DOCUMENTATION AND TOOLS TO SUPPORT WORST CASE EXECUTION TIME ANALYSIS

Documentation and Tools to Support Worst Case Execution Time Analysis

By
Jian Sun, B.S.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
Documentation and Tools to Support Worst Case Execution Time Analysis

AUTHOR: Jian Sun, B.S.
(Shandong Normal University, China)

SUPERVISOR: Dr. Alan Wassyng

NUMBER OF PAGES: xii, 141

Abstract

Knowing the timing behavior is essential when designing and inspecting real-time systems. Especially, the Worst Case Execution Time (WCET) of a program is of the utmost importance for schedulability and other timing analyses. The industrial deployment of critical systems presents an urgent need for WCET analysis methods and tools.

This thesis represents how the Display documentation method, introduced by Parnas and his colleagues in [32], is extended and used to aid WCET analysis and WCET Tool development. The work is performed within a Reverse Engineering Project, which has to recover high-level requirements of IBM 1800 assembler applications. Specifically, the displays are (primarily) manually composed from code, and then used by timing analysts for program understanding and flow analysis, which are essential phases in timing analysis. The thesis combines the Display documentation method with the WCET analysis techniques to solve several general problems in determining the upper bound of program execution time. It also includes a detailed example of a WCET analysis tool based on the documentation method.

Acknowledgments

First of all, I would like to express my sincere gratitude to my supervisor Dr. Alan Wassyng for his guidance, support, and faith throughout the research of this thesis, and to Drs. David L. Parnas and Robert Baber who also helped to supervise my research and provide direction and advice.

I wish to thank Dr. Mark Lawford, Dr. Wolfram Kahl, Dr. Ridha Khedri, Doris Burns and all my colleagues in the Reverse Engineering Project for their help and support during the research, and I would like to thank all the members of the Software Quality Research Laboratory.

I also thank Ontario Power Generation Inc. for the examples on which I could test my ideas and tools.

Special thanks to my wife, my son and my parents, for their endless love, encouragement and support.

Last, but not least, I would like to acknowledge the financial support from Ontario Graduate Scholarship of Science and Technology (OGSST) and the Natural Science and Engineering Research Council (NSERC).

Contents

List of Figures

Chapter 1

Introduction

This chapter provides an introduction of this thesis, including the motivation, project background, contributions and thesis outline.

1.1 Motivation

Industrial real-time systems impose timing requirements on functions implemented in the applications. The requirements imply *lower/upper limits* on the execution time of programs, i.e., it has to be guaranteed that the execution of a task does not take shorter/longer than the specified amount of time. Thus, extracting program execution timing properties is very important for software design especially in schedulability analysis and scheduling, and for testing and verification. In general, access to requirements specification and relevant design documents is desirable for performing the extraction task. However, in most cases, such documents are not complete nor accurate for inspectors to review the program.

This thesis presents a method that can be used for software designers and implementers to specify relevant program timing information precisely in their system documentation for further timing analysis, inspection and verification. The method is based on the *Display Method*, a form of precise documentation of well-structured programs introduced by Parnas, Madey and Iglewski. In [32], the authors do not explicitly address timing behavior. However, timing analysis can be viewed as a ver-

ification activity, and the analysts need to refer to such documents that can specify the programs' behavior precisely, systematically and readably. Thus, we plan to add a new variable T representing program execution time in the tabular expressions (function tables) [17][34] for timing analysis. The method has been used as the basis of a *Timing Analysis Tool* for a *Reverse Engineering Project* at McMaster University, discussed in the following sections.

1.2 A Reverse Engineering Project Background

Originally, our target was to recover timing information during the reverse engineering of legacy assembler control applications for which we have little or no requirements or design documentation. This work was motivated by the project, Reverse Engineering of High Level Requirements from Assembly Code, which is funded by Communications and Information Technology Ontario (CITO) and Ontario Power Generation (OPG). Five teams within the Department of Computing and Software at McMaster University, as well as engineers from OPG are involved in the project.

1.2.1 The Goal of the Project

The goal of the project is to create *methods* and *tools* to assist a developer in reverse engineering a legacy assembly language program to a high level requirements specification that is independent of arbitrary design decisions (but still captures the rationale of those decisions in terms of non-functional requirements). The methods will not just test the requirements against the original program for consistency, but will also provide additional assurances that the requirements have been consistently captured and documented. The entire process will be supported by *tools* that make the tasks easier and produce more reliable results. The theoretical results from the project will have practical application to the problem of reverse engineering requirements from legacy assembler programs.

In this project, like many, we have no software requirements and no software design documents. We want to recover program information, such as blocks of code that implement cohesive functionality, and to use this information to specify high-

level requirements. The target software is assembler code, and the program is not well structured and program variables as well as individual functions are not identified in any obvious way. The planned *tool* suite architecture of the reverse engineering project is shown in Figure 1.1. Note that the arrows in the figure represent data flow.

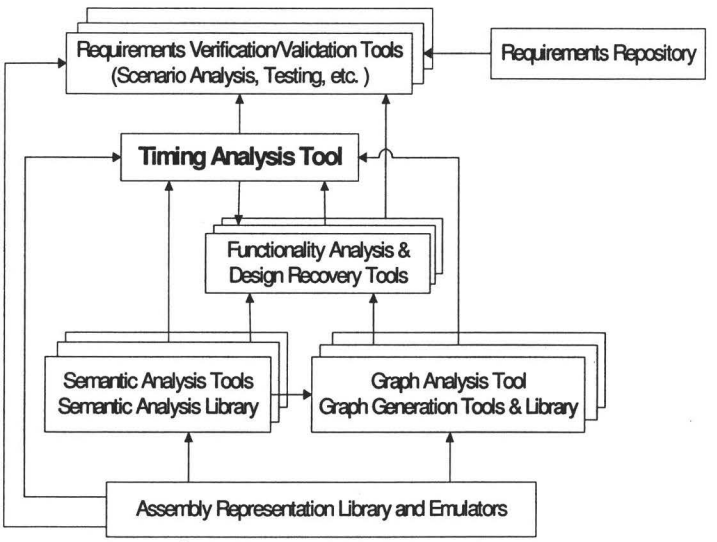


Figure 1.1: Reverse Engineering Tool Suite Architecture

In particular, the *Timing Analysis Tool (TAT)* is responsible for aiding timing analysts performing program flow analysis and determining the timing constraints in the assembly programs. Those constraints will be used to specify the timing requirements in the reverse engineered high-level requirements. The above architecture shows how the TAT fits into the suit. In our case, we focus on exploring the *Worst Case Execution Time (WCET) Analysis Tool* (*WAT*), a specific *TAT*, which calculates the worst (*upper limit*) possible execution time of a piece of code. Although the WCET can refer to both upper and lower bounds, we will be interested primarily in upper bounds.

1.2.2 IBM 1800 Assembler Language

The provided program, *Boiler Pressure Control (BPC)* code running on an emulator of the IBM 1800 Data Acquisition and Control System, is one of the modules which control the power generation plant. The application was written in a particular assembler language of the *IBM 1800* machine introduced in the 1960s. The legacy machine was designed to handle a wide variety of real-time applications, including process control and high-speed data acquisition.

The IBM 1800 assembler-language has more than 30 basic statements used to store/retrieve information, perform arithmetic and logic functions, control program and other operations. It also supports Macro assembler manipulations. Due to the small register space, most of the program instructions and data are stored in core storage and are manipulated through the registers such as Instruction Register (I), the Accumulator (Acc) and Accumulator Extension (Q) registers, and three Index Registers ($XR1$, $XR2$, and $XR3$). All of the registers are 16 bits in length, and the instructions are either in 16-bit short form or in 32-bit long form. Furthermore, some instructions can perform multiple tasks. For example, the MDX (*Modify Index and Skip*) instruction can (1) modify the value of some storage unit and (2) perform alternative branch based on the result of the calculation.

The 1800 assembly programs were written in a strict format that includes six fields: optional labels, opcodes, optional format bit, optional tag bit, operands, and optional comments. Each of the fields has its well-defined column number ranges. The programs can be assembled by the 1800 assembler into either binary code for execution, or assembly list (LST) data files. In addition to the original assembly program, the LST file includes the memory address assigned to each instruction, hexadecimal object representation of instructions and data, and other valuable information. One of the groups in the Reverse-Engineering project is working on exploring automatic tools to generate control flow/data flow graphs by extracting relevant information from the LST files. The tool offers *control flow/data flow graphs* as output in *GXL (Graph eXchange Language)* format . Such control flow/data flow information is required for further timing and functionality analysis in the project.

1.2.3 Reverse Engineering and its Difficulties

In [47][48][49], *reverse engineering* is defined as "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information." Currently, most reverse engineering research involves program understanding, and focuses on code analysis including subsystem decomposition, concept synthesis, design and programming pattern matching, and dependence analysis.

However, the code does not contain all the information that is needed for program understanding. Also, abstractions extracted from the code generally miss the big picture behind the evolution of the software system. Typically, the software architecture, design and implementation patterns and physical/business constraints known by the forward engineers who design and implement the software, are not obvious or complete to the reengineers. Moreover, over time, other features, such as software updates, staff migration, document decay, and an increase in complexity make it harder for program reviewers to figure out the program behavior. Thus, it is extremely important for both forward engineering and reverse engineering staff to document their work precisely and systematically for further inspection or verification.

1.3 Contributions and Thesis Scope

This thesis is addressed to illustrate how we successfully applied a precise documentation method to WCET analysis, and developed an associated tool in a reverse engineering project. The following is a list of specific contributions:

- Extended the Display Method for Software Execution Time Analysis
- Constructed a documentation-driven WCET Analysis Tool Architecture
- Implemented several components of the WAT suite to aid analysts in solving several general timing analysis problems met in a reverse engineering project

The remainder of this thesis is divided into four major parts. Firstly, chapters 2 and 3 introduce the literature of WCET analysis and the Display documentation method.

Next, chapter 4 introduces the difficulties met in the practical project. Then, chapters 5 and 6 represent how the Display method is extended and applied in constructing precise documents to aid both WCET analysis and tool development. Note, some of the tools are explored for solving the problems in chapter 4. Further, chapter 7 concludes solutions and suggests future work. Finally, other relevant information is attached in Appendixes.

Chapter 2

Overview and Literature Survey of WCET Analysis

This chapter briefly introduces concepts in WCET analysis, and references relevant literature.

2.1 Overview of WCET Analysis

The goal of WCET analysis is to generate a safe (no underestimation) and tight (small over-estimation) estimate of the longest execution time (upper bound) of the program [27]. Note that generally, WCET can refer to both upper and lower bounds, but conventionally, these are referred to WCET and BCET (Best Case Execution Time) analyses, respectively. In this thesis, only the upper bound is considered because BCET analysis is similar to WCET analysis.

2.1.1 Why analyze the WCET?

The concept of a worst-case execution time for a program has been part of the real-time community for many years, especially when doing schedulability analysis and scheduling [44]. In particular, many scheduling algorithms and all schedulability analysis assume some form of knowledge about the worst-case timing of an essential task.

Generally, in any product development where timelines are important, WCET analysis is a natural tool to apply. This is because designing and verifying hard real-time systems can be simplified by using WCET analysis instead of extensive and expensive testing. For instance, WCET estimates can be used to verify that the response time of a critical piece of code is short enough, that interrupts handlers finish quick enough, or that the sample rate of a control loop can be maintained. Moreover, WCET estimates can be used to determine whether performance goals are met for periodic tasks, to check that interrupts have sufficiently short reaction times, to find performance bottlenecks, to assist in selecting appropriate hardware and for many other purposes [8].

2.1.2 What features should be analyzed?

To determine the upper bound of execution time of a program, WCET analysis takes into account program flow information, hardware/low-level performance effects and appropriate calculation methods [8]. Correspondingly, the analysis phases are divided into flow analysis, low-level analysis and WCET calculation. Briefly, flow analysis extracts and represents program flow information that provides information about possible ways the program can execute, which functions get called, and how many times loops iterate. Low-level analysis determines the execution time of each basic block, a piece of code that is executed in sequence (contains no jumps and branches, and there are no jumps into the sequence). Further, given the flow information and the execution time of each basic block, the WCET calculator is responsible for finding the program path that takes the longest time in the entire target program.

Moreover, program execution time analysis generally is performed in specific circumstances and under particular conditions, and when performing the WCET estimation, normally, it is assumed that:

- there are no interfering background activities
- the program execution is finite
- the program cannot be interrupted nor preempted by others

2.2 Program Flow Analysis

Program flow analysis extracts and identifies the possible ways a program can execute. Conventionally, the program's executable paths are represented in directed (flow) graph form, in which nodes are executable instructions or basic blocks, and directed edges present the execution sequence. Thus, the task of the flow analysis can be divided into a set of subtasks such as: flow graph generation, path feasibility identification, and loop bound determination.

2.2.1 Flow Graph Generation

In general, the structurally possible flows of a program can be extracted from either the program source code or compiled code, and flow graphs constructed through the resources discussed below.

- Compilers: Some computer language compilers not only can convert high-level language code into object code, but generate flow information during compiling. For example, in [9], the researchers introduced a flow analysis module which tightly coupled with their compiler to collect flow information and generate a control flow graph.

- Assemblers: As well as interpreting assembler code into object code, some assemblers also can provide flow information. For instance, as mentioned in 1.2.2, the IBM 1800 assembler can generate LST data files including instruction addressing information which indicates the execution flow. In the project introduced in 1.2, a flow graph generation tool was developed to generate GXL format flow graphs [12].

- Libraries: The hardware/software vendors normally provide product libraries (and manuals), in which some hardware/software features are specified, and such knowledge is valuable in determining accurate flow information.

- Implementer's annotations: So far, in some WCET analysis methods [10][11], the flow information is given by manual annotations from designers and implementers rather than by fully automatic tools. This is because some program

flow properties, e.g. the maximum iterations of a loop slice, are very hard for tools or program reviewers to figure out, but known by the designers and implementers. Other research work tries to explore computer-aided tools to reduce the dependence on manual interventions, rather than attempt to fully automate the flow information extraction. In other words, automatic tools are designed for extracting some of the flow structure, and manual annotations are helpful in supplying and refining the solutions.

However, these graphs may have a huge number of possible execution cases or may be unmanageable in size because infeasible paths may increase the graph complexity exponentially. Hence, analysts have to impose constraints on the graphs to decrease the complexity.

2.2.2 Infeasible Path Identification

Infeasible paths are program paths that cannot be executed. Normally, analysis of finding infeasible paths is important to decrease the complexity of a generated flow graph and makes the WCET analysis more efficient, but it is not necessary to find every infeasible path because the path may be safe within the context of a WCET analysis. Moreover, feasible paths must not be noted as infeasible since it might lead to an underestimation.

In general, infeasible paths may be caused by either semantic or syntactic reasons. For instance, figure 2.1 shows an example program [8] containing a semantically infeasible path. In particular, in the loop body, the branch condition of the second $if - then - else$ statement, $x = 1$, is never valid, i.e. statement (S3), $x := x + 2$, will never be executed. A detailed explanation is given in 4.4. Moreover, in 4.4, an example of an infeasible path cased by syntactic interpretation of IBM 1800 machine statements is described. In [5][10] a Flow Information Language was used to express flow constraints as arithmetic relation expressions for WCET analysis. In particular, the language can be used to model flow graphs as scopes where a scope is defined as a certain repeating or differentiating code segments in a program like a function call or iteration loop, annotated with flow constraints, named *facts*. Some of the constraints

```
/** Input limits for x : 0 ≤ x ≤ 3 **/
readln(x);
While (x < 4) Do {
    if (x < 3) then x := x * 2;        (S1)
    else x := x + 1;                   (S2)
    if (x = 1) then x := x + 2;        (S3)
    else x := x + 1;                   (S4)
}
```

Figure 2.1: An Example Program

can be generated automatically e.g. $if-then-else$ exclusive branches, and some are annotated manually for items such as loop bounds and infeasible path identification.

If given appropriate documentation, flow information such as feasibility knowledge can be obtained from both source code and relevant documents. However, if there is no relevant documentation, as in many reverse engineering projects, or only part of the documentation is provided, it is much harder for analysts to extract such information. In some cases, they depend on designers' and implementers' annotations.

2.2.3 Loop Identification and Loop Bound Determination

Knowing the maximum number of loop iterations, called the *loop (upper) bound*, is required for getting a tight WCET solution. In [5][10], the loop bound information is obtained from manual annotations, and in [22][25] relevant research work is aimed at performing loop bound approximation automatically or semi-automatically.

The difficulty of determining the loop bound is largely dependent on how complex the loops are. For example, nested loops and recursion structures make it much harder to analyze and understand iterative behavior. For high-level languages, it is not difficult to determine loop structure and iteration blocks, but for assembler language, the loop structure is not obvious and it is not easy to determine iteration blocks because arbitrary jumps may occur during the execution. Moreover, for analysts to determine the loop bound, detailed understanding of the program is required. However, this is normally hard for program reviewers when the applications are legacy,

complex and poorly documented.

In SQRL (Software Quality Research Laboratory), McMaster University, researchers use *tabular expressions (function tables)* [51], to specify program behavior. Such specifications are not only precise, they are also usually easier to understand than the code. Some table tools also were explored to manipulate table operations. In the project, some loop assembly code slices were identified and function tables were manually constructed. Then, their loop bounds were determined by executing some automatic table tools. A detailed example of determining the loop bound of the program shown in figure 2.1 is illustrated in 4.4.2-3.

2.3 Low-level Analysis

The purpose of low-level analysis is to determine the execution time of atomic machine statements and basic blocks, to find out the timing effects caused by hardware architecture that can improve or delay a program's timing performance, and to model such timing information in a proper form.

2.3.1 Hardware Feature Identification

For modern computers, instruction execution time is commonly affected by advanced hardware features such as pipelined CPUs, caches and branch predictors. To determine these hardware effects, specific analyses should be performed. For example, in [16], instruction cache, and in [7], global pipeline analysis methods are illustrated. Further, it is common for analysts to get much of the above information from the hardware manual provided by the product vender.

For some older machines and micro-controllers, like the IBM 1800, each instruction (in same format) has a fixed execution time, and the execution time of basic blocks can be determined by simple addition. Moreover, the IBM 1800 system provides three "internal timers" that can supply real-time information for the program. They can be started or stopped under program control. Once started, they are automatically incremented, one count at a time, by the cycle stealing facility of the process-controller. Thus, in the Reverse-Engineering project, both the instruction and basic block exe-

cution time can be obtained from the timers. Note, the instruction execution time is also specified in the hardware manual.

2.3.2 Simulation

Currently, simulation techniques are widely used to measure the execution time of program segments to help analysts determine average instruction execution time, and identify the performance effects caused by modern architectures [27]. Some researchers also apply static analysis to the above hardware features [16][23].

The IBM 1800 also contains a Time-Sharing Executive System (TSX), a real-time, process-control programming system that affords user an easy means of generating, testing and executing a complete process control program. The user's process programs are built in the non-process monitor mode, tested by the TSX simulator, and executed in the on-line, process-control mode. Therefore, some program execution scenarios can be simulated to verify the timing analysis solutions.

2.3.3 Timing Graph Generation

To model the timing information for WCET analysis, it is efficient to cluster individual statements into basic blocks (atomic units) of the object code. Then, timing graphs can be generated through assigning relevant timing information into the program flow graphs to represent the low-level analysis results.

In the Reverse-Engineering project, the original GXL flow graph [12] is clustered first, in which the graph nodes are basic blocks. Next, the execution time of each block is determined. The timing information is assigned to edges rather than nodes in the graph see section 6.2.1.

2.4 WCET Calculation

WCET calculation is the last step for WCET analysis. It is responsible for combining results obtained from flow analysis and low-level analysis to generate the final solution. In the literature, WCET calculation methods can be divided into three categories: 1)

tree-based [7] 2) path-based [40][44] and 3) the Implicit Path Enumeration Technique (IPET) [24].

2.4.1 Tree-based calculation method

In the tree-based method, the final WCET is calculated by traversing a tree, representing the flow of the program and assigned execution times for each node, in a bottom-up way. This method is good for estimating the WCET for well-structured programs that are not very complex, but not good for handling unstructured flows like assembler code. Note, although it is well known that any program can be rewritten so that it has a tree structure, it is not practical to reconstruct the legacy unstructured programs for their functionality and performance analysis. Thus, the tree-based method normally is used to calculate estimates for small parts that can be integrated in order to generate the WCET solution for large parts of the program.

2.4.2 Path-based calculation method

For the path-based category, the WCET solution is generated through searching for the path with the longest execution time in the timing graph of the target program. A number of theorems and algorithms were proposed in the literature. For instance, Figure 2.2 shows Dijkstra's longest path search algorithm [44] which is used to find the worst execution case in the Timing Graph (TG). Nevertheless, another important problem for analysts is to determine whether the found path is feasible, and whether the found path is a desired functional computation path (a typical nonfunctional path example is an error-handling program). If not, further searching and analysis has to be performed.

In the Reverse-Engineering project, the timing analysis group has chosen the path-based method since the extended Display documentation method provides a good mechanism for this method, and also because the flow graph can be obtained from other groups in the project, graph analysis tool and functionality analysis information. The WCET calculation tool suite contains a set of sub-tools for clustering the original flow graph, searching for the longest path, detecting path feasibility, and tagging

```
/** Initialization **/
For each node v in TG Do
   predecessor[v] := null;
   timeSum[v] := 0;
endFor

/** Breadth first search **/
For each node u in TG in breath − first order Do
   For each outgoing edge e = (u, v) in TG Do
      d := timeSum[u] + t_u + δ_e;
       /** Is u on the longest path to v **/
      if timeSum[v] < d then
         predecessor[v] := u;
         timeSum[v] := d;
   endFor
endFor
return TG
```

Figure 2.2: Dijkstra's longest path search algorithm

nonfunctional paths. Detailed illustration will be presented in 6.2.1.

