{1,L1,N19} = 22)</td>
</tr>
<tr>
<td>5</td>
<td>41,460</td>
<td>1,048,560</td>
<td>66,274</td>
<td>infeasible paths:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1) (F_{1,N3} + F_{1,N5} \leq 1),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2) (F_{1,N3} \leq F_{1,N10}),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3) (F_{1,N10} + F_{1,N20} \leq 1),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4) (F_{1,N5} \leq F_{1,N20}),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>loop bounds:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5) (F_{1,L1,N} \leq 70),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6) (F_{1,L1,L1,N} \leq 5),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7) (F_{1,L1,L2,N} \leq 5),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8) (F_{1,L2,N} \leq 6),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>impractical paths (no exception):</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9) (F_{1,L1,N8} = 0)</td>
</tr>
<tr>
<td>6</td>
<td>5,401</td>
<td>7,797</td>
<td>7,605</td>
<td>infeasible paths:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1) (25 \times F_{1,N3} \leq F_{1,L1,N28}\</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2) (F_{1} &lt; &gt; : F_{1,L1,N7} +</td>
</tr>
</tbody>
</table>

45
Table 8.1: (continued)

<table>
<thead>
<tr>
<th>No.</th>
<th>Measured WCET (cycles)</th>
<th>Measured WCET (no user constraints)</th>
<th>Estimated WCET (with FFL constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>31,512</td>
<td>37,322</td>
<td>32,137</td>
</tr>
<tr>
<td></td>
<td></td>
<td>118.4%</td>
<td>loop bound:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1) $F_1.L_1.N_{header} \leq 5$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2) $F_1.L_1.L_{header} \leq 3$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3) $F_1.L_1.L_{header} \leq 4$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4) $F_1.L_1.N_{header} \leq 6$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>infeasible paths:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5) $F_1.L_1.N_7 = 1$</td>
</tr>
</tbody>
</table>

Table 8.2: WCET estimation by TimeBonder2 using FFL and UCL.

<table>
<thead>
<tr>
<th>No.</th>
<th>Estimated WCET (cycles)</th>
<th>Estimated WCET (with FFL constraints)</th>
<th>Estimated WCET (with UCL constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>48,771</td>
<td>105.8%</td>
<td>48,771</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) $F_1.L_1.N_{Header} \leq 20$,</td>
<td>1) $F_1.L_1.N_{Header} = 10$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) $F_2.L_1.N_{Header} \leq 4$,</td>
<td>2) $F_1.L_1.N_{Header} = 3$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impractical paths (pre-defined data type, exception)</td>
<td>3) $F_1.L_1.N_{Header} = 10$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) $F_1.L_1.N_8 = 10$,</td>
<td>4) $F_1.L_1.N_{Header} = 1$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4) $F_1.L_1.N_8 = 2$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5) $F_1.L_1.N_8 = 9$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6) $F_1.L_1.N_8 = 0$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7) $F_1.L_1.N_23 = 1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>211,966</td>
<td>143.9%</td>
<td>211,230</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infeasible paths:</td>
<td>causal dependency constraints:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) $F_1.L_3.N_{Header} \leq 3$,</td>
<td>1) $F_1.L_3.N_{Header} \leq 10$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) $F_1.L_5.N_{Header} \leq 3$,</td>
<td>2) $F_1.L_5.N_{Header} \leq 3$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) $F_1.L_7.N_{Header} \leq 3$,</td>
<td>3) $F_1.L_7.N_{Header} \leq 3$,</td>
</tr>
<tr>
<td>3</td>
<td>898</td>
<td>107.1%</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>107.1%</td>
</tr>
<tr>
<td>4</td>
<td>78,370</td>
<td>101.6%</td>
<td>78,370</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impractical paths (no exception):</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) $F_1.L_1.N_{Header} \leq 23$,</td>
<td>1) $F_1.L_1.N_{Header} = 22$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) $F_2.L_1.N_{Header} \leq 5$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) $F_1.L_1.N_{Header} = 30$,</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>66,274</td>
<td>159.9%</td>
<td>65,906</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infeasible paths:</td>
<td>causal dependency constraints:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) $F_1.L_3.N_{Header} \leq 1$,</td>
<td>1) $F_1.L_3.N_{Header} = 10$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) $F_1.L_5.N_{Header} \leq 1$,</td>
<td>2) $F_1.L_5.N_{Header} = 10$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) $F_1.L_10 \L_1.N_{Header} \leq 1$,</td>
<td>3) $F_1.L_10 \L_1.N_{Header} = 1$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4) $F_1.L_20 \L_1.N_{Header} \leq 1$,</td>
<td>4) $F_1.L_20 \L_1.N_{Header} = 1$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>loop bounds:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5) $F_1.L_1.N_{Header} \leq 70$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6) $F_1.L_1.N_{Header} \leq 350$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7) $F_1.L_1.N_{Header} \leq 350$,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8) $F_1.L_20 \L_1.N_{Header} \leq 6$,</td>
<td>8) $F_1.L_20 \L_1.N_{Header} = 6$,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impractical paths (no exception):</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9) $F_1.L_1.N_{Header} = 0$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7,603</td>
<td>140.8%</td>
<td>7,237</td>
</tr>
<tr>
<td></td>
<td></td>
<td>infeasible paths:</td>
<td>causal dependency constraints:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) $F_1.N_3 \leq $</td>
<td>1) $F_1.N_3 \leq $</td>
</tr>
</tbody>
</table>