2.4.3 IPET calculation method

Avoiding a potential explosion in the number of examined paths, the Implicit Path Enumeration Technique (IPET) method determines the WCET estimate by maximizing the goal function [24] in Figure 2.3 (subject to specified constraints), where *Blocks* are basic program execution blocks decomposed by the flow analysis and *Edges* are the edges connecting blocks corresponding to the flow relationships represented in the timing graph.

This approach reduces the WCET problem to an optimization problem, and it can be solved either by constraint satisfaction methods (CSM) or integer programming (IP) [9]. Note, existing tools can support such techniques very well, e.g. lp-solver [52] can be used to solve the IP problem. Compared with other approaches, the IPET

Goal Function :

$$WCET = max\left[\sum_{\forall Block}(x_{Block} \cdot t_{Block}) + \sum_{\forall Edge}(x_{Edge} \cdot t_{Edge})\right], where$$

x_{Block} *is the number of times that the code Block has been executed;*
x_{Edge} *is the number of the times that the Edge has been passed;*
t_{Block} *is the execution time of the code Block for one iteration;*
t_{Edge} *is the time effect caused by the computer architecture when two continuous code Blocks connected by Edges are executed.*

Constraint Equation Group :

$$\begin{cases} constraintExpression\_1; \\ constraintExpression\_2; \\ \cdots\cdots\cdots\cdots\cdot \\ constraintExpression\_n. \end{cases}$$

Figure 2.3: IPET Goal and Constraint Functions

solution gives no information about the precise execution order (path) but simply delivers the worst-case count on each node. The method requires that we provide global constraints to construct the constraint function (group). Moreover, the IPET calculation is easy to integrate into new analysis methods. For instance, if analysts introduce new flow analysis methods, they just need to specify the new flow analysis solutions in proper constraint functions and do not need to change the calculator at all.

2.5 WCET Analysis Tool

2.5.1 Interactive Tool

As discussed in 2.2, at this stage, it seems too difficult to explore fully automated tools to perform WCET analysis. However, it is practical to design an interactive tool which requires intervention from people who know the program very well and have relevant domain knowledge of the system, to aid the timing analysts in determining the program's execution upper bound. It is also clear that user intervention occurs mainly during the flow analysis phase.

However, annotations normally are error prone and should be reduced. Furthermore, over time, memory decay, people leave, and complexity increases make it harder for obtaining accurate annotations for program reviewers to figure out the program's behavior. All of these urge the need for constructing appropriate and precise documents to aid timing analysis.

2.5.2 WCET Tool Architecture

Currently, it is widely accepted by WCET researchers that the WCET tool should contain three main modules to perform flow analysis, low-level analysis and WCET calculation. In [9][22][44], different WCET tool architectures are given, and some automatic or semi-automatic analysis tools were developed based on such architectures. In 6.1, a particular WCET tool architecture based on the combination of precise documentation and WCET analysis techniques discussed in the literature is proposed and illustrated.

Chapter 3

Literature Survey of Precise Documentation of Software

This chapter presents the literature on a precise documentation approach introduced by Parnas and his colleagues.

3.1 Overview of a precise documentation approach

When constructing computer systems, as when developing other engineering products, engineers are required to provide precise documentation to describe how they use science, mathematics and technology to build their products. Especially for software engineers, their documentation is vital for software development, verification, inspection and maintenance. In software engineering, software engineers can benefit from using mathematical notations to make their documents consistent, precise and complete.

3.1.1 Functional Documentation

In [33], Parnas and Madey discussed documentation at a high level of abstraction, dealing uniformly with many types of systems and documents. They defined the *content* of documents that should be provided in computer systems, rather than specifying the format or notation used in the documents. In this approach, instead

of vague, inaccurate, and intuitive language, they applied a mix of standard engineering and mathematical concepts to construct documentation. Particularly, all the essential properties of computer systems, and their components are seen as a set of mathematical *relations*. By describing these relations, software designers can construct *relational documentation*, also named *functional documentation*, to document their designs systematically and precisely. Associated with industry and company standards, the documents may be divided into the following categories.

- **System Requirements Document**, provides a black-box description of the system including descriptions of environmental quantities of concern to the system, denoted as mathematical variables, and relationships between the values of the quantities that result from physical and other constraints.

- **System Design Document (SDD)**, describes the hardware structure and how the computers in the system communicate. It also determines the relationship between the inputs and outputs, also denoted as mathematical variables, and the environmental variables identified in the system requirements document.

- **Software Requirements Document (SRD)**, is extracted from a System Requirements Document and System Design Document to determine the software requirements.

- **Software Behavior Specification (SBS)**, specifies actual software behavior.

- **Software Module Guide (SMG)**, describes the system module decomposition and the responsibilities of each module.

- **Module Interface Specification (MIS)**, provides a black-box description of access-programs for each module specified in the SMG and the effects of using them.

- **Module Internal Design Document (MIDD)**, provides a clear-box specification of implementations of the modules listed in the SMG.

- **Data-flow Document**, describes the "data flow" between variables or between communicating sequential processes.

- **other documents** such as: Service Specification, Protocol Design Document, Chip Behavior Specification and User-relation Document are normally used in some particular cases and will not be disscussed in this thesis.

Each kind of document represents one or more relations. For instance, as described in [17], a system requirements document should contain the representation of two relations. NAT describes the environment, and REQ describes the effect of the system when it is installed. It is important to note that "*relation*" means "*binary relation*", and, as defined in [28], a *binary relation* R on a given set U is a set of ordered pairs with both elements from U, i.e., $R \subseteq U \times U$. The set U is called the universe of R. The set of pairs, R, can be described by its characteristic predicate, $R(p, q)$, i.e., $R = \{(p, q) : U \times U \mid R(p, q)\}$. The domain of R, denoted as $Dom(R)$, is $\{p \mid \exists q\ [R(p, q)]\}$, and the range of R, denoted as $Range(R)$, is $\{q \mid \exists p\ [R(p, q)]\}$.

Unlike some impractical approaches, the above documentation theories were successfully applied in a variety of military and civilian applications. For example, (1) an early version of the relational requirements model was used to write a software requirements document for the Onboard Flight Program used in the U.S. Navy's A-7 aircraft [14][15], and (2) the relational model and the program documentation model were used to inspect a safety-critical program for the Darlington Nuclear Power Generation Station in Ontario, Canada [20][39][50]. All of the industrial experience shows that the relational documentation theories are useful.

3.1.2 LD-Relations and Program Description

Based on the views that

1. digital computers are *finite state machines*

2. a *program* is a text description of a set of states in such machines

3. a *program execution* is a sequence of states of a program

program executions can be described through a kind of mathematical relation, *Limited Domain Relation* (*LD-Relation*). As defined in [36], a *Limited-Domain Relation L* on the universal set U is an ordered pair (R_L, C_L) where R_L is a binary relation on U and C_L, called the *competence set* of L, is a subset of the domain of R_L.

In detail, the relation component of a LD-relation describing a program is the set of states (x, y) such that when the program is executed starting in state x it may terminate in state y. The competence set of that LD-relation is the set of states in which the program is guaranteed to terminate. For example, a particular LD-relation can be represented as $((x, y), C_L)$ where: x is one of the start states, y is one of the termination states, and C_L is the state set in which termination is guaranteed. Note, in LD-Relation descriptions, only the start and termination states are documented and the intermediate states are ignored, and U is the universal set of machine states.

Furthermore, considering our particular task, analyzing program execution time, in this thesis, we pay more attention to the documentation of program effects. This is because, as discussed in chapter 2, without knowing the behavior and effects of a program, it is impossible to figure out its timing properties. In [32], Parnas, Madey and Iglewski introduced a precise documentation method, *Display*, to document software products. This method uses LD-Relation representations, in *tabular* form, to specify and describe programs. The method has the advantage that it can be used to document the effects of large program precisely and understandably. The following sections will illustrate relevant theories and the application of the method.

3.2 Tabular Representation in Functional Documentation

Industrial software systems are developed for solving scientific and engineering problems. In practice, such problems are modeled using mathematical models (by experts in the system's subject area) first, then, mathematicians determine how to solve the models and software developers determine how to design and implement the software products. Parnas and other researchers who proposed a *functional documentation* approach found that using conventional mathematical expressions to represent the

relations is too complex and hard to parse, and instead, using *tabular expressions* (*tables*) [19] is much more practical.

3.2.1 Why use tabular expressions?

First of all, as stated in [17], functions implemented in digital computers have many discontinuities, which can occur at arbitrary points in the domain of the function, and tables are ideal for describing such functions. Further, it is very common that a function's domain and range have distinct types. Thus, it is sometimes difficult to use traditional mathematical notations to describe the functions in an understandable way. However, *tabular representations* can be used to describe the functions in a succinct and readable format [51].

Secondly, tables enable us to represent relations with multiple conditions completely and concisely. Especially, in computer programming, many conditional expressions are involved, and tables are good at representing them.

Thirdly, tabular representations can simplify the process of the documentation. As discussed in [42], tables have the following advantages in making their representations simple.

- The table parses the expression for readers. Many nested pairs of parentheses are eliminated, and the interned structure of the expression is revealed.
- The table eliminates repetitions of the sub-expressions that appear in column headings.
- Since each table entry only applies to a small part of the relation domain, the expression in the entry can be simplified

Furthermore, as discussed before, computer systems are often constructed by people who come from different fields. Although all may work in domains in which mathematical models are relevant, the communication among these team members can be difficult because of their different backgrounds. However, tabular expressions based on *predicate logic* [31] (which is easy to learn, use and understand) can make the

descriptions of mathematical models clear and more readable to people with diverse backgrounds.

Finally, tabular representation can help model designers and reviewers in thinking and inspection [17]. For example, tables can be used to examine the completeness of a model easily and efficiently. Note that, tables are really helpful in the WCET flow analysis, especially in feasibility analysis as shown in 4.4.

3.2.2 Program Function Table

Software can be described as a set of *functions* and associated output data items. Each function determines the values for one or more output data items, and each output item is given values by one or more functions. Correspondingly, the tabular expression for the function is called *function table.*

In [18][21], tabular expression notation was defined precisely as a mathematical notation. Function tables also were categorized in a variety of forms such as *normal function table, inverted function table, vector function table, normal relation table* and so on. They appear to be useful in specific circumstances.

Examples of function tabular expressions

Following are three examples of program function tables in three kind of forms.

(1). As illustrated in [18], function $f(x, y)$, where

$$f(x,y) = \begin{cases} 0 & if\ x \geq 0 \wedge y = 10 \\ x & if\ x < 0 \wedge y = 10 \\ y^2 & if\ x \geq 0 \wedge y > 10 \\ -y^2 & if\ x \geq 0 \wedge y < 10 \\ x + y & if\ x < 0 \wedge y > 10 \\ x - y & if\ x < 0 \wedge y < 10 \end{cases}$$

can be represented in the following *normal function table*, which is precise and more readable.

			H1
$\frac{H1 \wedge H2}{G}$	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$	$x - y$
H2			G

Note, $\frac{H1 \wedge H2}{G}$ indicates that *Headers* H1 and H2 represent the predicate conditions, and *Grid* G represents the solutions of function $f(x, y)$.

(2). As shown below, a program, named *maxvalue*, finds the maximum of two input variables a and b, and saves the value in variable max.

```
Procedure maxvalue()
{
real a, b, max;
readln (a, b);
if (a >= b) then max := a;
else max := b;
return(max);
}
```

The following vector table represents the behavior of the program.

	$a \geq b$	$a < b$
max =	a	b

$\wedge NC(a, b)$

Notes, (1) $NC(variable\_list)$ represents that the variables in the *variable_list* are not changed after the program execution, and (2) in later chapters, tables in this form will omit operator "=" and take the first column as *result* column. (3). Using vertical bar, "| ", which means "such that" as explained in [32], the table in (2) can be rewritten as below.

	$max \mid$
$a > b$	$max = a$
$a < b$	$max = b$

$\wedge NC(a, b)$

Note, when "| " is used, the entries in the column must be boolean expressions and the value of the variable must satisfy the predicate described in the relevant row.

In later sections and chapters, the above form of function tables is used to represent function behaviors.

3.2.3 Table Operations and Table Tools

At McMaster University, a research group was engaged in a *Table Tool System* (TTS) project exploring prototype tools to assist people using the above notations. Figure 3.1 [42] shows an overview of how the TTS system works. The kernel of the system is a "table holder" that creates objects representing tables in the internal (storage) format, and other separate tools, as shown in figure 3.1, can use the kernel to store and communicate tabular expressions. These tools can assist users to perform a variety of table operations such as table creating, inverting, checking and so on. Moreover, the researchers also proposed a variety of theorems and implementation algorithms for the TTS. For instance, in [17], transformations of tables of one kind to another and interrelations between transformations were stated, and in [42][43], some table inversion algorithms and transformation tools were proposed.

Currently, a *table composition tool* is being explored. It will be used in the Reverse-Engineering project to compose unit tabular expressions identified code slices to individual functions. In 4.4.2, a composition example is illustrated.

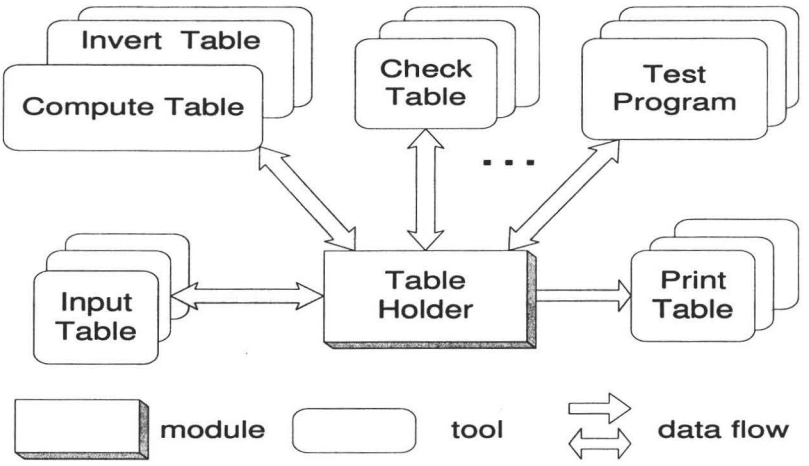


Figure 3.1: Overview of Table Tool System

3.3 The Display Documentation Method

3.3.1 What is Display?

Introduced by Parnas, Madey, and Iglewski [32], *Display* is a form of program documentation which can be used by software engineers as a reference documentation during inspection and maintenance. Formally, a *Display* is a document that consists of the following three parts:

1. $P1$: a specification for the program presented in this Display,
2. $P2$: the program itself. The names of other programs may appear in this text; we say that these programs are invoked in this Display,
3. $P3$: specifications of all programs (other than that specified in $P1$) invoked in $P2$ that are not known.

Note, in this approach, it is assumed that programs can be described by mathematical functions, and *function tables* are used to specify those functions, relations, and sets.

Since containing the precise specifications of all invoked programs, each Display can be reviewed and its correctness can be verified without reference to other Displays. Furthermore, it is important to know the following definitions.

- A Display is *correct* if the program in $P2$ will satisfy the specification in $P1$, provided that the programs invoked in $P2$ satisfy the specifications given in $P3$.
- A set of Displays is *complete*, if for each specification of a program that is found in $P3$ of a Display, there exists another Display in which this specification is in $P1$.
- A set of Displays is *correct* if 1) the set of Display is complete, and 2) all Displays are correct.

Display Documentation Method

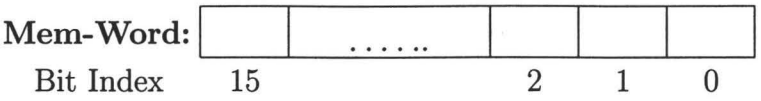
As defined in [32], *Display Method* is a precise, systematic and readable program documentation method. This documentation consists of a set of *Displays*, supplemented by a *lexicon*, a dictionary containing definitions of the terms used in the program being documented, and an *index*, a list of all the variables, programs, etc. indicating where those items appear in the Displays.

3.3.2 A Display Example

Supplemental Table Notations

Notation for tabular expressions is defined in [34]. The notation below has been designed to be used in function tables that describe the behavior of IBM 1800 assembler programs.

- $XXX[\,]$: denotes a one dimensional array with the index starting from 0, and each element is a machine word containing 16 bits.
- $< Num >$: denotes a bit value operation which returns the value at bit Num of a machine/memory word. Note, each word has bit 0 at the right, and bit 15 at the left as shown below.



- *.address* : denotes the address operation which returns the core storage address of a word or a label.
- "<< *Num*" and ">> *Num*": denote a shift left and a shift right *Num* bits, respectively.
- "<<< *Num*" and ">>> *Num*": denote a shift left and a shift right *Num* bits with extension register Q, respectively.

Function table format

All the function tables are formatted as follows.

- **Name**: the identifier of each function table.
- **External Variables**: the list of variables that are defined and assigned by other programs and used inside the target program.
- **Internal Variables**: the list of variables that are defined and initialized inside the target program, or used to save temporary values.
- **Preconditions**.
- **The function table**.
- **NC - not changed variables**: the list of variables are used without changing their original values during the execution of the target program.

It is important to note that the modifications of *Acc*, *accumulator register* and its extension register, *Q*, in some cases, are specified in the function table. This is because it is very common for program segments to use them for passing parameters, and tracking the values in such registers is required when studying a given program.

An example of an IBM 1800 program Display

Following is an example of using *Display* to represent an IBM 1800 assembly program segment, labelled *TRBFF*, which invokes a subroutine, labelled *DI2F3*.

1. **Display Specification**
   Name: TRBFF
   External Variables: XR1, XR2, BPCD[ ], GST[ ], DIW2.
   Internal Variable: Acc.
   $XR1 = GST.address \wedge XR2 = BPCD.address \Rightarrow$

	BPCD[13] $< 15 >= 1$	BPCD[13]$< 15 >= 0$	
		$(\sum_{i=9}^{11} DIW2 < i >) > 1$	$(\sum_{i=9}^{11} DIW2 < i >) \leq 1$
$Acc =$	*0*	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	*0*
$GST[47] =$	*0*	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	*0*

$\wedge\ NC(XR1, XR2, BPCD[\,], DIW2, GST[\,]$ *except* $GST[47])$

2. **Program**

Address	Label	Option	TF	Operands
(35be)	TRBFF	LD	2	13
(35bf)		BSC	L	TRBFD,E
(35c1)		LD		DIW2
(35c2)		SLA		9
(35c3)		BSI		DI2F3
(35c4)		MDX	3	-1
(35c5)		MDX		TRBFE
(35c6)	TRBFD	SLA		16
(35c7)		STO	1	47
(35c8)		MDX		TROUT
(35d5)	TRBFE	LD	2	
(35d6)		S	1	
(35d7)		M	1	48
(35d8)		SLA		5
(35d9)		STO	1	
(35da)		MDX		TROUT

3. **Display Specification**

Name: DI2F3

External Variables: Acc, XR2, XR3, BPCD[ ].

	true
XR2 =	*BPCD.address*
XR3 =	$\sum_{i=15}^{13} Acc < i >$
Acc =	$Acc << 3$

$\wedge\ NC(BPCD[\,])$

3.3.3 Display Documentation for Software Inspection

Software Inspection

As described in [35], *software inspection* is responsible for systematically examining a program in detail to determine whether or not the program is fit for its intended use. The goal of such an examination is to assess the quality of the software product in question. To make sure that it is precise and complete, the inspection process is systematically prescribed and documented. Complementing *program testing*, detecting code errors, and *formal verification*, which determine mathematical correctness, the *inspection* method plays an important role of improving software quality. This is because in addition to finding errors in code and related documents, it can help inspectors to find problems that are not directly related to theorem proving, model checking and automatic testing. For example, determining whether coding style guidelines are followed.

Why use Display to inspect software?

When examining a lengthy program implemented by others, inspectors desire precise and well-structured documentation. This is because the combination of a large amount of detail with inaccurate or vague descriptions of the structure makes it quite common for serious errors to escape the reviewers' attention. Thus, the program documents should be: (1) well organized in terms of structure, (2) readable, (3) consistent, (4) complete and (5) independent of other documents. The *Display documentation method* was introduced taking into account of above features. Several benefits of using Display to inspect software are now discussed.

First of all, the heart of the *Display Method* is to precisely summarize the effects of a program and its components, so that each Display can be examined and verified without looking at any other Displays. This applies a "divide and conquer" policy which is a key to inspection. Systematic program decomposition makes it practical for inspectors examine small parts of a long program in isolation, while making sure (1) nothing is overlooked and (2) the correctness of all inspected components implies the correctness of the whole.

Secondly, the Displays are the main method used to ensure the correctness and completeness of the code. As discussed in 3.2, tabular expressions used for program specifications in Display can represent the behavior of program segments completely and precisely. It is also easy for inspectors to understand the mathematical notations used in the tables. These are critical for software inspection because the success of the inspection is largely dependent on the understanding of the product and the underlying technologies.

Furthermore, the well-structured and standard mathematical notations which are easy to understand make it possible for people of different backgrounds to be involved in the inspection work efficiently.

3.4 Display Documentation for Program Execution Time Analysis

3.4.1 The relationship between software verification and timing analysis

Program timing analysis is a kind of verification activity. If using one or more variables to represent program execution time, timing analysts can use program tabular expressions to specify program timing properties, and related verification techniques can be used in the analysis. Particularly, when a program is being executed, each calculation (e.g. a multiplication) changes the value of a variable that is already there and also adds to the variable representing calculation time. The amount of time may be constrained rather than fully predictable so this makes the program a non-deterministic program. However, when applying the program function composition as outlined by Mills and those who follow him, the upper and lower bounds for the calculation time may be derived.

3.4.2 Why use Display to analyze WCET?

The Display Method is helpful for representing code segments as functions which is valuable in solving the problems met in timing analysis described in 4.2.

First of all, the tabular function specification used in each Display helps program reviewers understand the function assigned to the program easily and precisely. This is important, because to determine the execution time, analysts have to know the program's function in order to make some of the decisions for the WCET tool's calculation. Also, function tables can specify more information than flow graphs, e.g. initialization information for global variables that do not appear in the specified program segment.

Secondly, the tables are also helpful for flow analysis such as function composition, loop identification, loop bound determination and feasibility analysis. Again, in 4.4.2-3, an example of using a table tool to compose isolate functions, and to determine a loop bound for a program segment containing a while loop is described.

Thirdly, we need to find the execution time of function segments in a long program, i.e. if each segment was assigned one or more individual functions, rather than the execution time between arbitrary start and end statements. In this case we want, the boundaries of our timing analysis match the boundaries of the Displays.

Further, to find the execution time of a particular segment, it is necessary to include all invoked programs' functions. This is true of any verification, and the Display Method provides such information in a precise and concise way that reviewers can use and easily understand.

Chapter 4

General Difficulties in the WCET Analysis

In this chapter, several typical timing analysis difficulties, and WCET analysis problems met in the Reverse-Engineering project are presented, and proposed.

4.1 Overview of WECT analysis difficulties

In modern life, more and more industrial, civilian and military computer systems present timing requirements in their software applications. Correspondingly, the need for effective timing property verification methods and tools is extremely important. However, there is a conspicuous lack of such methods and tools because timing analysis has proved to be an extraordinarily complicated task. The main obstacles encountered in timing verification and inspection are the complexity of software, especially non-terminating concurrent software, like the target control program in the Reverse-Engineering project, and the complexity of such software's possible timing behaviors. These difficulties come from both fundamental, theoretical limitations and practical mechanisms [45].

Program verification is in general *undecidable*. Methods and tools for program verification are generally subject to the problem that the state space size grows exponentially with the size of the program description. Undecidability also causes the

verification methods to be partial or incomplete, and we are forced to use *approximation* techniques, which may take many different forms. Another difficulty is that timing correctness depends largely on logical correctness because it is clear that logical errors can cause timing errors.

The following practical facts also make timing analysis a daunting task [45]:

(1) Some environmental external/internal events affecting the computer system may happen nondeterministically at any time, and this may cause the the associated asynchronous response processes to execute nondeterministically. When processing nondeterministic behavior, the complexity of the logic and timing behaviors of the system increase significantly.

(2) Unless we have sufficient information about the runtime resource, assertions about the timing properties will be inaccurate, and often overly pessimistic. For example, pipeline techniques are used to improve average performance, but they may cause the WCET to be less accurate (more pessimistic) than without pipelining.

(3) To get precise estimation, it is necessary to perform timing analysis at object/assembler code level because the high-level language compiler may optimize the program flow during compilation. Hence, for complex applications, the number of paths that needs to be examined may be increased to the extent that it becomes unmanageable.

(4) Current real-time system design unavoidably applies complex synchronization, priority preemption, interrupts and other mechanisms required to construct concurrent systems. All of them can affect an application's timing properties in subtle ways, and make the timing unpredictable.

As explained by Xu in [45], considering the above difficulties, if the software and its timing behaviors are overly complex, the timing verification and inspection may be practically impossible. Therefore, imposing specific constraints in particular environments to reduce the complexity is necessary. For example, the (early version of) WCET analysis tool that will be discussed in latter sections is restricted by:

- Estimating the WCET of IBM 1800 assembly programs running in a single machine (CPU).

- Estimating the WCET of decomposed program segments with assigned individ-

ual functions and finite executions, rather than of the entire system.

- There are no interfering background activities and the input/output operations e.g. reading a disk takes a fixed execution time.
- The program segment cannot be preempted by other programs.

Associated with the real project, the following sections illustrate several particular problems, met in the WCET analysis of IBM 1800 assembly programs from OPG, and the methods introduced to solve them.

4.2 Decomposing Long Programs

The aim of *program decomposition* is to decompose a long program into small parts and then, provisionally, associate a function with each part [32]. In this way, analyses can be performed individually such that (1) if each part implements its assigned function, the whole program will be correct, and (2) that each part implements its assigned function.

4.2.1 Function Table and Display Construction

In the Reverse-Engineering project, recovering decomposed modules and their interface information from assembly code is time consuming and error prone. This is not only because the assembler language syntax, such as arbitrary jumps and invisible variable types and names, make the program very long and its architecture not obvious (functions are normally implemented by separated blocks), but also because it is hard to extract mathematical functions, which are associated through knowledge from a variety fields, without referring to related specification documents.

In such cases, function tables were found to be helpful in recovering functions from code. In practice, two groups worked individually to construct function tables from the program. One of them constructed tables from source code only, and another group's construction work was allowed to refer to program comments written by the programmers. Then, the two kinds of tables were compared to find differences, and

revised by the two groups working together. It is natural that table construction is started by extracting function units of small program slices, and then functionally composing these smaller functions until the required function is created. Further, the early draft of function tables present "low-level" function behavior, e.g. changes in core storage content and shifting of binary bits. There is no general template to define the function scope, and the scope is normally defined by annotations from the analysts. It requires an iterative process to extract functions that implementers intended. It is also common that comments are incorrect, incomplete or missing, but the point is we are trying to use all of the given valuable information to recover the functions assigned to the code.

Constructed tables and their related code are composed into *displays* as discussed in chapter 3. For instance, in the (display) example in 3.3.2, the function table in the first part of the display is constructed from the code listed in the second part. The tabular expression is much easier for readers to understand the function than reading the code. Compared with other reverse engineering approaches, e.g. [47][48], which first extract assembler code to high-level language representation such as C, the tabular expressions are more concise and abstract. In another words, it is much easer for readers to get the big picture of the program accurately and clearly through reading function tables extracted from code, whether the code is written in assembler or a high level language.

To capture the intent of the original design, it is required to perform functional recovery for all the elements of a given program. For example, again, the program labelled *TRBFF* in the display in 3.3.2 invokes a subroutine labelled *DI2F3* and its display specification is shown below. The function *TRBFF* can not be specified unless we know the function *DI2F3*. Thus, given a (long) program, we first do trials to find some segments which do not branch "far away" to construct unit tables, then combine or nest them into "bigger" tables. For instance, the display of *DI2F3*, shown below, was first created. Then, its functionality was included in one of the header conditions in the function table of *TRBFF*.

1. **Display Specification**
   Name: DI2F3
   External Variables: Acc, XR2, XR3, BPCD[ ].

	true
$XR2 =$	$BPCD.address$
$XR3 =$	$\sum_{i=15}^{13} Acc < i >$
$Acc =$	$Acc << 3$

$\wedge\ NC(BPCD[\,])$

2. **Program**
   Address Label Option TF Operands

```
(35c9)  DI2F3   DC        0
(35ca)          LDX   2   3
(35cb)          LDX   3   0
(35cc)          BSC       -
(35cd)          MDX   3   1
(35ce)          SLA       1
(35cf)          MDX   2   -1
(35d0)          MDX       *-5
(35d1)          LDX   L2  BPCD
(35d3)          BSC   I   DI2F3
```

3. **Specifications of Invoked Programs**
   Null

So far, the above work is manually performed, which is time consuming and error prone. Thus tool aided table construction is desired. For example, the table operation tools discussed in 3.4 and automatic tools explored by other groups to extract arithmetic expressions and pre/postconditions for partial segments, will improve the efficiency and accuracy of the table construction.

4.2.2 Subroutine Invocation Identification

During the program decomposition, subroutines should be identified and their invocation properties, e.g. parameters and other interface information, should be determined. This is because they not only fulfill part of the functionality of the caller program, but also may be called by different modules. Unlike most high-level languages that use specific invocation statements such as Call, subroutine calls in assembler sometimes must be implemented by branch or jump statements. Thus, identification of subroutine invocations should:

- figure out the scope, related variables (e.g. parameters) and behavior of the subroutines,
- specify identified subroutine call properties properly,
- identify subroutine call templates used by programmers.

For example, Figures 4.1 and 4.2 show two of the templates commonly used in the target application (BPC code) in the Reverse-Engineering project. The difference between the two templates is that the first one (BSI-DC-BSC) is used to call a subprogram without arguments, while the second (BSI-DC-BSI) is typically used to call a program with arguments. In the latter case, the parameters passed to the program are specified in the DC area that follows the call (the initial BSI).

4.3 Determining Loop and Loop Properties

4.3.1 Loop Identification

Assembly programs use conditional and unconditional branches to achieve the desired control flow. This often makes the program structure artificially complex. Similarly, the loop as well as the subroutine invocation structures may not be as clear as in high-level language. For example, a FOR loop in the *DI2F3* display in 4.2.1 is not obvious in the code. However, it is not difficult for readers to find a potential loop (between nodes 35cc and 35d0, and exits from 35d1) by viewing its *flow graph* shown in

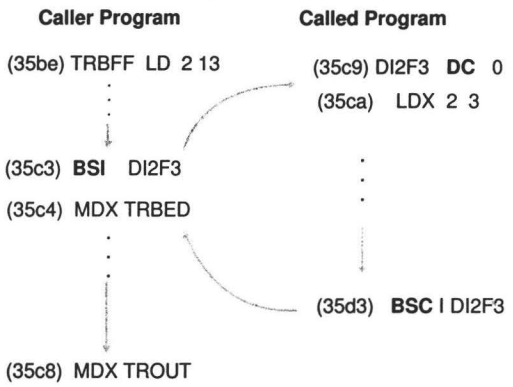


Figure 4.1: BSI-DC-BSC Subprogram Invocation Example

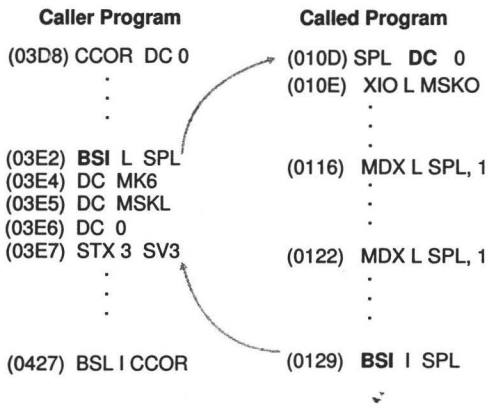


Figure 4.2: BSI-DC-BSI Subprogram Invocation Example

Figure 4.3. In our work, the *control flow graph* (CFG), generated by graph generator (mentioned in 2.2.1) created by other colleagues in the Reverse-Engineering project, plays an important role in helping program reviewers to find loops in the program segments. It is intuitive to use "back" edges or "circle edges" in the flow graph to identify potential loops. It is also true that the visible directed graph is good for representing flow information.

Another important method used to find loops is reading abstracted function specifications to look for the functions that normally are implemented by loop structures.

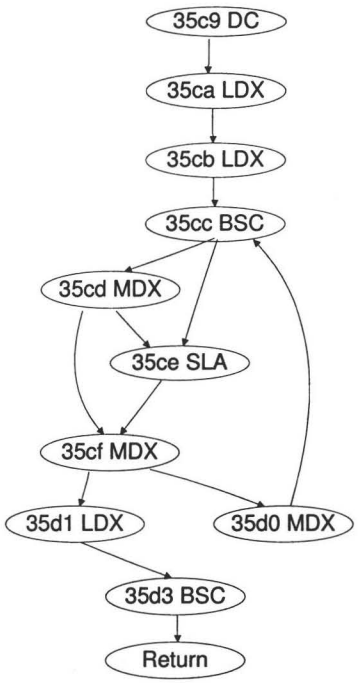


Figure 4.3: Original Control Flow Graph of DI2F3

For instance, the function assigned to program $DI2F3$:

$$XR3 := \sum_{i=15}^{13} Acc < i >$$

would be implemented by a loop by most programmers.

4.3.2 Loop Bound Determination

In 2.2.3, the importance and difficulties of determining the *loop bound* for timing analysis was explained. The most essential fact of the determination is understanding the semantics of the program correctly and completely. Again, in the above example, if we know that the identified loop in $DI2F3$ computes $XR3 := \sum_{i=15}^{13} Acc < i >$, it is obvious that the loop will iterate a fixed number of times, i.e. its loop bound

is 3. In general, this kind of problem is clear to programmers or implementors, and once it is abstracted to a mathematical form as above, it will not be difficult for program reviewers to get the solution. However, it is very hard and complex to develop automated tool to determine the number of times that loops iterate [10]. Thus, *manual annotation* is one way to determine loop bounds in programs. This work can be done either by program implementers or by program reviewers.

Moreover, at McMaster University, researchers have introduced a new way to determine loop bounds based on tabular expressions. Briefly, the programs are first transformed into function tables, and then tables are input into the TTS and loop bounds are calculated by a series of table compositions. Our example shows that some of the loop bounds of programs represented by tables can be determined automatically. An example of using tabular expressions to determine loop bound is given in 4.4.3.

4.3.3 Loop Type Classification

In practice, loops can be used in some *non-functional computations.* A typical case is an *error-handling process.* For instance, as shown below, a $Do/While$ loop is used to prevent illegal user input to a $SQRT$ function which computes square roots for non-negative reals.

$$
\begin{aligned}
&Do\ \{println(\text{“}Please\ input\ x\ >=\ 0:\text{”});\\
&\qquad readln(x);\ \}\\
&While(x < 0);\\
&SQRT(x);
\end{aligned}
$$

It is clear that user inputs are unpredictable, and the task in timing analysis is to determine the time of functional computation e.g. $SQRT$. Therefore, such loop can be assumed to execute once and only once but taking bounded time. In other words, loops in the programs should be categorized into different classes, such as *functional, error-handling* and other *non-functional* loops. The *functional* loops can be divided into *FOR-Loop* and *WHILE-Loop* further, and for such nonfunctional loops, specific constraints can be used to simplify the loop property analysis.

Furthermore, it is important to document those results for further analysis and verification, especially for programs that have complex loop structures. This is because determining loop properties normally is a complex and iterative process, and it largely depends on how well people understand the target program.

4.4 Identifying Infeasible Paths

4.4.1 Infeasible paths caused by MDX statements

As mentioned in 2.2.1, in the Reverse-Engineering project, the program *flow graphs* can be generated automatically by syntactic interpretation. However, the graphs contain a number of *infeasible paths.* For example, in the IBM 1800, the MDX instruction is specified as "MDX can increase or decrease the contents of a register or memory unit. If the result is equal to 0 or has a different sign than the original, the next instruction is skipped." Based on this specification, MDX instructions are interpreted as alternative branch instructions by the flow graph generation tool. But in some cases, programmers use MDX only to change the values of some variables rather than as branch statements because they are sure that the increase or decrease operations will not cause the results to be zero nor to change the sign of the variable. Such infeasible paths result in skipping some instructions and so will not affect the WCET solution. However, while analyzing example slices of the code, we found a number of MDX instructions that make the graphs much more complex than their real flow structure. Thus, the effects cannot be ignored.

Again, to identify the feasibility of program flows, program reviewers should understand the program behavior very well. For example, in the $DI2F3$ segment, there are three MDX statements used in three different forms as:

1. **assignment statement:** (35*cd*) MDX 3 1, which can be interpreted as "variable $XR3$ adds 1 to itself. Then, if its value neither becomes 0 nor changes sign, the next statement (35ce) will be executed, otherwise it will be skipped". However, in this slice, $XR3$'s original value is 0, and it is in the loop whose loop bound is 3, which indicates the maximum value of $XR3$ is 3. Thus, the

above condition will always be true, i.e. the (35ce) will never be skipped. In other words, this MDX can not have an alternative execution, and it is used as an assignment statement only. Correspondingly the CFG of DI2F3 in Figure 4.3 should be refined by removing the infeasible path from (35cd) to (35cf) as shown in Figure 4.4.

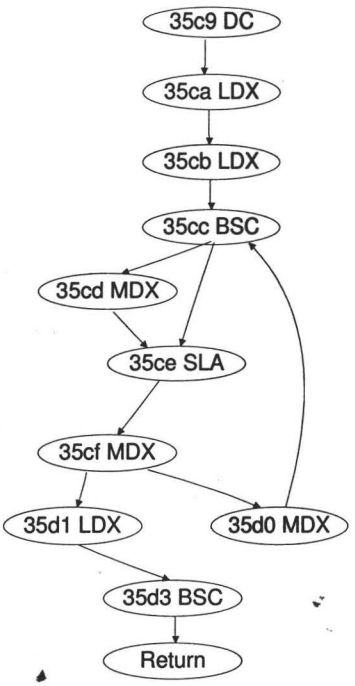


Figure 4.4: Refined Control Flow Graph of DI2F3

2. **alternative branch statement:** $(35cf)$ MDX 2 -1, which can be interpreted as "variable $XR2$ subtracts 1 from itself. Then, if its value becomes 0 or changes sign, jumps to the statement addressed (35d1), otherwise its next statement is executed". In this case, $XR2$ is used as a *loop counter* whose value is 3 originally, and decreases by 1 in each iteration. When its value is decreased to 0, the loop is terminated by jumping to (35d1).

3. **unconditional branch statement:** $(35d0)\ MDX\ *-5$, which can be interpreted as "goto the statement whose memory address is current instruction address minus 5". Hence, this statement is used to implement a loop structure.

Similar to the loop bound determination, it is too hard to develop an automated feasibility determination tool. Our WCET analysis tool helps analysts find the "suspect" nodes e.g. MDX statements, which may cause infeasible paths, then, provides displays to aid analysts in determining whether the path is feasible or not.

4.4.2 Use tables to determine flow feasibility

Function tables can also be used to determine flow feasibility because they represent the related program's behavior precisely. For example, the loop body, containing two sequential *if-then-else* statements, of the program segment shown in Figure 2.1 has the flow shown on the left in Figure 4.5. If we compose function expressions for each

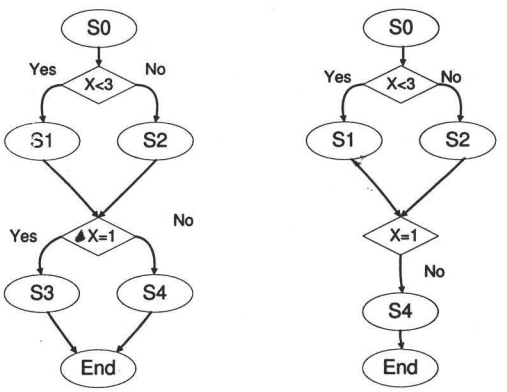


Figure 4.5: Original/Refined Flow Graphs of the Program in Figure 2.1

statement, we get the individual tabular expressions, Table 4.1 and Table 4.2 shown in Figure 4.6. These tables can be composed to form Table 4.3, which indicates the program has the flow shown on the right of Figure 4.5, and can be simplified into Table 4.4.

Table 4.1	$x =$
$x < 3$	$2x$
$3 \leq x$	$x+1$

Table 4,2	$x =$
$x = 1$	$x + 2$
$x \neq 1$	$x + 1$

Table 4.3		$x =$
$x < 3$	$2x = 1$	*false*
	$2x \neq 1$	$2x + 1$
$3 \leq x$	$x+1 = 1$	*false*
	$x+1 \neq 1$	$(x+1) + 1$

Table 4.4	$x =$
$x < 3$	$2x + 1$
$3 \leq x$	$x + 1 + 1$

Figure 4.6: Function Tables of the Program in Figure 2.1

4.4.3 Example of using tables to determine loop bound

For the program in Figure 2.1, associated with the initial loop conditions and Table 4.4, the TTS tool can

- enumerate each case of the legal input iteratively until the termination state. Tables from 4.5-(1) to 4.5-(3) in Figure 4.7 show the results of each iteration step.
- generate a tabular expression for the example program as shown in Table 4.6.

Correspondingly, the loop bound can be determined to be 3.

Table 4.5-(1)	***x* =**	**Terminated**
x = 0	*2x+1*	*False*
x = 1	*2x+1*	*False*
x = 2	*2x+1*	***True***
x = 3	*x+1+1*	***True***

Table 4.5-(2)	***x* =**	**Terminated**
x = 0	*2(2x+1) + 1*	*False*
x = 1	*(2x+1) + 1 + 1*	***True***
x = 2	*2x+1*	*True*
x = 3	*x+1+1*	*True*

Table 4.5-(3)	***x* =**	**Terminated**
x = 0	*((2(2x+1) + 1) + 1) + 1*	***True***
x = 1	*((2x+1) + 1) + 1*	*True*
x = 2	*2x+1*	*True*
x = 3	*(x+1)+1*	*True*

Table 4.6	***x* =**	**Iteration Number**
x = 0	*4 x + 5*	*3*
x = 1	*2 x + 3*	*2*
x = 2	*2 x + 1*	*1*
x = 3	*x + 2*	*1*

Figure 4.7: Use Tables to Determine Loop Bound of the Program in Figure 2.1

4.5 Understanding Program Behaviors

Without access to related documentation, it is very difficult to recover the program's behavior as intended by the original designers. Especially, when studying assembler code, reviewers are almost "blind" to the abstract functions implemented in the program. In our example, we found that the function table is an effective tool for reviewers to collect related data and represent their analysis results. In particular

tables are good for

- interpreting program statements,
- presenting low-level functionalities of program such as variable or state changes caused by program execution,
- presenting abstracted functionality of program in readable mathematical form,
- collecting and supplementing information related to the program behavior's effect on the repository e.g. data dictionary

The following section presents an example of analyzing the small code slice shown in Figure 4.8, from the BPC module.