46
Table 8.2: (continued)

<table>
<thead>
<tr>
<th>No.</th>
<th>Estimated WCET (with FFL constraints)</th>
<th>Estimated WCET (with UCL constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cycles)</td>
<td>(with FFL constraints)</td>
</tr>
</tbody>
</table>
| 3   | 32,137   | F_1 L_1 N_28, 2) F_1 < ~> F_1 L_1 N_7+  
  F_1 L_1 N_15 + F_1 L_1 N_21  
  + F_1 L_1 N_28 ≤ 1 | F_1 L_1 N_15 + F_1 L_1 N_21  
  + F_1 L_1 N_28 ≤ 1 | F_1 L_1 N_15 + F_1 L_1 N_21  
  + F_1 L_1 N_28 ≤ 1 | (cycles) |
|     | 102.0%   | 31,512              | causal dependency constraints |
|     | 1) F_1 L_1 N_7 = 1 | 100% | iteration constraints: |
|     |           | 3) F_1 L_1 N_7[0:1] |
9. Conclusion and Future Work

Static WCET analysis provides a priori knowledge about the WCET of a program without running it. 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 real-time embedded systems which have 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 thesis, we proposed a new flow information representation language called UCL to facilitate the specification of user constraints. We chose finite automata to represent the static structure of program and to characterize its possible execution paths. User constraints specified in UCL formulas are converted into corresponding finite automata. They are combined with the automaton representing control flow graph of the target program through cross production. The combined automaton does not contain infeasible or impractical execution paths which are main causes of loose estimation of WCET. UCL’s visual notations are intuitive and expressive enough to specify complex flow information which cannot be specified by existing flow representation languages. 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 with the codes of real-time systems in various domains to evaluate the usefulness of the UCL notations. 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.
要 約 文

정확한 최장수행시간 예측을 위한 사용자 제약사항의
시각화 및 정형화

실시간 시스템 소프트웨어는 정해진 시간 내에 필요한 작업을 수행하여야 하므로 각 소프트웨어 수행 타이밍 분석이 필수적이며 이 중 최장수행시간을 구하는 것이 중요하다. 기존의 실시간 시스템 소프트웨어의 타이밍 분석은 개발자의 수작업에 의존하여 많은 시간과 노력이 요구되며 정확성에 문제가 있을 수 있는 단점이 있었다. 이를 해결하기 위해 자동화된 정적 타이밍 분석에 대한 연구가 활발히 진행되고 있다. 자동화된 정적 타이밍 분석기법은 실제로 목적시스템에서 소프트웨어를 수행하여 타이밍을 측정하는 대신 프로그램 코드와 대상 하드웨어
시스템 특성을 분석함으로써 최장수행시간을 예측하는 기법이다. 그러나 정적 타이밍 분석기법은 실제 프로그램의 동적 행위나 프로그램이 수행되는 환경에 대한 가정이 충분히 고려되지 않을 경우 실제보다 과대평가된 최장수행시간을 제공한다. 보다 정확한 최장수행시간을 예측하기 위해서는 사용자가 추가로 프로그램 제어흐름에 대한 정보를 제공하여야 한다.

본 논문에서는 사용자 제약사항의 명세를 위한 새로운 방법과 시각적 언어인
UCL (User Constraint Language)를 제안한다. 본 논문은 소프트웨어 프로그램과 제어흐름 정보를 finite automata로 정형화하여 나타낸다. UCL 언어는 직관적인 시각적 기호를 제공하여 사용자가 프로그램 수행경로를 제한하기 위한 다양한 수준의 제어흐름 정보를 기술 가능하게 한다. 사용자가 UCL로 기술한 제약사항은 finite automata로 변환되며, 프로그램의 제어흐름도를 나타내는 finite automaton과 결합
된다. 결합된 finite automaton은 사용자가 제한한 프로그램 수행경로만을 포함하
고 있어 이를 path-based 또는 IPET-based 제산기법을 사용하여 보다 정확한
최장수행시간 예측을 구할 수 있게 된다. 본 논문에서는 UCL 언어의 시각적
기호의 사용법을 예를 들어 기술하고 UCL의 formal syntax 및 semantics를 정의하다. 또한 UCL 공식을 finite automata로 변환하고 이들 간의 consistency를 검사하는 알고리즘을 제시하였다. 이를 기존의 정적 타이밍 분석도구에 구현하고 인공위성

49
소프트웨어에 적용하여 기존의 언어로는 명세하기 어려운 프로그램 제어흐름 정보를 UCL을 사용하여 기술함으로써 보다 정확한 최장수행시간을 구할 수 있음을 실증하였다. 또한 본 논문에서 제시한 UCL은 특정한 계산기법에 제한되지 않고 path-based 나 IPET-based 기법에 모두 적용 가능한 장점이 있다.
References