4.5.1 Program Statement Interpretation

The main tasks of statement interpretation are to determine (1) the initial states or conditions of executing a given program and (2) which variables' values will be changed, and how they will be changed. With reference to Figure 4.8, this phase results in the following descriptions.

```
Address Label Option TF  Operands
(3886)  WU0   LD     2 2
(3887)        SRA      1
(3888)        BSC  L   WU1, E
(388a)        MDX  L   BPCD+2, 2
(388c)        LD       WCMN+1
(388d)        BSI      EMNI
(388f)  WU1   LD     2 6

.....
```

Figure 4.8: Warm Up Example Segment

1. Initially,

   (1)register variable $XR2$ stores the address of data list named *BPCD*.

   (2) *WCMN* and *SMT* also are data lists used in the system.

2. (3886) loads the content stored in the third word of *BPCD* table to *Accumulator*, through indirect addressing by $XR2$.

3. (3887) shifts *Accumulator* right with one bit.

4. (3888) if *Accumulator's* value is not even, goto WU1(388f).

5. (388a) adds 2 to the content stored in the third word of *BPCD* table, and if the value is 0 or there is a change in sign, goto (388d), otherwise goto next statement (388c).

6. (388c) loads the content stored in the second word of *WCMN* to *Accumulator*.

7. (388d) calls *EMNI* subroutine, which is responsible for modifying the values of SMT stack message list, with a function named EMN, based on the value saved in accumulator,

8. (388f) the end of WU0 section

4.5.2 Function Extraction

Based on the above results, a function table can be constructed as shown in Figure 4.9, in which some named variables and data structures, e.g. an array, are introduced. They are used to present changes in state and modification of core storage data. Note that, this kind of table is not sufficient to represent the real physical behavior of the program without additional semantic knowledge.

Name: WU0

External variables: *Acc*, $XR2$, $BPCD[\,], WCMN[\,], SMT[\,]$.

$XR2 = BPCD.address \Longrightarrow$

	$BPCD[2] < 1 >= 0$	$BPCD[2] < 1 >= 1$
$BPCD[2] =$	$BPCD[2] + 2$	$BPCD[2]$
$Acc =$	$Valueof Acc(EMN(WCMN[1]))$	$BPCD[2]$
$SMT[\,] =$	$EMN(WCMN[])$	$SMT[\,]$

$\wedge NC(XR2,\ WCMN[\,],\ BPCD[\,]\ except\ BPCD[2])$

Figure 4.9: Low-level Function Table of Warm Up Segment

4.5.3 Collecting Supplementary Information

After studying the program context in detail, we can deduce the following supplementary facts.

- the third (16 bit) word in the *BPCD* table, $BPCD[2]$, was used as a *flag* word, named *Warm-up Flag* word, in which its first six binary bits represent six *flags* used in the plant warm up process respectively.
- the bit indexed with 1 is used as a *whether-warmed-up-flag* in which :
  (a) the value 0 indicates the plant has not been warmed up.
  (b) the value 1 indicates the plant has been warmed up.
- to determine the value of the *whether-warmed-up-flag*, the programmer used the following steps as shown in Figure 4.10:
  (a) load the *Warm-up Flag* word to *Accumulator*,
  (b) shift right the content in *Accumulator* one bit, which copies the value of the *whether-warmed-up-flag* in the bit 0 of *Accumulator*,

(c) test whether the value of *Accumulator* is even or not. Yes means that the *whether-warmed-up-flag* has a value of 0, otherwise it has the value of 1.

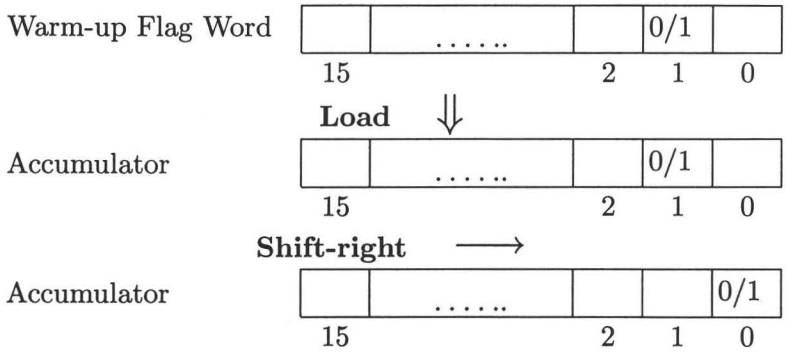


Figure 4.10: Warm-up Flag and Accumulator

- the MDX statement in (388a) is executed with the precondition specified in (3888) which implies that the value of *whether-warmed-up-flag* is 0. Figure 4.11 shows that if *whether-warmed-up-flag* is 0, adding 2 to the *Warm-up Flag* word has the effect of changing the *whether-warmed-up-flag* to 1. It will not cause the word, BPCD[2], to equal zero or change sign. In other words, the MDX works

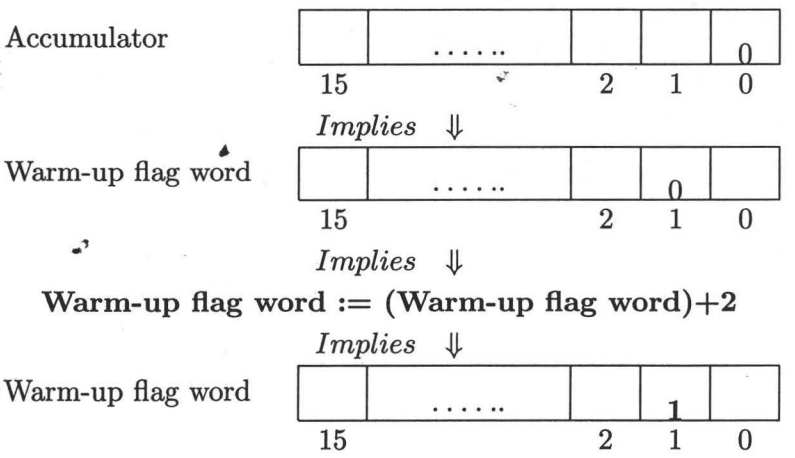


Figure 4.11: Warm-up Flag Setting

as an assignment statement to set *whether-warmed-up-flag* and not branch at this node.

- the second word of *WCMN*, $WCMN[1]$, stores the initial information, *warm-up-turned-on-message*, for invoking subroutine *EMNI*.

4.5.4 Function Table Abstraction

Knowing the above information, the implementer's intended program behavior now can be described as: "If the value of *whether-warmed-up-flag* in the *Warm-up Flag* word (stored in the third word of the *BPCD* data table) is 1, then skip to $WU1$ directly without modifying any variable, otherwise, (1) set *whether-warmed-up-flag* to 1, (2)assign initial data to *Accumulator*, (3) call subroutine $EMNI$ and (4) return to $WU1$ after executing the subroutine". Correspondingly, the previous table in Figure 4.9 can be reconstructed as shown in Figure 4.12.

Name: WU0

External variables: *whether-warmed-up-flag*, *warm_up_turned_on_messgae.*

whether-warmed-up-flag=0	*whether-warmed-up-flag=1*
whether-warmed-up-flag=1 $\wedge$ *EMN(warm-up-turned-on-messgae)*	*NULL*

Figure 4.12: Abstract Function Table of Warm Up Segment

It is important to note that constructing function tables is also helpful for finding other specific features that exist in the applications. For example, we can use tables to help identify *self modifying* code which is possible in assembly programming.

Chapter 5

Extended Display Method for WCET Analysis

This chapter and the next illustrate how the conventional display documentation method is extended to support WCET analysis and related tool development.

5.1 Display Documentation Extension

In chapter 3, the advantages of using precise display documentation methods to analyze program execution time properties were explained. However, when performing timing analysis, such documentation methods should be extended as described below.

5.1.1 Timing-variable

To aid timing analysis, one more variable, T representing the time for a function computation, can be added into each function table in the displays. In particular, to determine the execution time of a given program, T is originally defined as 0, and each calculation (e.g. multiplication) changes its value by adding the time taken by the calculation. The amount of time taken by the whole program is represented in the table, and it may be constrained rather than fully predictable, so this makes the program a non deterministic program with respect to its timing properties. However, when composing program functions, the upper and lower bound for the value of time

on completion will be determined. For example, Figure 5.1 shows the extended (added time variable T in the last row) tabular expression of the example program $DI2F3$ discussed in 4.2.1.

Notes: (1) N denotes the number of possible execution paths of the program (2) $t_{p1}, \ldots, t_{pN}$ denote the execution time for path1, ... , path$N$ respectively.

Name: DI2F3

External Variables: Acc, XR2, XR3, BPCD[ ].

	true
$XR2 =$	$BPCD.address$
$XR3 =$	$\sum_{i=15}^{13} Acc < i >$
$Acc =$	$Acc << 3$
T	$\{t \mid t_{p1}, t_{p2}, \ldots, t_{pN}\}$

$\wedge NC(BPCD[\,])$

Figure 5.1: Timed function table example

In particular, a loop is used to calculate $\sum_{i=15}^{13} Acc < i >$. It iterates three times and contains an alternative branch statement in its loop body as shown in its refined CFG in Figure 4.4. Thus, N is equal to 8 since the program has 2^3 possible execution paths. Correspondingly, the detailed specification of time variable T is shown in Figure 5.2, and the *upper* and *lower bounds* of execution time of the program are $max\{t_{p1}, \ldots, t_{p8}\}$ and $min\{t_{p1}, \ldots, t_{p8}\}$ respectively. Note that $len(subroutine) = 0$ indicates the program does not invoke other programs, and $len(loop) = 1 \wedge loopCount_1 = 3$ indicates that the program contains one and only one loop which iterates 3 times. It is clear that if more branches and loops are used, the number of the possible execution paths will increase exponentially. This makes the determination of the timing lower and upper bounds much more complex and difficult. Hence, it is desired that some execution paths can be ignored if their execution times are not limiting values. For instance, with reference to $DI2F3$'s refined CFG in

Name: timeDI2F3.

External Variables: Acc, *loop*, $loopCount_1$, *subroutine*.

$len(subroutine) = 0 \ \wedge \ len(loop) = 1 \wedge \ loopCount_1 = 3 \Rightarrow$

H1

$\frac{H1 \wedge H2}{G}$	$Acc < 14 >= 0$		$Acc < 14 >= 1$	
	$Acc < 13 >= 0$	$Acc < 13 >= 1$	$Acc < 13 >= 0$	$Acc < 13 >= 1$
$Acc < 15 >= 0$	$T = t_{p1}$	$T = t_{p2}$	$T = t_{p3}$	$T = t_{p4}$
$Acc < 15 >= 1$	$T = t_{p5}$	$T = t_{p6}$	$T = t_{p7}$	$T = t_{p8}$

H2 G

Figure 5.2: Specification of time variable T

Figure 4.4, if it is found that the alternative branch at (35cc) is valid in all iteration scenarios, removing the edge from (35cc) to (35ce) will not affect determining the execution time *upper bound*, and ignoring edges from (35cc) to (35cd) and from (35cd) to (35ce) will not affect the *lower bound* result. Hence, after this adjustment, only one execution path is left. This indicates that it is possible to analyze paths in the graph, in a systematic way, to remove paths that can not affect the minimum/maximum execution time. This practical approach serves to decrease the complexity of the WCET analysis.

5.1.2 Program Flow Graph

An important property of *function tables* is that they define a function that maps from old values of variables to new ones, without considering any intermediary status. In other words, they provide a high level, precise abstraction of the function. However, in timing analysis, some detailed information is required. For example, as discussed in chapter 2, program *flow analysis* is an essential phase of timing analysis. It takes program *flow information* as input, and outputs and then evaluates *flow properties*

and refined *flow representations.* In the Re-Engineering project, as in many other WCET analyses, program flow information is represented in a graphical form called a *flow graph.* Thus, in WCET analysis, the program *flow graph* as well as timing variables must be added to the Displays.

However, the flow graphs may contain redundant flows or may miss some flows, i.e., their correctness and completeness needs to be verified and improved by either implementors or program reviewers. For example, as illustrated in 4.4.1, the original automatically generated flow graph contains an infeasible path, and after related analysis, it can be refined to the graph shown in Figure 4.4.

To retain the original concept of the Display, the generated and refined flow graphs should be added into the fourth section of displays [32]. This section contains a demonstration of the correctness of the display and its components. As defined by the authors, this part could be either a description of the informal reasoning routinely done by a programmer, or a more formal argument. The existence of this additional section would make the reviewer's task simpler – one would not have to invent a "proof", only to check one.

Another important reason for involving graphs in the WCET analysis is because existing graph theories, tools and other techniques can make the timing analysis more efficient. Some of the tasks can be performed automatically or semi-automatically. In the Reverse-Engineering project, a fully automatic tool was developed for generating a GXL format control flow graph. In the graph, each *node* corresponds to a statement in the program, and contains a set of *attributes* used to specify statement information such as: assigned memory location (address), operation code, operands, etc. Further, each *edge* corresponds to an unique flow starting from a "source" node and ending at a "sink" node. Figure 5.3 shows the detailed GXL graph node and edge formats. This kind of graph is used to store related information to support other applications. Examples include: (1) To make the assembly program more readable, an XML application was explored to display related information on web browsers and other popular applications. (2) An expression identification prototype tool was used to extract pre/post-conditions by analyzing the graph. (3) A prototype WCET calculator was used to search the longest (execution time) path in such graph. The

```
< node id = "program ID + statement.address">
    < attr name = "label" >< string > label Name < /string >< /attr >
    < attr name = "address" >< string > memory address < /string >< /attr >
    < attr name = "opcode" >< string > operation code < /string >< /attr >
    < attr name = "binary" >< string > machine code < /string >< /attr >
    < attr name = "rel" >< string > XX < /string >< /attr >
    < attr name = "stno" >< string > line number < /string >< /attr >
    < attr name = "format" >< string > format < /string >< /attr >
    < attr name = "tag" >< string > tag < /string >< /attr >
    < attr name = "operands" >< string > operands < /string >< /attr >
    < attr name = "comment" >< string > comments < /string >< /attr >
< /node >

< edge from = "source address" to = "sink address" >
    < attr name = "indirect" >< bool > True/False < /bool >< /attr >
    < attr name = "backedge" >< bool > True/False < /bool >< /attr >
    < attr name = "backedge" >< real > time < /bool >< /attr >
    < attr name = "backedge" >< int > Execution Num  < /bool >< /attr >
< /edge >
```

Figure 5.3: GXL flow graph node and edge format

GXL graphs also can be generated in a conventional visible form (Figures 4.3 and 4.4) by the tool developed in the Reverse-Engineering project.

5.1.3 Program Flow Property variable

To estimate the execution time of a given program, detailed flow properties such as which subroutines get called, how many times loops iterate, and whether the graph contains infeasible paths, should be specified for flow analysis and further calculation. Considering the general difficulties of flow analysis discussed in chapter 4, and the fact that, in practice, some of the flow property determinations are dependent on manual annotations [10], we believe that it is necessary to include a variable, named fp, which is used to specify detailed program flow properties in the Display. In particular, it is used to determine *loop, subroutine, infeasibility* properties and *verification status.* For example, the flow property variable, *fp*, contains an element, *Loops*, which is used to describe (1) the loop property analysis status by a sub-variable named *lstatus* and

(2) the flow information of all the loops in the program by a vector named *loops*. Each element of the vector corresponds to an identified loop in the target program, and it specifies the following information of the loop.

- loop index, an ID or index number of the identified loop
- loop scope, a list of core address pairs in form of (start_address, end_address) that are used to specify the scopes of the slices that constitute the loop. Note, in assembly programs, it is common that a loop or a subroutine is implemented in several separate slices
- nested loops, a vector used to present the nested loops of the target loop
- loop type, the type of the loop such as: WHILE, FOR and Error-Handling
- loop execution number, the number of times that the loop executes in a specific analysis scenario
- loop bound, the upper bound of the number of times the loop may execute.

In Figure 5.4, detailed variables are illustrated. In our particular case, other than code, we do not have any documentation about the application, and we have no chance to obtain related information from designers or implementers. We have developed an interactive tool to aid program reviewers to determine and annotate flow properties such as *loop bound*, *infeasible paths* and so on, when relevant documentation is not available. It is an iterative phase in which the tool is responsible for providing "basic" information and concepts (such as function specifications and flow graphs) which can help people develop appropriate analysis.

5.1.4 Using extended displays for WCET analysis

Generally, in the WCET analysis, the extended displays are used to provide valuable information, including program behavior (tabular) specification, source code, flow graphs, and specifications of invoked programs during the *flow analysis* and *WCET calculation* phases. Related work includes:

Program Flow Variable: ***fp***			
Element Variable Name		**Variable Type**	**Reference**
Status		String	Represents the flow analysis status for a program, {"Unprocessed", "In process", "Confirmed"}
Loops	*lstatus*	String	Represents the loop property analysis status, {"Undetermined", "Determined", "Confirmed"}
	loops	***Loop*** Vector	List of identified loops
Subs	*sstatus*	String	Represents the subroutine property analysis status, {"Undetermined", "Determined", "Confirmed"}
	subs	***Sub*** Vector	List of identified subroutines
InfPaths	*istatus*	String	Represents the infeasible path analysis status, {"Undetermined", "Determined", "Confirmed"}
	paths	***InfPath*** Vector	List of identified infeasible paths

Loop Property Variable : ***Loop***		
Element Variable Name	**Variable Type**	**Reference**
index	String	ID or index of an identified loop.
scope	String Vector	The scope of code slices that constitute the loop.
nested_loop	***Loop*** Vector	The list of nested loops.
type	Sting	Loop type: {"FOR", "WHILE", "Error Handling", "Undetermined"}
iNum	Int	The number of times the loop executes in a specific analysis scenario
iBound	Int	Upper loop execution bound

Subroutine Property Variable : ***Sub***		
Element Variable Name	**Variable Type**	**Reference**
name	String	The name of an identified subroutine.
scope	String Vector	The scope of code slices that consist the subroutine.
sub-subroourine	***Sub*** Vector	The list of sub-subroutines invoked by the target subroutine.
parameter	Sting Vector	Parameters for invoking the target subroutine.
iNum	Int	The number of times the target subroutine executes in a specific analysis scenario
iBound	Int	Upper subroutine execution bound

Feasibility Property Variable : ***InfPath***		
Element Variable Name	**Variable Type**	**Reference**
start	String	The start address of an identified infeasible path
end	String	The end address of an identified infeasible path
comments	String	Annotation of the analysis solution

Figure 5.4: Program Flow Property Variables

1. **Construct displays for given program from code**. This includes a set of subtasks such as decompose long programs into short ones, identify function blocks, understand and extract functions assigned to the blocks, and compose function tables. In 5.2, these tables will be illustrated in detail. It is also important to note that the set of constructed *displays* will be saved in a knowledge reference library, *repository*, for WCET and other analyses.

2. **Explore display management tool**. The tool is responsible for managing all of the constructed Displays, and making it easy for analysts or other software components to use and modify the related Displays.

3. **Determine flow property variables**. In another point of view, the essential responsibility of a *flow analysis tool* is to determine or aid analysts to determine the values of the flow property variables as discussed in 5.1.3. These variables are added to the conventional Displays so as to present information required in the WCET calculation. In our case, before a semantic analysis tool is successfully explored to provide sufficient information automatically, part of the flow property variable determination work is performed manually using a flow analysis assistant tool, which provides interactive panels for analysts to determine the values of flow property variables, and provides a Display Viewer for analysts to refer to Display documents to make appropriate decisions.

4. **Refine flow graphs**. In this step, the tool facilitates CFG refinement, which includes clustering subroutine graph nodes, removing infeasible paths, adding missed edges and other manipulations. This refinement work is based on the flow properties specified in relevant Displays.

5. **Determine timing variables**. For each timing variable added to the Displays, its value is calculated by summing up the block execution time in the related execution path. Then the maximum value of all timing variables is the WCET of the given program.

5.2 Constructing Displays from Code

The main phases of constructing display documentation are:

- Decomposing a program into a set of modules, where each module encapsulates a number of functions.
- Extracting mathematical functions and representing them in tabular expressions for each module.
- Determining interfaces between function invocations.

Associated with an example from the Reverse-Engineering project, the following sections present the methods used to extract assigned functions and construct Displays for the IBM 1800 assembly programs, especially the construction of function tables from code segments. Note that this work is done manually so far, and related source code, LST files generated by IBM 1800 assembler, GXL flow graph files generated by the flow graph generation tool and visible flow graphs were referred to during the construction of the Displays.

5.2.1 Source Code of the Sample Segment

When perform timing analysis, it is essential to know the behavior of a given program. Especially reverse engineering assembly code, the functions are not obvious and invisible to analysts or program reviewers. To analyze the methods used in the construction work, an IBM 1800 assembly program segment as shown in Figure 5.5 was chosen as the example slice. It performs *turbine output computation* in the BPC (Boiler Pressure Control) module. Our analysis found that this segment:

- performs control variable computations which were assigned particular mathematical functions, and does not branch out to invoke other segments outside the BPC code.
- contains typical program structures such as sequential execution, alternative branch, loop iteration and subroutine invocation.

- uses registers which were previously assigned specific values, which should be determined in other segments.
- contains an infeasible path caused by the flow graph generator.

Note, this segment is a combination of $TRBFB$ (Turbine Feedback) , and $TRBFF$ (Turbine Feedforward) which with the nested $DI2F3$ slices mentioned before in chapters 3 and 4.

Address	Label	Option	TF	Operands
(35b6)	TRBFB	LD	1	41
(35b7)		M	1	42
(35b8)		SLT		2
(35b9)		S	1	43
(35ba)		M	1	44
(35bb)		SLT		15
(35bc)		D	1	45
(35bd)		STO	1	46
(35be)	TRBFF	LD	2	13
(35bf)		BSC	L	TRBFD,E
(35c1)		LD		DIW2
(35c2)		SLA		9
(35c3)		BSI		DI2F3
(35c4)		MDX	3	-1
(35c5)		MDX		TRBFE
(35c6)	TRBFD	SLA		16
(35c7)		STO	1	47
(35c8)		MDX		TROUT
(35c9)	DI2F3	DC		0
(35ca)		LDX	2	3
(35cb)		LDX	3	0
(35cc)		BSC		-
(35cd)		MDX	3	1
(35ce)		SLA		1
(35cf)		MDX	2	-1
(35d0)		MDX		*-5
(35d1)		LDX	L2	BPCD
(35d3)		BSC	I	DI2F3
(35d5)	TRBFE	LD	2	
(35d6)		S	1	
(35d7)		M	1	48
(35d8)		SLA		5
(35d9)		STO	1	
(35da)		MDX		TROUT

Figure 5.5: TRBFB and TRBFF code slice

5.2.2 Tabular Expression of the Sample Segment

Figure 5.6 shows the function table extracted from the program in Figure 5.5, and related notations were explained in 3.3.2.

Name: TRBBF/FF.
External Variables: XR1, XR2, BPCD[ ], GST[ ], DIW2.
Internal Variables: *Exp*, *Acc*.
$XR1 = GST.address \wedge XR2 = BPCD.address \wedge$
$Exp = (((((GST[41] \times GST[42]) << 2) - GST[43]) \times GST[44]) << 15) \div GST[45] \Rightarrow$

	$BPCD[13]$ $< 15 >= 1$	BPCD[13]$< 15 >= 0$	
		$(\sum_{i=9}^{11} DIW2 < i >) > 1$	$(\sum_{i=9}^{11} DIW2 < i >) \leq 1$
$GST[46] =$	Exp	Exp	Exp
$Acc =$	0	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	0
$GST[47] =$	0	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	0
T	–	–	–

$\wedge NC(XR1, XR2, BPCD[\,], DIW2, GST[\,]$ *except* $GST[46]$ *and* $GST[47])$

Figure 5.6: TRBFB and TRBFF slice function table

5.2.3 Function Table Construction Methods

The following illustrates the main steps we used in constructing program function tables.

(1) interpret program statements - to understand each statement in context and represent each statement in pseudo code (natural language). These representations are much easier for program reviewers to understand and remember than the

assembly statements. For instance, the first two statements that have the memory addresses (35b6) labelled "TRBFB" and (35b7) in Figure 5.5 are interpreted below.

> (35b6): $Acc := mem(XR1 + 41)$; load the content of a memory word, whose address is the sum of the value stored in register $XR1$ and an offset value (41), to the register *Accumulator*.
>
> (35b7): $Acc := Acc \times mem(XR1+42)$; replace the *Accumulator*'s value with the product of its original value multiplied by the content of the memory word whose address is the sum of the value stored in register $XR1$ and an offset value (42).

(2) determine the *initial conditions* for program segment - from other program segments, using a guess/verify method.

It is clear that knowing the *initial conditions* is essential to determine the behavior of each statement. As for the example in (1), it is impossible to determine the values of each operation without knowing the content saved in register $XR1$. However, the initial value of $XR1$ for the example segment is assigned in another segment. This makes it very hard to determine its value because $XR1$ is one of the registers most frequently used in all segments, and its value can be changed in any segment. In principle, its value only can be determined by tracing back to find the last statement which assigns its value, but it is not practical because of the complexity of control flow in assembly programs.

We applied a *guess and verify approach* to determine those initial conditions not specified in the given segment. For example, when determining the initial value of $XR1$ in the example segment, we guessed its value based on facts such as:

- the most common value used in other segments
- implementers' annotations written in comments

and then verified the assumption by studying its context in the program and by referring to comments. In particular, $XR1$ was assumed to have the value of the start address of the *data table* named GST, the label of the data table slice. It was verified based on:

- the address of GST was assigned to $XR1$ at the beginning of BPC module
- some segments use $XR1$ to store some temporary values during their processing and restore its value to GST address before their termination
- on the assumption, all of the variables specified by $XR1$ and its offsets are valid the descriptions of the comments.

(3) determine used *variables* and their modifications - from comments, other program segments, and initial conditions. In particular, determine what variable memory locations are used in this segment, and how/if they are changed.

To determine the used variables, program reviewers first should name and categorize the variables. After studying a set of program segments, we classified variables into:

- *address label*: used to represent a memory address for a memory unit which may store either data or a binary statement
- *general variable*: assigned an individual address and a "label" which can be used as the variable name
- *data table*: assigned a unique labelled start address and offsets for each element. Thus the data table can be described as an array, named by the *start label* and each of its elements can be specified by the "index", the value of the *offset*
- *registers*: used to perform arithmetic and address manipulations. Note, such variables can be named as relevant register names

Secondly, two kind of statements can indicate which variables are used: (a) *load* statements with the prefix of *LD* and (b) *arithmetic manipulation* statements such as A(add), S(subtract), M(multiply), etc. Normally, their operands specify the memory locations, variables, used for the manipulations.

Thirdly, to determine which variable values are modified, program reviewers should pay attention to MDX and STO statements which are used to modify storage unit values and save the content of the *Accumulator* to particular storage units

respectively. Similarly, their operands specify the variables whose value will be modified. Note, some of the determinations can be supported by the comments provided by the implementers.

Moreover, identified variables are categorized into *internal* and *external* variables as shown in previous function tables. In particular, variables that (a) were used to process intermediate data during the program execution, and (b) do not affect the execution of other programs, and (c) were not defined by other programs are classified as internal variables. Otherwise, they are defined as external variables. Normally, the *external variables* are used as inputs, outputs and (subroutine) parameters that should have been identified.

(4) extract expressions assigned to a series of statements. Knowing which variables have been modified, the next step is to determine how they were modified. Thus, previous operations related to the variable can be collected to construct arithmetic expressions. For example, in Figure 5.5, the eighth statement (35bd) is used to store the value obtained by operations formed through its previous seven statements between (35b6) and (35bd). Associated with the initial condition, $XR1 = address.GST$, and related variables, the above eight statements can be represented by the following expressions:

$$(a) Acc := \frac{((((GST[41] \times GST[42]) << 2) - GST[43]) \times GST[44]) << 15}{GST[45]}$$

$$(b) GST[46] := Acc$$

These expressions are used as elements of functions in the program segment, and to compose related tabular expressions. For instance, the above expression is defined as Exp in the function table shown in Figure 5.6.

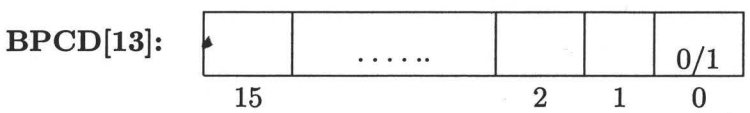
(5) determine subtasks or subroutines. To recover the decomposition structure defined by the implementors, it is important to identify the implementers' templates used to invoke subroutines that implement individual function as mentioned in 4.2.2. For slices that have those features, it is useful to regard each of them as an individual function, which may be accessed by one or more other segments, and to construct individual tables which may be used by other tables. For instance, the

subroutine named $DI2F3$ in the example segment can be represented as the function table in Figure 5.1.

(6) determine the program flow structure. As well as subroutine determination, branch and loop structures should be determined. In the case study, program *flow graph* plays an important role to aid analysts in finding such structures, especially in identifying *loops* which are not obvious when reading the assembly code as explained in 4.3.1.

Moreover, branches in the flow graph indicate the *header conditions* of the function tables. In practice, we construct function tables by both reading the code and referring to flow graphs. We also compare table and graph structures to determine infeasible paths in the graph. For instance, the infeasible path from (35cd) to (35cf) discussed in 4.4.1, also can be determined by comparing the structures of the table condition headers and the flow graph.

(7) determine *strategies* used by implementers, and any *programming templates* they may have used. When implementing the software, programmers typically applied some *strategies* and particular *programming templates* to make their products efficient and to deal with specific hardware features. This knowledge is not clear to others unless it is documented. For instance, several alternative flags used to describe plant status are represented in different bits in a machine word, named *flag word* shown below.



The 1st bit in the word $BPCD[13]$ is a flag used to specify whether the BPC has been initialized (flag value 1) or not (flag value 0). To determine whether the plant has been initialized, instead of directly checking the value of the bit in the specific memory location, the flag word is loaded into the Accumulator first, and then the Accumulator is checked to see whether it is odd or even which indicates the least significant bit is 1 or 0 respectively. Note, in the example, statements (35be) and (35bf) implement the above operations, and for bits other than the least significant bit, after been loaded into Accumulator, they are shifted right into the least significant bit so that they can be tested to determine if they are odd or even.

The above strategies are influenced by the properties of the assembler and by efficiency considerations. For instance, as shown in Figure 4.11, if we know the 2nd bit in the *Warm-up flag word* is 0, adding 2 to the flag word using the MDX statement will set the bit to 1 without changing the value of any other bits. Note, in such cases, the MDX is used as an assignment statement without any possibility of an alternative branch.

These templates were widely used in the applications, so program reviewers need to read context and comments to figure out the usage of the flags assigned in the word, otherwise, it is hard to recover the implementers's intended behavior of the program.

(8) combine the expressions into functions assigned to programs, represent functions in tables. To share programs for different modules, in general, some functions were implemented as *subroutines* which have specific features as discussed in (5). They can be identified and separated from other programs. During the function table construction, firstly, all the *subroutines* are composed into individual tables. Then, they are used as "elements" to construct function tables for the caller programs. For instance, the table in Figure 5.1 represents the behavior of *DI2F3*, and one of its responsibilities is computing $XR3 = \sum_{i=15}^{13} Acc < i >$. Associated with the caller program *TRBFB/FF*, it was found that *DI2F3* performs plant status checking and the above computation is used as a *branch condition* for further output calculation. The result can then be used as a header condition of *TRBFB/FF*'s function table as shown in Figure 5.6.

However, some functions are not easy to separate from other program segments because assembly programs allow "goto" statements to branch to arbitrary statements, which makes the structure complex and vague. In such cases, program reviewers can extract expressions for code blocks, separated by *labels*, and then, compose such expressions into "bigger" functions associated with comments provided by programmers. Correspondingly, small tables for each block are constructed and composed into bigger tables. For example, the function table of *TRBFB/FF* as shown in Figure 5.6 was composed from two individual blocks, *TRBFB* and *TRBFF*, which are used to compute *turbine feedback delta* stored in *GST[46]* and *turbine feedforward delta* stored in

GST[47] respectively. They are combined into a function named *TRBFB/FF* because it is part of the "bigger" function, *turbine output*, separated from other segments by the programmer using distinct separation comments.

(9) determine supplement table notations. Assuming conventional predicate logic and arithmetic operator notations, some assembler operator notations should be defined. This is because the function table extracted from code needs to represent detailed bit manipulations in order to represent the appropriate abstract function (as design specification). For example, a memory bit *shift operation* is widely used in assembly programs. They may be used as divide/multiply, clear memory or flag shift operations. As defined in 3.3.2, they can be used to represent related functions as shown in Figure 5.6.

In assembly programs, variable names and types are not as obvious as in high-level languages. Hence, to make the program function table more readable, it is necessary to use some high-level structure to represent a program's variables and behavior. For instance, using an *array* to represent *data table*, and using *V.address* to denote the *memory address* of variable *V*.

(10) determine function table construction rules. Parnas and his colleagues defined more than 10 forms of function tables such as *normal tables* and *inverted tables*. They are used to represent functions with different properties. In practice, we found that during different analysis and function extracting phases, different forms of tables may be appropriate. For example, as shown in Figure 5.1 and Figure 5.6, to represent changes to variables, the variables concerned are listed in the *left header* in the table. The *branch conditions* are specified in the *top header*, and the related mathematical expressions are specified in the *Grids*. This structure is very clear for determining the modifications of each target variable. Also, variables in the *left header* can be listed in the order of modification sequence (time), which indicates the data flow. Further, the *predicate table*, as illustrated in 4.5.4, is good at representing program invocation and other behavior.

Intuitively, *normal* and *inverted* tables are good at representing functions with complex flow structures and multi-conditional branches. Thus, the above tables can be translated and composed into *normal* or *inverted* tables that may contain multi

headers and nested tables, using the principles and tools introduced by McMaster researchers.

5.3 An example of extended timing analysis display

The following is the extended display constructed from the example program segment in Figure 5.5. It is important to note that it is extended for WCET analysis, and it combines the results obtained through the above analysis.

1. Program Specification

Name: TRBBF/FF.

External Variables: XR1, XR2, BPCD[ ], GST[ ], DIW2.

Internal Variables: *Exp*, *Acc*.

$XR1 = GST.address \wedge XR2 = BPCD.address \wedge$

$Exp = (((((GST[41] \times GST[42]) << 2) - GST[43]) \times GST[44]) << 15) \div GST[45] \Rightarrow$

	$BPCD[13]$ $< 15 >= 1$	BPCD[13]$< 15 >= 0$	
		$(\sum_{i=9}^{11} DIW2 < i >) > 1$	$(\sum_{i=9}^{11} DIW2 < i >) \leq 1$
$GST[46] =$	Exp	Exp	Exp
$Acc =$	0	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	0
$GST[47] =$	0	$((BPCD[9] - GST[27]) \times GST[48]) << 5$	0
T	t_{p1}	$t_{p2} + time(DI2F3)$	$t_{p3} + time(DI2F3)$

$\wedge NC(XR1, XR2, BPCD[\,], DIW2, GST[\,]$ *except* $GST[46]$ *and* $GST[47])$

2. Program Source Code

See complete source code in Figure 5.5.

3. Specification of Invoked Program

Name: DI2F3

External Variables: Acc, XR2, XR3, BPCD[ ].

	true
$XR2 =$	$BPCD.address$
$XR3 =$	$\sum_{i=15}^{13} Acc < i >$
$Acc =$	$Acc << 3$
T	$\{t \mid t_{p1}, t_{p2}, \ldots, t_{p8}\}$

$\wedge NC(BPCD[\,])$

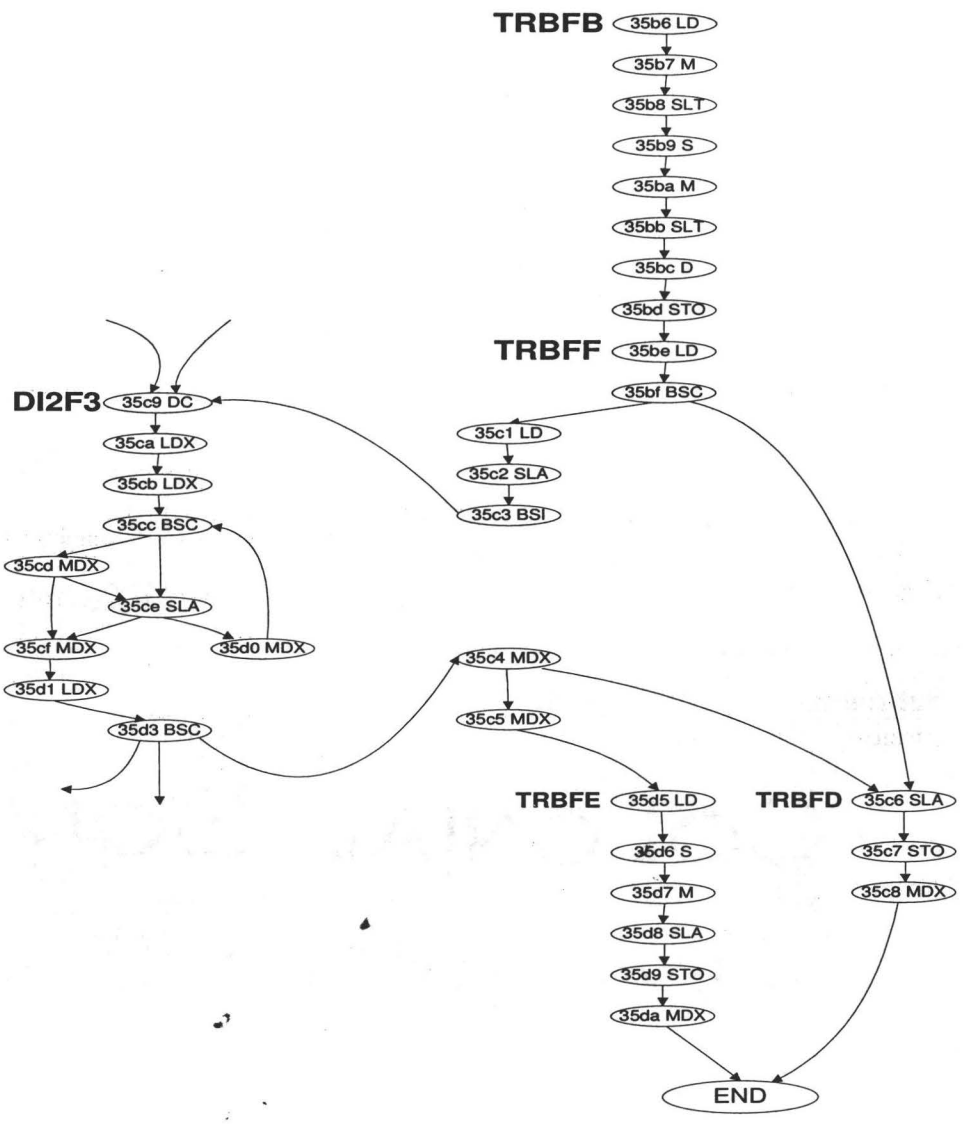
4. Supplemental Information

- **program flow property variable** fp

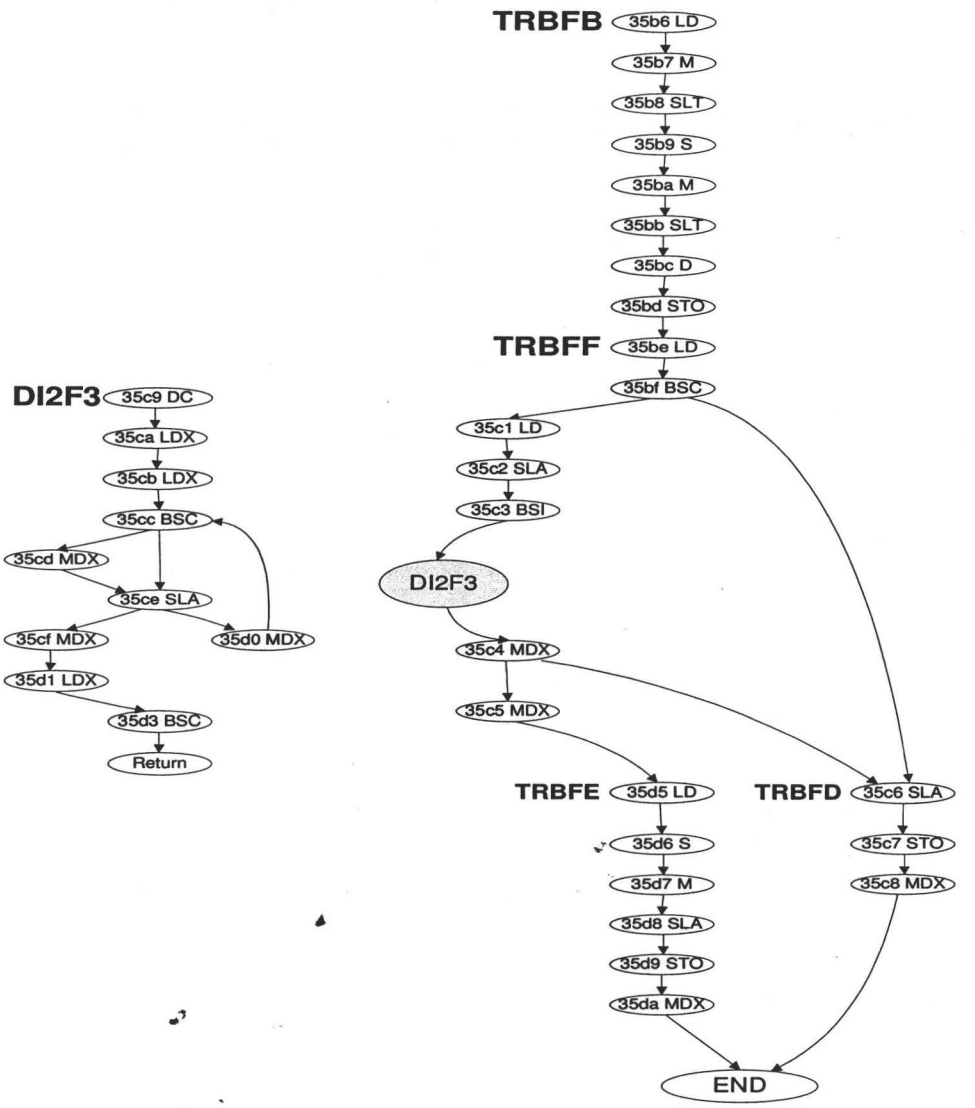
Program Flow variable : ***fp_TRBFB***			
Element Variable Name		**Value**	**Reference**
Status		"Confirmed"	Flow Analysis of program TRBFB is accomplished.
Loops	*lstatus*	"Confirmed"	Loop property analysis is accomplished.
	loops	***Null***	There is No loop structure in program TRBFB
Subss	*sstatus*	"Confirmed"	Subroutine property analysis is accomplished.
	subs	***subs[0]***	There exists a subroutine presented in variable ***subs[0]***.
InfPaths	*istatus*	"Confirmed"	Feasibility property analysis is accomplished.
	paths	***Null***	There is no infeasible path in program TRBFB.

Subroutine Property Variable : ***subs[0]***		
Element Variable Name	**Value**	**Reference**
Name	DI2F3	The name of the subroutine.
Scope	(35c9, 35d3)	Subroutine DI2F3 starts from 35c9 and ends at 35d3
sub-subroourine	***Null***	DI2F3 does not invoke any other programs.
parameter	Acc	DI2F3 takes the value in accumulator as parameter .
iNum	1	DI2F3 is called by TRBFB once.
iBound	1	DI2F3 is called by TRBFB once most.

- **original flow graph**



- **refined flow graph**



manipulations: clustered "DI2F3" subroutine node.

graph refinement: removed infeasible edge "from 35cd to 35cf"

Chapter 6

The WCET Analysis Tool

This chapter describes how a WCET Analysis Tool was developed to help program reviewers in the Reverse-Engineering project to estimate the upper execution time bound of IBM 1800 programs, based on precise program documentation, *Displays*, that aid users of the tool in performing WCET analysis.

6.1 The WCET analysis tool overview

The WCET Analysis Tool (WAT) is designed to compute the *worst case execution time* of given programs written in IBM 1800 assembler language in the Reverse-Engineering project. The first version of the tool was developed for the program reviewers and timing analysts to determine timing constraints of the target programs in the project. These constraints will be used to specify the timing requirements in the reverse engineered high-level requirements.

6.1.1 Overview of the WAT tool and WAT Architecture

To support different activities in the complete reverse engineering process, a *tool suite* is planned and its architecture is shown in Figure 1.1. In particular, the *Timing Analysis Tool* (TAT) is responsible for determining the timing constraints in the assembler program, and the architecture shows how the TAT fits into the suite.

The WAT is one of the components of TAT, which is designed to determine the *upper bound* of the execution time of the target programs. Corresponding to the three main phases, *flow analysis*, *low-level analysis* and *WCET calculation*, of WCET analysis, the tool contains three main sub-tools:

- flow analysis tool
- timing graph generator
- WCET calculator

and each of them consists of a set of components that will be illustrated later.

Considering the difficulties discussed in early chapters, the WAT is designed as an interactive (semiautomatic) tool, in which part of the information is obtained from users' interventions and some manipulation decisions are made by the user. For example, as discussed in 4.4.1, when it cannot determine whether a particular MDX is a branch or just an assignment statement, the flow analysis tool displays related information and prompts the user to decide the MDX's flow property.

The WAT provides *Display* documentation in which program behavior, program decomposition, variable value modification/changes and other valuable information are precisely specified. Hence, unlike other WCET tools [8][44] which ask the program designer or implementers to specify such information during the timing analysis phases, our tool takes advantage of: (1) facts concluded systematically from group rather than from individual opinions, (2) detailed information extracted from the code by tools, and (3) provides a mechanism for the system designer, implementer, inspector, program reviewer and others to integrate the timing analysis work effectively to obtain more accurate solutions.

A *Display Manager* is added to the WAT to manage program *Displays* constructed by program designers or reviewers, and to make *extended Displays* for the timing analysis. It is also designed to make it easy for users to search and view Displays when they need reference information during their analysis phases. Figure 6.1 shows the architecture of the WAT.

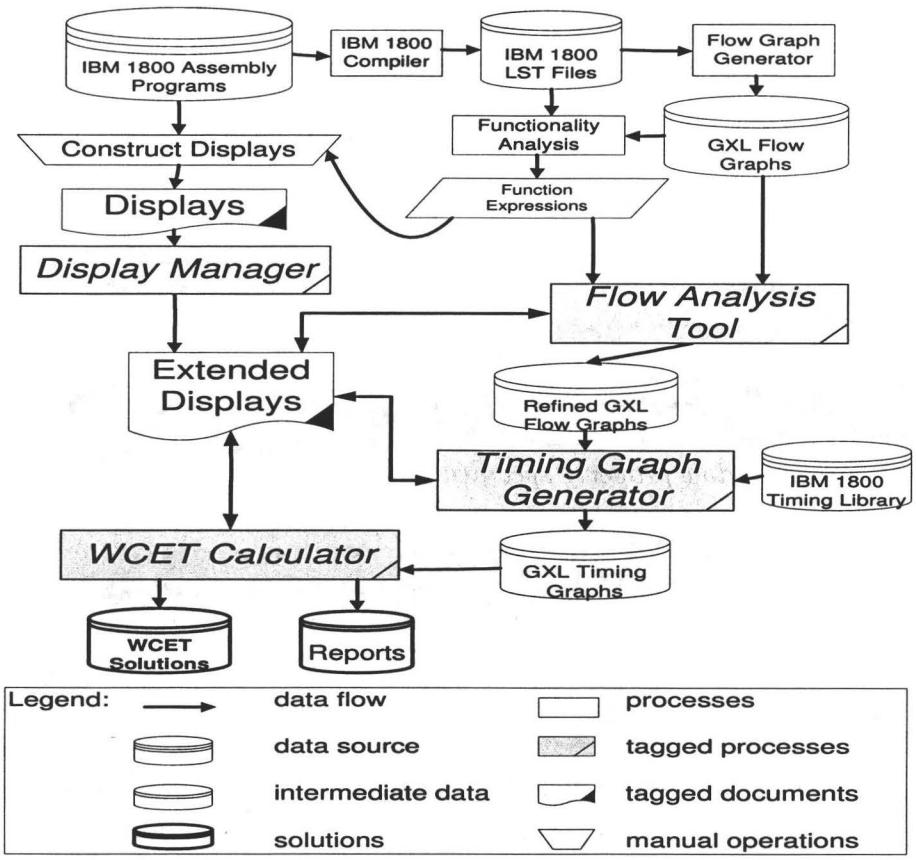


Figure 6.1: WCET Tool Architecture

WAT inputs:

1. *program control flow graph* (CFG) generated by Graph Generation and Analysis Tools;

2. *program flow information* extracted by automatic tool or specified by analysts' intervention;

3. *atomic instruction timing* and other *timing data* obtained from Assembly Representation Library & Emulators;

4. *program behavior/function specifications* documented as *displays*, which include

results extracted from Functionality Analysis & Design Recovery Tools, and program behavior tabular expressions;

5. *other information* extracted from Semantic Analysis Tools or Semantic Analysis Library.

WAT outputs:

1. *refined program control flow graph* in which sub-graph of subroutines are separated and infeasible paths are removed;
2. *program flow property specifications* which describe identified subroutines, loops, infeasible paths and other flow properties used for WCET calculation;
3. *timing graphs* which are assigned statement or basic block execution time information;
4. the *WCET solution*;
5. *report information* including timing information appended to displays, records of users' annotations and etc.

Detailed descriptions of the tool elements are illustrated in latter sections, and note that, so far, program displays are constructed manually.

6.1.2 The WCET tool environment

The WAT is implemented in Java 2 Platform Standard Edition (J2SE), version 1.5, which can run under Unix/Linux/MS-Windows operating systems on either individual PCs or intranet terminals. The required information, listed in section 6.1.1, provided by other groups is stored in GXL(Graph eXchange Language), HTML or XML script formats, which have been provided proper interfaces in Java environments, as well as J2SE 1.5 contains XML parser applications which can be used to process the above script data. Moreover, the outputs of the WAT are also in the above formats, thus all the components in the tool suite shown in Figure 1.1 can communicate easily and efficiently.

6.2 WCET Analysis Tool Description

6.2.1 Sub-tool Functional Overview

In this section, the required functions for each sub-tool are described, and related Java classes are listed. Note that detailed class descriptions for each class including secrets, responsibilities, an assumption list and access function table are presented in the Appendix.

Flow Analysis Tool

Flow Analysis is responsible for determining the program's flow information, also called flow properties, such as information concerning possible ways the program can execute, which functions get called, and how many times a loop iterates. Thus, in the WAT, the *flow analysis tool* shall,

- perform program *flow graph* manipulations including:
  - (a) read provided CFG data,
  - (b) generate a sub-graph for the identified program function block,
  - (c) present graph items such as graph nodes and edges,
  - (d) modify and refine the CFG based on identified flow properties.
- determine and present *subroutine* invocation properties including:
  - (a) subroutine scope,
  - (b) invocation conditions e.g. the value of parameters and the number of programs that call the subprogram,
  - (c) nested sub-subroutines, i.e., other programs invoked by the called programs.
- determine and present *loop* iteration properties including:
  - (a) loop scope and type, e.g. FOR or WHILE loop,
  - (b) loop bound, i.e., the upper bound of the number of loop iterations,

(c) nested loops, i.e., the loop contains more loop structures within its body.

- determine and present execution *path feasibility* properties, e.g., identify the infeasible paths caused by *MDX* statement interpretations.
- refine the original CFG based on the identified flow properties.

FlowGraph, Sub, Loop, Path and *MDX* classes fulfill the above tasks. Moreover, the Flow Analysis Tool also shall:

- combine identified individual flow properties,
- record analysis status, save (intermediate) results and display appropriate prompts for related operations or events,
- provide user interactive interfaces for user to make related decisions and annotations,
- output analysis results including refined CFG and flow property variables.

Thus, *SubAna, LoopAna, PathAna* and *FlowAnalysis* classes are designed to achieve above tasks. They also play a role in controlling the communications between different components. For example, one of *SubAna*'s responsibilities is to take the subroutine properties determined by *Sub* to call the method in *FlowGraph* that refines the CFG.

Timing Graph Generator

The *Timing Graph Generator* is designed to generate a *timing graph* by assigning timing information associated with flow properties in the refined CFG. Related timing information, including atomic statement execution time, timing mechanisms used in the program, and statement (sequence) emulation or simulation results, are stored in a *Timing Library.* Note that some of this information can be concluded from hardware manuals provided by vendors, and some can be obtained through dynamic simulations. In the first version of WAT, with the assumption that the program being analyzed can not be interrupted by others, only the average statement execution time specified in IBM 1800 manual as shown in Appendix A-4 is considered. However, the

timing graph generator should be extendable for processing other timing information. Moreover, program flow properties are mainly used to determine the specific statements that are executed and how many times each is executed. Thus, in particular, the Generator shall

- specify the atomic statement execution time for each statement in every scenario,
- specify hardware features that affect program execution time,
- cluster sequential statements into "basic blocks", in which only the "start node" has one or more incoming branches and only the "end node" has one or more outgoing branches, and all other nodes inside the block only have a single entrance and a single exit,
- generate the timing graph through assigning the following information to each graph edge:
    (a) the (average) execution time of its source node/block,
    (b) the difference caused by any hardware effects,
    (c) the number of times the edge would be passed.

Related classes include *TimeTable, HDFeature* and *TimingGraph.*

WCET Calculator

Given the timing graph, the *WCET Calculator* searches for a path, called the *longest path*, that takes the longest execution time in the graph, and determines the worst case execution time of a program. Particularly, the Calculator shall

- determine how to use the timing information of invoked subroutine/s,
- create the graph data structure to facilitate searching for the longest path,
- search for the longest path in the timing graph,

- determine the type, such as functional computation path, error-handling path or other types, of the longest path that was found,
- present and store *WCET* analysis solutions including:

  (a) the worst case execution time,

  (b) the path that determines the WCET,

  (c) information that is missing for programs that we do not have sufficient flow/timing information.

ProSub, Graph, LongestPath and *wcet* classes are designed to achieve the above functions, and class *Calculator* controls the above modules.

It is necessary to note that, firstly, to find the longest path of a program, it is necessary to know the timing information of each subroutine, and we also need to make correct decisions of how to use the timing information to calculate the WCET of the target program. This is because, normally, the execution time of a subroutine is not unique. The WCET calculator provides several methods for users to determine such timing information:

1. assign the WCET of each subroutine to be the execution time of the subroutine node;
2. replace subroutine nodes with their timing graphs in the target graph for the longest path search;
3. process some parts of subroutines by method 1 (or assign other identified values), and process others by method 2;
4. make assumptions that "remove" paths that will not affect the WCET solution.
5. use the simulation solutions for those subroutines do not have generated graphs.

It is clear that using method 1 normally result in a pessimistic estimate, and using method 2 will increase the complexity exponentially when the analyzed program invokes many subroutines or the subroutines have complex flow structure. In other words, these methods are useful in calculating the WCET for programs that have a

simple structure. Otherwise, method 3 is practical, but it requires that users make appropriate decisions for each subroutine. Method 4 is used to decrease the complexity of a timing graph. For example, assume subroutine SUB has a WCET. If there exists a path that does not invoke SUB and it takes longer time than some paths that do, those paths can be ignored because they do not affect the WCET solution. Method 5 can be used in the case when the execution time of subroutines cannot be determined or analyzed.

Secondly, it is possible that the longest path found is used to process some task, e.g. error-handling as discussed in 4.3.3, that is not the functional computation considered by the analyst. Then, the calculator should tag that path, look for the second longest path, and repeat such work until the longest path does perform the required functional computation. To simplify such work, the Calculator is designed to search for the longest path in the scope indicated by the *function table* in the display.

Furthermore, the WCET calculator is not designed as a "one-click-tool", with which users can click a button to get the final results. On the contrary, users are required to make some decisions for the tool, e.g. choose the method/s to determine the timing of subroutines, to compare solutions obtained through different methods, and to adjust operations and operands to obtain the desired solution. Therefore, the WCET calculation manipulations such as subroutine timing determination, longest path search and other tasks may be performed iteratively.

Display Manager

The Display Manager shall,

- manage the constructed display information (data base).
- invoke "Display Viewer" to show the requested displays.
- generate *extended Displays* through supplementing the conventional display with the program flow graph, the flow property variable, and the timing variable.
- modify extended Displays.

Related classes in the WAT are *Display* and *ExtDisplay*. The former fulfills the first two tasks, and the later fulfills the remaining tasks associated with the access functions in classes Display and FlowGraph.

6.2.2 Tool structure

Our goal is to construct the WAT for ease of extension and contraction [38]. Thus, principles [37] such as *design for change* [1] and *information hiding* [29] are applied in the tool implementation. To avoid having it process data in a Data-Transforming chain, the WAT sub-tools are decomposed into a set of components (Java classes) considering their responsibilities and changeability, which are organized in a hierarchical structure based on invocation relations as shown in Figure 6.2.

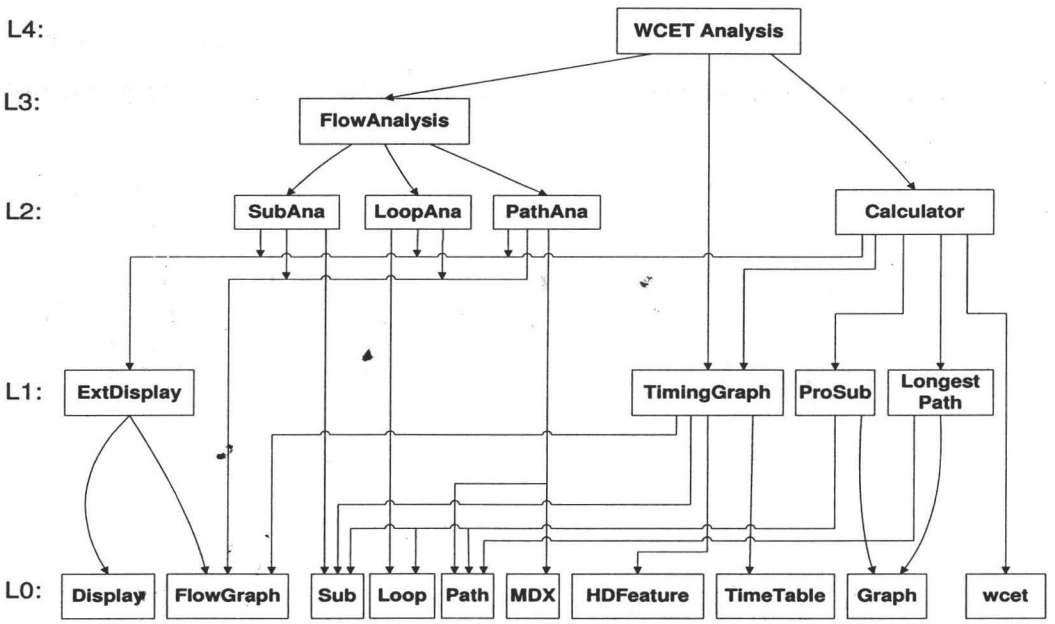


Figure 6.2: WAT Class Invocation Hierarchy

First of all, classes are defined based on required "services" (see detailed examples

of requirements specifications in Appendix A), and categorized into five levels by the following rules:

1. Level 0 is the set of classes that invoke no programs in other classes,
2. Level $i(0 < i < 5)$ is the set of classes that invoke at least one program on level $i-1$ and no program at a level higher than $i-1$.

For instance, *ExtDisplay* class in level 1 which is designed as a *Display Manager* calls methods in *Display* and *FlowGarph* classes in lower level, level 0, to make *extended Displays*. Also, its methods may be called by the programs in a *SubAna* class in higher level to provide some information.

Secondly, class definition should take into account items that are likely to change. For example:

- if the flow graph generator is improved and has the capability to fix the MDX infeasible path problem, class MDX will be redundant,
- flow graphs in GXL/XML format are replaced by a relational data base format,
- program Displays are updated from static picture image into dynamic HTML for web viewing.

Note that these changes may be caused by: a change in user's requirements, compatibility with other modules, or other unanticipated reasons. To handle these changes, related services should be separated from others. For instance, as shown in Figure 6.2, MDX is an individual class. If the MDX infeasible path problem is solved by another program, MDX can be removed without affecting other classes except for the PathAna class.

Furthermore, we tried to define changeable aspects as "secrets" hidden from other programs, and designed interfaces for them to make them usable by other programs. For example, the data structure of class *FlowGraph* in level 0 is defined as a secret, and a set of access functions were implemented to read the provided flow graph data from a disk file, to provide user programs in higher levels e.g. *TimingGraph*, the required information. In this way, a data format change, e.g. from GXL/MXL script to relational data base, of *FlowGraph* will not affect programs at all in higher levels.

Space and other considerations make it hard to present detailed design for all of the classes in Figure 6.2. Thus, in Appendix B , the main responsibilities of each class are introduced briefly and several classes are highlighted to illustrate the strategies used in the system design.

6.3 Using the WCET Analysis Tool

In this section, associated with a couple of examples, we illustrate how the WAT works and how it aids the user in performing WCET analysis.

6.3.1 Start WCET Analysis

As explained in earlier sections, the WAT is designed to process programs that have been analyzed and for which Displays have been constructed. These programs are categorized into "unprocessed" and "in process" groups. Their Displays and other data files are also categorized this way. The WAT provides a *New* operation to start analysis for programs in the former group, and *Open* operation to continue the analysis for those in the latter group. For instance, Figure 6.3 shows the panel of the WAT to open an analysis case for program TRBFB, in a set of provided programs.

6.3.2 View Program Display

In the WAT, a *Display Viewer* is responsible for showing *Display* images to users. They can invoke the Viewer either through the main File menu, or click Display Viewer buttons that appear in panels in the different analysis phases. Figure 6.4 shows an example of the Display of program TRBFB using the Viewer. Note that current displays are represented in PNG image format, and later version of the WAT will provide HTML format displays so they can be viewed in web viewers.

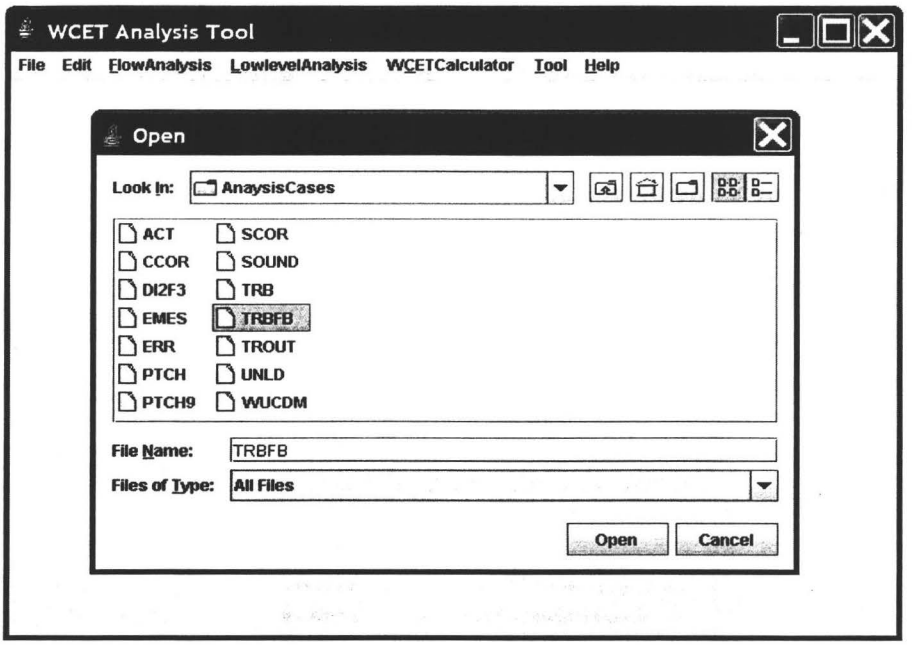


Figure 6.3: Open a WCET Analysis Case

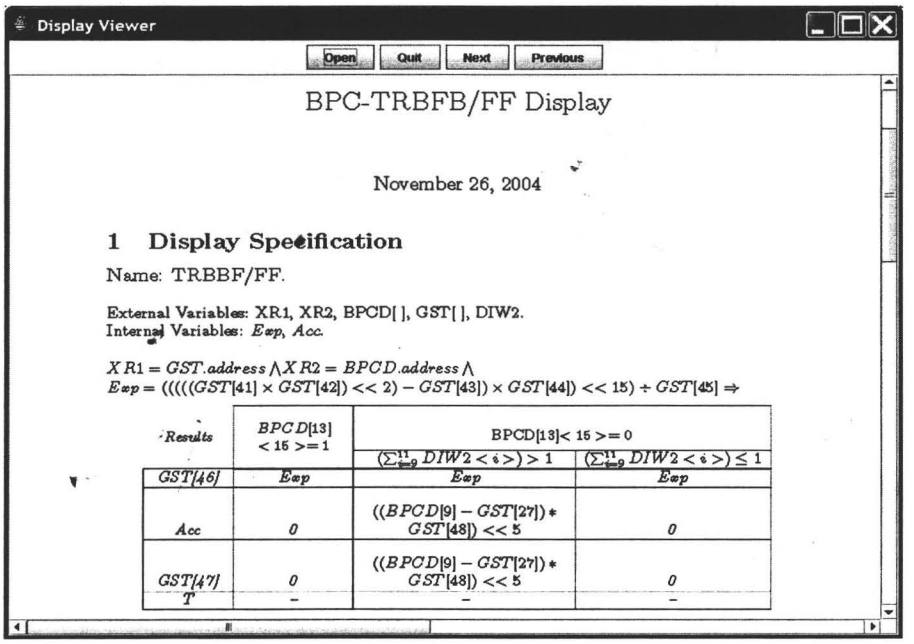


Figure 6.4: Display Viewer

6.3.3 Program Flow Property Analysis

As shown in Figure 6.5, the program flow analysis panel presents entire and individual analysis status information of a given program, e.g. TRBFB including subroutine, loop, and infeasible path analyses. It also provides a set of operation buttons to invoke the above analysis tasks, refine the provided CFG or start the next analysis phase, and determine low-level effects. Particularly, associated with the sample program TRBFB and its subroutine DI2F3, the following subsections illustrate how the WAT aids the user in fulfilling these subtasks of flow analysis.

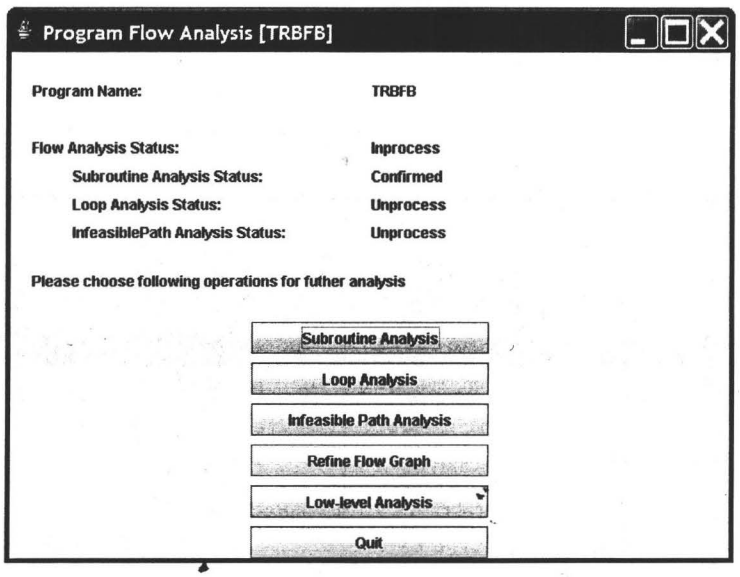


Figure 6.5: Program Flow Analysis Panel

Determine Subroutine Properties

Figure 6.6 shows an example of a subroutine analysis panel on the top, and a subroutine data panel on the bottom applied to program TRBFB. The former panel shows that TRBFB contains a subroutine DI2F3, and provides a set of operations such as add, remove, view, modify and save subroutine items. Further, users can click the "View Display" button to invoke the Display Viewer to view related information, or

enter the next analysis phase through clicking "Low-level Analysis" button. The sec-

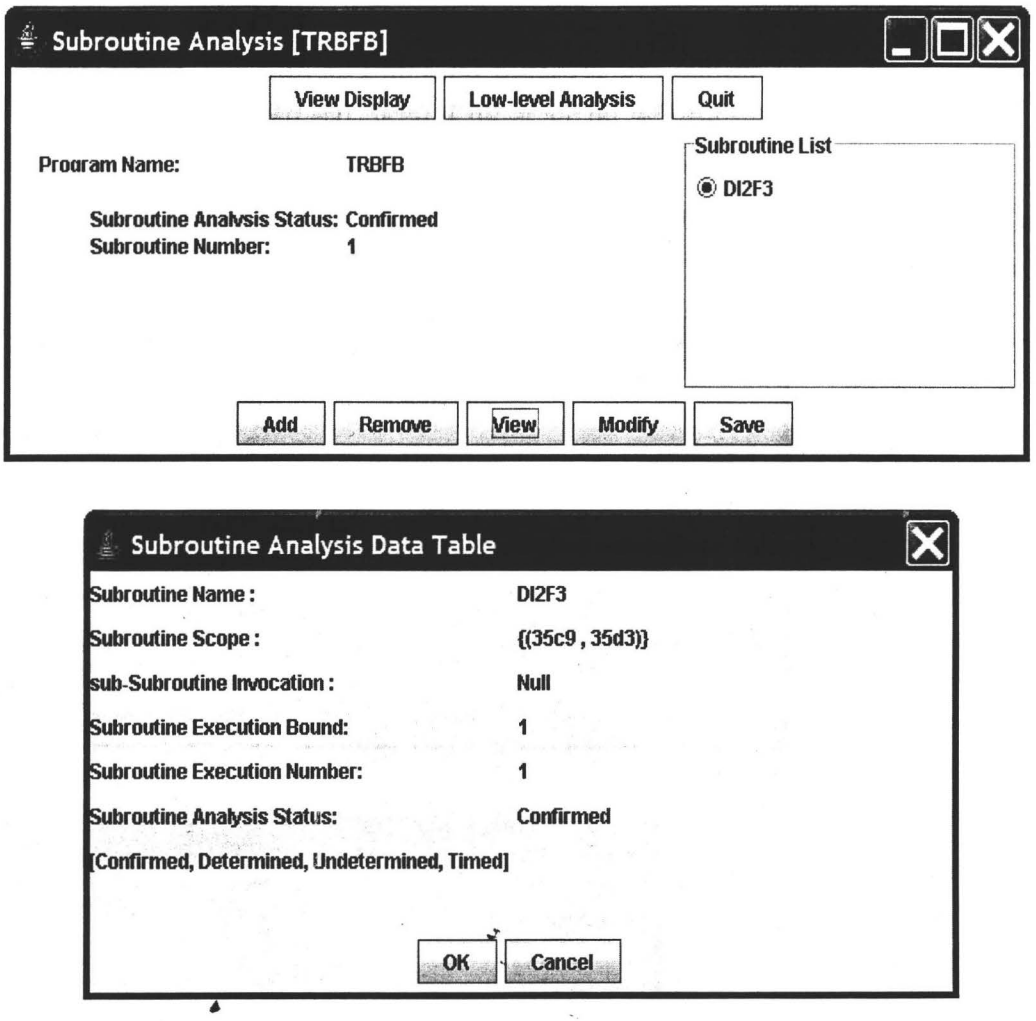


Figure 6.6: Subroutine Analysis Panels

ond panel shows the subroutine (DI2F3) invocation information for program TRBFB, including: name, scope, list of invoked sub-subroutines, the (maximum) number of the subroutine invoked and other data. It is important to note that the WAT is not responsible for identifying subroutines. Instead, it takes the subroutine decomposition from provided displays, and describes the scope of subroutines defined in the Displays.

Determine Loop Properties

Figure 6.7 shows a *loop* analysis solution panel, on the top, and a loop data panel, at the bottom, applied to program DI2F3. The former shows a loop indexed as No.1 in the program, and the latter is an interactive panel used by the user to specify loop properties including the index, scope, list of nested loop/s, loop type, upper execution bound and other information applicable to the identified loop.

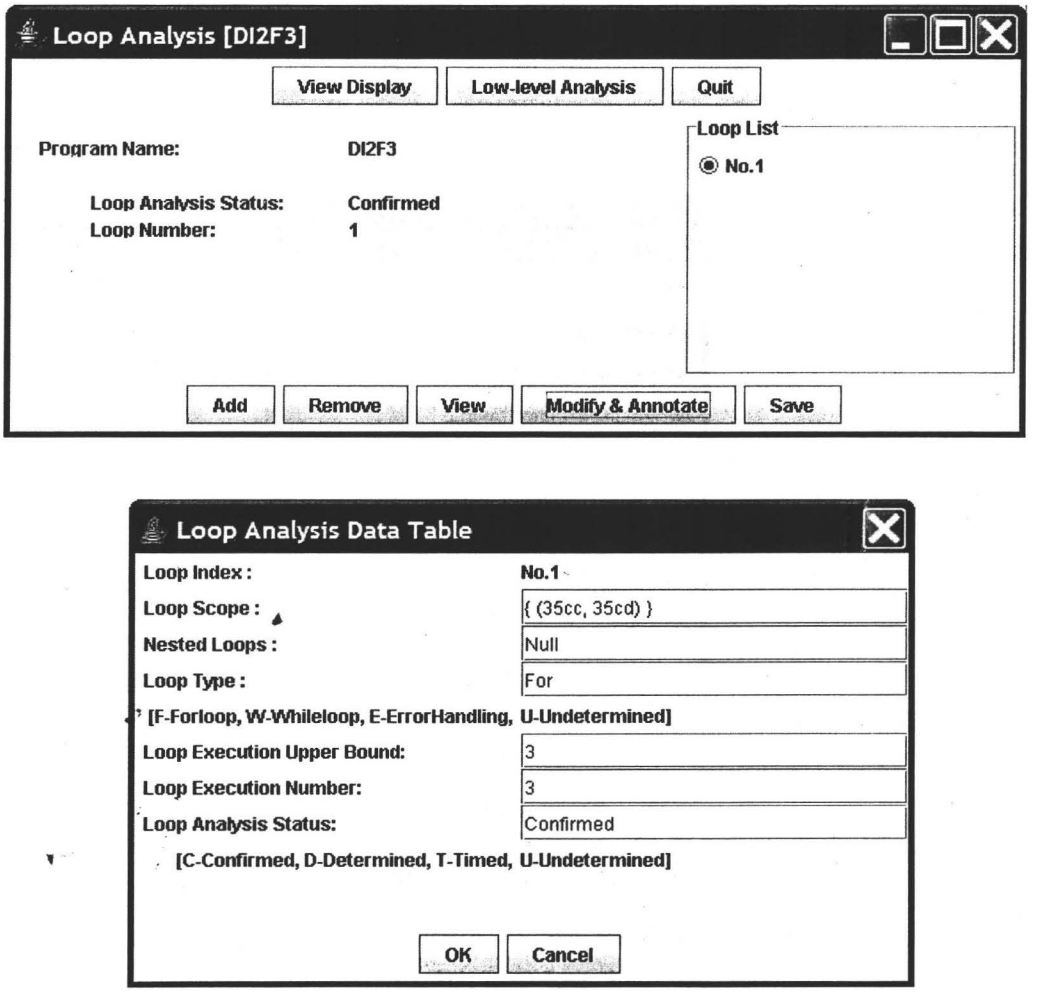


Figure 6.7: Loop Analysis Panels

Unlike subroutines, loop structures should be identified by the user of the tool. Normally the user can find loops by:

- looking for "backward" edges, or flow "circles" in CFGs in extended displays,
- looking for functions specified in function tables, which may be implemented in iteration structures, e.g. in DI2F3 the formula $\sum Acc < i >_{i=3}^{N}$ indicates a FOR loop,
- reading program source code in a Display to find loops implemented with conditional or unconditional jumps.

Determine Infeasible Paths

As shown in the top panel of Figure 6.8, the current WAT fulfills infeasible path identification by:

- processing MDX statements,
- providing interactive panels for the user to annotate the infeasible path manually.

For MDX statement processing, the tool first collects all the MDX statements in a given program. Next, it sets the status of all the MDXes interpreted as "alternative branch" to *suspect* (infeasible) source nodes as shown in the left bottom panel in Figure 6.8. Then, the user is required to determine the correctness of interpreting such suspect nodes as branch nodes. It is clear that incorrect interpretations will cause infeasible paths. For instance, as shown in the right bottom panel in Figure 6.8, the first MDX statement is determined as a "sequential" statement, which indicates that, in the original CFG, one of the node's outgoing edges, from 35cd to 35cf, is infeasible.

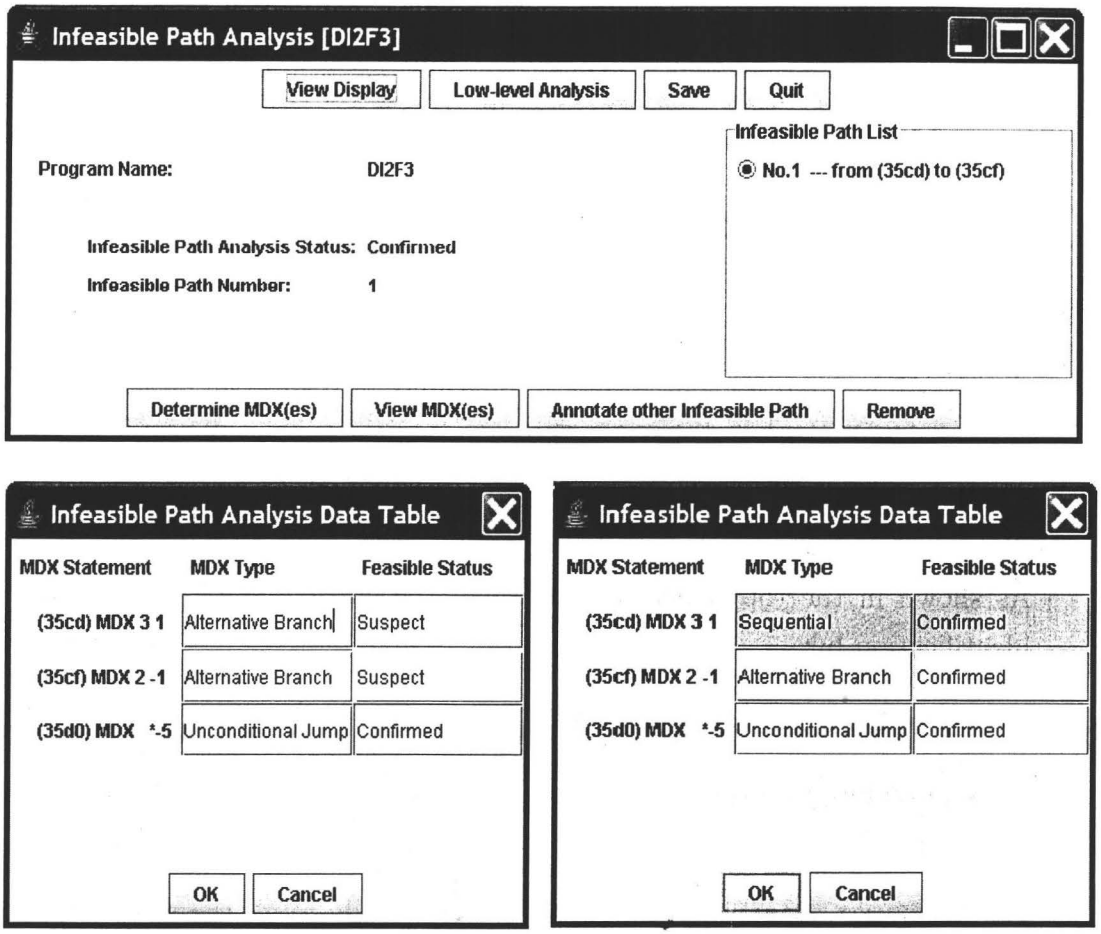


Figure 6.8: Infeasible Path Analysis Panels

Furthermore, for infeasible paths caused by other reasons, the WAT provides interactive panels for users to annotate them including their positions and related analysis comments. Figure 6.9 shows a panel for the user to annotate identified infeasible paths, and a panel to present information about an identified infeasible path caused by MDX interpretation.

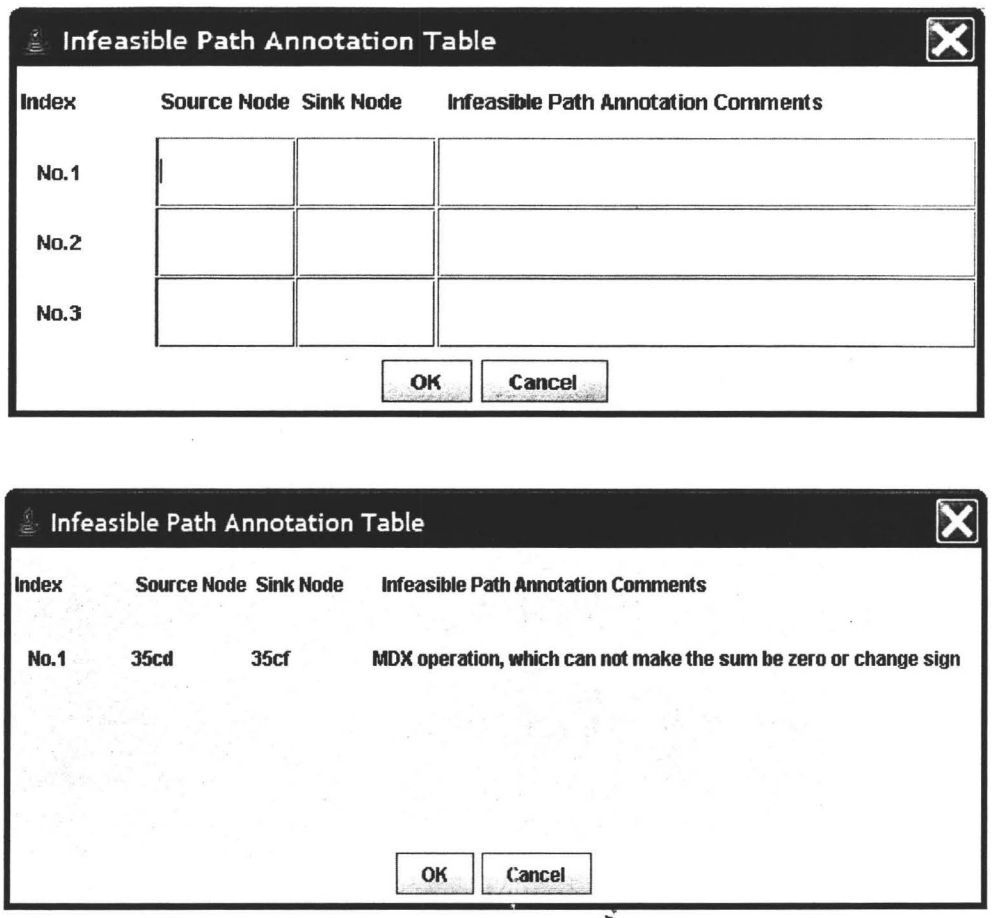


Figure 6.9: Infeasible Path Annotation Panels

6.3.4 Generate Timing Graph

To generate a timing graph, the WAT tool first reads the original CFG information of a target program into memory variables. Figure 6.10 shows the CFG of program TRBFB loaded by a *graph reader*.

```
E:\WINDOWS\System32\cmd.exe
C:\DataDriver\Java\OPGTOOL\Figures>call java BPCSubgraph
bpcSub35b6Con.gxl
(35b6)  TRBFB LD    1 41      (35b6)-->(35b7)
(35b7)        M     1 42      (35b7)-->(35b8)
(35b8)        SLT     2       (35b8)-->(35b9)
(35b9)        S     1 43      (35b9)-->(35ba)
(35ba)        M     1 44      (35ba)-->(35bb)
(35bb)        SLT     15      (35bb)-->(35bc)
(35bc)        D     1 45      (35bc)-->(35bd)
(35bd)        STO   1 46      (35bd)-->(35be)
(35be)  TRBFF LD    2 13      (35be)-->(35bf)
(35bf)        BSC   L  TRBFD,E (35bf)-->(35c6)(35bf)-->(35c1)
(35c1)        LD      DIW2    (35c1)-->(35c2)
(35c2)        SLA     9       (35c2)-->(35c3)
(35c3)        BSI     DI2F3   (35c3)-->(35ca)
(35c4)        MDX   3 -1      (35c4)-->(35c5)(35c4)-->(35c6)
(35c5)        MDX     TRBFE   (35c5)-->(35d5)
(35c6)  TRBFD SLA     16      (35c6)-->(35c7)
(35c7)        STO   1 47      (35c7)-->(35c8)
(35c8)        MDX     TROUT   (35c8)-->(35db)
(35ca)        LDX   2 3       (35ca)-->(35cb)
(35cb)        LDX   3 0       (35cb)-->(35cc)
(35cc)        BSC     -       (35cc)-->(35cd)(35cc)-->(35ce)
(35cd)        MDX   3 1       (35cd)-->(35ce)(35cd)-->(35cf)
(35ce)        SLA     1       (35ce)-->(35cf)
(35cf)        MDX   2 -1      (35cf)-->(35d0)(35cf)-->(35d1)
(35d0)        MDX     *-5     (35d0)-->(35cc)
(35d1)        LDX   L2 BPCD   (35d1)-->(35d3)
(35d3)        BSC   I  DI2F3  (35d3)-->(35c4)
(35d5)  TRBFE LD    2 9       (35d5)-->(35d6)
(35d6)        S     1 27      (35d6)-->(35d7)
(35d7)        M     1 48      (35d7)-->(35d8)
(35d8)        SLA     5       (35d8)-->(35d9)
(35d9)        STO   1 47      (35d9)-->(35da)
(35da)        MDX     TROUT   (35da)-->(35db)
```

Figure 6.10: TRBFB Original CFG Script Representation

Note that the CFG is in script format and includes:

- graph node IDs (statement core addresses), which are represented as "(XXXX)",
- statement, including label, operate code, format, tag and operands,
- node outgoing edges, represented as "(source node)−− >(sink node)".

Figure 6.11 shows the timing graph of TRBFB generated by the WAT, in which the CFG was refined and the statement execution time was specified. Particularly, compared with Figure 6.10,

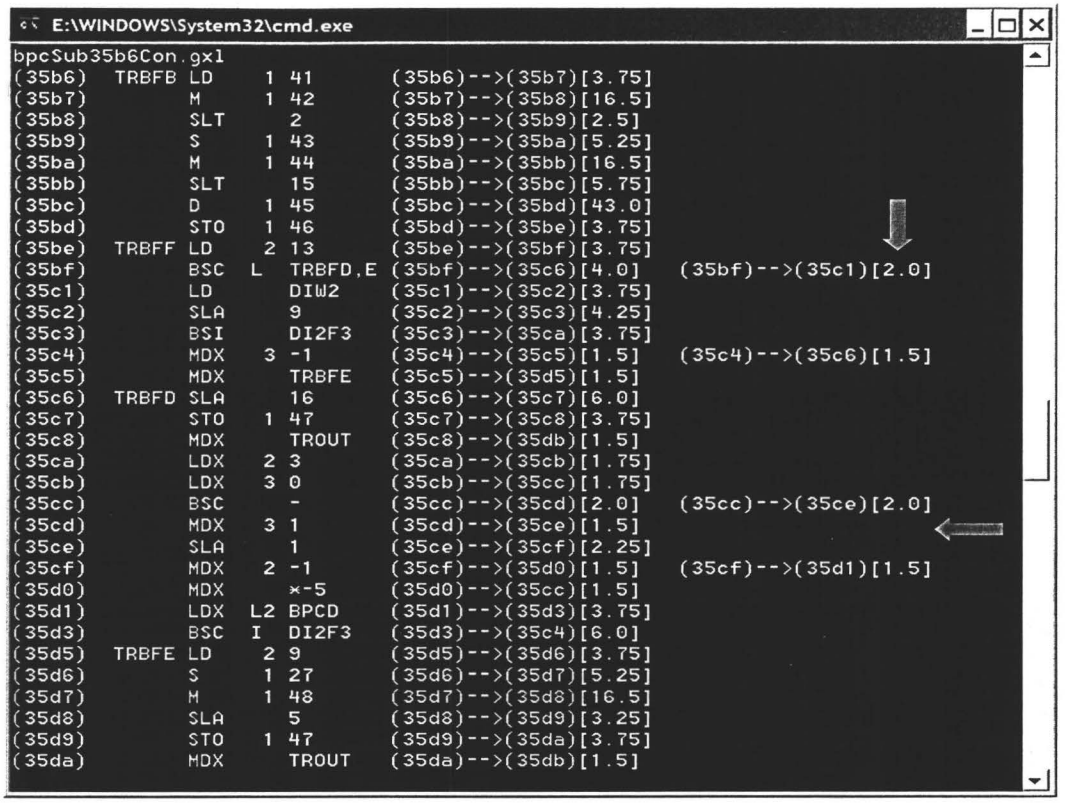


Figure 6.11: TRBFB Timing Graph Script Representation

- an infeasible path, from node 35cd to 35cf, indicated by the horizontal arrow, was removed,

- statement execution time, represented as "$[t_s]$", was appended to outgoing edges for each graph node as indicated by the vertical arrow.

Note that, the generated *timing graphs* are still stored in GXL script data files, and they can be used to generate visible graphs by the graph process tools developed by other groups in the Reverse-Engineering project.

6.3.5 WCET Calculation

Figure 6.12 shows a WCET output of program TRBFB. It was calculated based on the program flow property and timing results obtained through 6.3.3 and 6.3.4. In particular, in this example, as illustrated in the Figure:

- a flow graph for TRBFB was combined with a sub-graph of its subroutine, DI2F3,
- longest path search was performed within the combined flow graph,
- the found longest path that was found contains a loop executed three times,
- worst case execution time is 189.5 milliseconds.

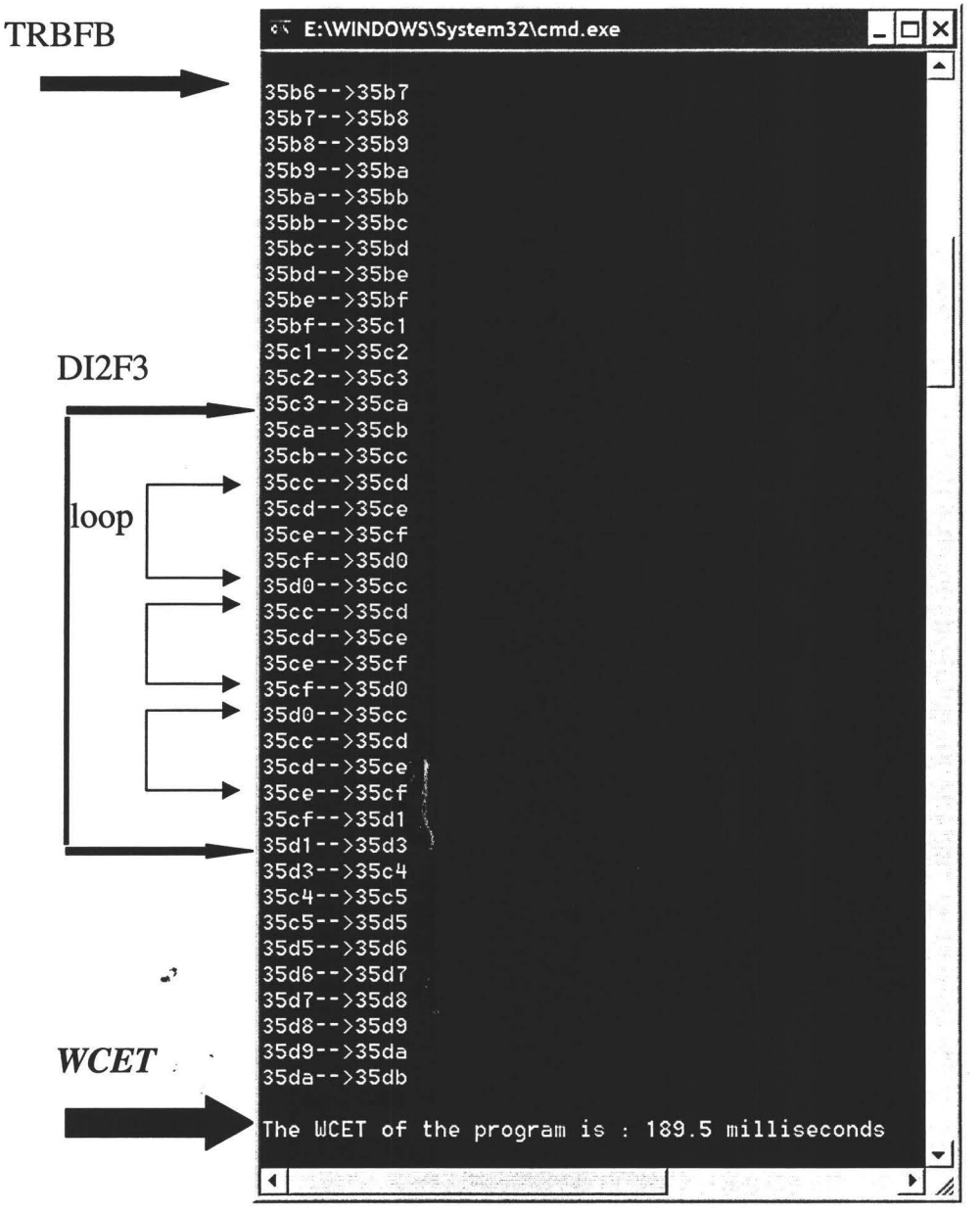


Figure 6.12: WCET Output of TRBFB

Chapter 7

Conclusions and Future Work

The major contribution of this thesis is that it introduces a documentation driven worst-case execution time analysis method. It applies an extended precise Display documentation technique to aid WCET analysis, and makes the analysis more systematic and accurate. A tool to support this analysis is included in the thesis. The tool uses a precise documentation approach as a way to overcome the general difficulties met in conventional WCET analysis. Advantages of this approach are:

- Displays provide timing analysts precise and complete behavior specifications of the analyzed program for determining its flow properties,
- tabular notations used in Displays make the specifications more readable and make it possible for both program implementers and reviewers to describe the program in the same manner. In other words, the above WCET analysis method can be used in either software development or reverse engineering,
- it is practical to develop tools to store and provide required information to aid program execution time analysis.

Our conclusions are supported by experience gained when the above method was applied to develop tools to analyze the IBM 1800 applications.

Future work may explore tools to automatically generate program function tables from the code. This is important because manually constructing function tables

for long programs is time consuming and error prone. Another interesting direction for future work would be to explore ways of developing tools to solve the problems discussed in chapter 4, e.g., developing tools to determine loop bound and infeasible path automatically.

Bibliography

[1] Britton, K.H., Parker, R.A., and Parnas, D.L., A Procedure for Designing Abstract Interfaces for Device Interface Modules, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6, pp. 295–313.

[2] Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., and Lisper, B., Worst-Case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System, 2nd Workshop on Real-Time Tools (RTTOOLS 2002), Copenhagen, Denmark, August 1, 2002. URL: *http* : *//www.mrtc.mdh.se/publications.*

[3] Chikofsky, E. and Cross, J., Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1): 13–17, January 1990.

[4] Corti, M., Brega, R. and Gross, T., Approximation of Worst-Case Execution Time for Preemptive Multitasking System, Proceedings of the ACM SIGPLAN 2000 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2000), June 18, 2000, Vancouver B.C., Canada. (c) Springer-Verlag, Lecture Notes on Computer Science, Vol 1985, 2000. URL: *http* : *//www.lst.inf.ethz.ch/research/publications/.*

[5] Engblom, J. and Ermedahl, A., Modeling Complex Flows for Worst-Case Execution Time Analysis in Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00), Nov. 2000.

[6] Engblom, J., Ermedahl,A., Pipeline Timing Analysis Using a Trace driven Simulator, Extended version of a paper presented at the 6th Internation Conference

on Real-Time Computing Systems and Applications (RTCSA '99), Hong Kong, December 1999.

[7] Engblom, J., Ermedahl,A., and F. Stappert, Comparing Different Worst-Case Execution Time Analysis Methods, Presented at the Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium RTSS 2000), Orlando, Florida, USA, December 2000.

[8] Engblom, J., Ermedahl,A., Sjodin, M., Worst-Case Execution-Time Analysis for Embedded Real-Time Systems, accepted for publication in the STTT (Software Tools for Technology Transfer) special issue on ASTEC, 2001.

[9] Engblom, J., et al, Towards Industry Strength Worst-Case Execution Time Analysis, ASTEC Technical Report 99/02 and DoCS Technical report 99-109, April 1999.

[10] Ermedahl, A., Engblom, J., Stappert, F., A Unified Information Language for WCET Analysis, WCET Workshop, Wien, June 18 2002. URL: *http : //user.it.uu.se/ jakob/*.

[11] Ermedahl, A., Engblom, J., Stappert, F., Clustered Calculation of Worst-Case Execution Times, (To appear in) Proceedings of the 6th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES 2003), San Jose, California, USA, Oct 30th to Nov 1st, 2003. URL: *http : //user.it.uu.se/ jakob/*.

[12] Everets, K., Assembly Language Representation and Graph Generation in a Pure Functional Programming Language, Master's Degree Thesis, Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada, Jan 2005.

[13] Faulk, S.R., Parnas, D.L., On Synchronization in Hard-Real-Time System, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 664 pgs., ISBN 0-201-70369-6.

[14] Heninger, K., Specifying Software Requirments for Complex Systems: New Techniques and Their Application, Collected Papers by David L.Parnas, Addison-Wesley, ISBN 0-201-70369-6. pp. 111–133. 2001.

[15] Heninger, K., Kallander,J., Parnas,D.L., and J.Shore, Software Requirements for the A-7E Aircraft, Naval Res. Lab., Memo Rep. 3876, Washington, DC, Nov. 27, 1978.

[16] Hirvisalo, V. Combining Static Analysis and Simulation to Speed up Cache Performance Evaluation of Programs. Nordic Workshop on Software Development Tools and Techniques, Copenhagen, IT University of Copenhagen, pp. 117–128, August 2002.

[17] Janicki, R., Parnas, D.L., Zucker, J., Tabular Representations in Relational Documents, in Relational Methods in Computer Science, Brink, C., Kahl, W., Springer Verlag Vienna, pp. 184–196, 1997.

[18] Janicki, R., Khedri, R., On Formal Semantics of Tabular Expressions, Science of Computer Programming, 39(2001), 189–214, 2001.

[19] Janicki, R., Wassyng, A., On Tabular Expressions, In D.A. Stewart, ed. Proceedings of CASCON 2003, Markham, Ontario, Canada, 38–52, October 2003.

[20] Joannou, P., Wassyng, A., Modelling for Requirements Analysis and Design, in Software Important to Safety in Nuclear Power Plants, International Atomic Energy Agency, Vienna, Technical Reports Series No. 367, 1994.

[21] Kahl, W., Compositional Syntax and Semantics of Tables, SQRL Report No. 15, McMaster University, Hamilton, Ontario, Canada, 2003, to appear in Formal Methods in System Design.

[22] Kirner, R., Puschner, P., A Simple and Effective Fully Automatic Worst-Case Execution Time Analysis for Model-Based Application Development, Workshop on Intelligent Solutions in Embedded Systems (WISES'03), Vienna, Austria, June 2003.

[23] Lee, L. and Hwang, S.H., Abstract Simulator: A DSP Software Timing Analysis Tool; web pdffile, URL: *http : //www.icspat.com/papers/* ;

[24] Li, Y.S., Malik, S., Performance Analysis of Embedded Software Using Implicit Path Enumeration in Proceeding of the 32nd Design Automation Conference, page 456–461, 1995.

[25] Lisper, B., Fully Automatic, Parametric Worst-Case Execution Time Analysis, MRTC Report, Mardalen Real-Time Research Centre, Mardalen University, Number: ISSN 1404-3041 ISRN MDH-MRTC-97/2003-1-SE, April 2003.

[26] Malik, S., Static Timing Analysis of Embedded Software Proc. of 34th Design Automation Conf. ACM/EEE, pp. 147–52, June 1997.

[27] Ottosson, G., Sjodin, M., Worst-Case Execution Time Analysis for Modern Hardware Architectures, In Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS's97) June 1997.

[28] Parnas, D.L., Precise Description and Specification of Software, Software Fundamentals, in Mathematics of Dependable System II, edited by V. Stavridou, Clarendon Press, pp. 1–14, 1997.

[29] Parnas, D.L., On the Criteria Be Used in Decoposing Systems, Communications of the ACM, 15, 12, pp. 1053-1058, Dec 1972.

[30] Parnas, D.L., Some Software Engineering Principles, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 664 pgs., ISBN 0-201-70369-6, 2001.

[31] Parnas, D.L., Predicate Logic for Software Engineering, IEEE Transaction on Software Engineering, Vol.19, No. 9, 1993, pp. 856–862, Sep. 1993.

[32] Parnas, D.L., Madey, J., Iglewski, M., Precise Documentation of Well-Structured Programs, IEEE Transactions on Software Engineering, pp 948–976, Vol. 20, No.12, Dec.1994.

[33] Parnas, D.L., Madey, J., Functional Documentation for Computer Systems Engineering, in Science and Computer Programming, (Elsevier) 25[1], pp. 41–61, Oct. 1995.

[34] Parnas, D.L., Tabular Representation of Relations, CRL Report 247, McMaster University, Communications Research Laboratory, TRIO(Telecommunications Research Institute of Ontario), 17 pages, Oct. 1992.

[35] Parnas, D.L., Lawford, M., The Role of Inspection in Software Quality Assurance, IEEE Transaction on Software Engineering, Vol.29, No.8, pp 674–676 Aug. 2003.

[36] Parnas, D.L., Less Restrictive Constructs for Structured Programs, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 31–48.

[37] Parnas, D.L., Some Software Engineering Principles, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 255–266.

[38] Parnas, D.L., Design Software for Ease of Extension and Contraction, Software Fundamentals, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 269–290.

[39] Parnas, D.L., Inspection of Safety-Critical Software Using Program-Function Tables, Collected Papers by David L.Parnas, Addison-Wesley, 2001, ISBN 0-201-70369-6. pp. 371–382.

[40] Puschner, P., A. Schedl, A., Computing Maximum Task Execution Times - A Graph-Based Approach, Journal of Real-Time systems, 13(1): 61-79, Jul. 1997;

[41] Puschner, P., Koza, C., Calculating the Maximum Execution Time of Real-Time Programs, Journal of Real-Time Systems, Volume 1, Number 2, pp. 159–176, September 1989.

[42] Shen, H., Implementation of Table Inversion Algorithms, CRL Report 315, McMaster University, Communications Research Laboratory, TRIO, Dec. 1995.

[43] Shen, H., Zucker, J.I., Parnas, D.L., Tabular Transformation Tools: Why and How, proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS' 96), publised by IEEE and NIST, Gaithersburg, MD., pp. 3–11, June 1996.

[44] Stappert, F., Engblom, J., Ermedahl,A., Efficient Longest Executable Path Search for Programs with complex Flows and Pipeline Effects, In Proceedings of the 4th International Conference on Compilers, Architectures, and Synthesis

for Embedded Systems (CASES 2001), Atlanta, Georgia, USA, November 16-17, 2001. URL: *http : //user.it.uu.se/ jakob/*

[45] Xu, J., On Inspection and Verification of Software with Timing Requirements, IEEE Transaction on Software Engineering, Vol.29, pp. 705–720, No.8, Aug. 2003.

[46] Wang, Yali, Display Management System, (A tool to support the Display Method), CRL Report 297, McMaster University, Communications Research Laboratory, TRIO, Apr. 1995.

[47] Ward, M.P., The FermatT Assembler Re-engineering Workbench, International Conference on Software Maintenance 2001, 6th-9th, Florence, Italy IEEE Computer Society, pp. 659-662, November 2001.

[48] Ward, M.P., Reverse Engineering from Assembler to Formal Specification via Program Trans-formations, 7th Working Conference on Reverse Engineering, 23rd-25th Nov., Brisbane, Queensland, Australia, 2000.

[49] Ward, M.P., Specification from Source Code - Alchemists' Dream or Practical Reality? 4th Reengineering Forum, Sep.19-21, 1994, Victoria, Canada, 1994.

[50] Wassyng, A., Lawford, M., Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project, Proc. of FME03 (Formal Methods Europe), Lecture Notes in Computer Science 2805, Springer 2003, pp. 133–153.

[51] Wassyng, A., Janicki, R., Using Tabular Expressions, In Proceedings of International Conference on Software and Systems Engineering and their Applications, Paris, Vol. 4, 1–17, December 2003.

[52] URL: *ftp : //ftp.es.ele.tue.nl/pub/lpsolve*

Appendix A: Example WAT Requirements

Appendix A-1

Requirements specification of **WAT**

Input :

operation	Operation choice from users of the tool
program_name	The name of the program to be analyzed
new_list	The name list (set) of programs that have not been analyzed
analysis_list	The name list (set) of programs that have been processed or in process

Output :

pro	Program variable containing the following elements:
pro.name	The name (ID) of a program
pro.fp	Program's flow property variable as defined in 5.1,
pro.fg	Program's GXL flow graph as defined in 5.1
pro.tg	Program's timing graph
pro.wcet	Program's WCET solution
new_list	The name list (set) of programs that have not been analyzed
analysis_list	The name list (set) of programs that have been processed or in process

Table_WAT :

$new\_list \cap analysis\_list = \phi \wedge new\_list \cup analysis\_list \neq \phi$
$\wedge$ *program_name* $\in$ {new_list $\cup$ analysis_list} $\wedge$ *operation* $\in$ {"Open", "New"} $\Rightarrow$

	operation = "New"	*operation* = "Open"
program_name $\in$ *new_list*	*pro* = **WCET-Analysis** (**Initial**(*program_name*)) $\wedge$ *analysis_list* = *analysis_list*. add(*program_name*) $\wedge$ *new_list* =*new_list*. delete(*program_name*)	*Not defined*
program_name $\in$ *analysis_list*	*Not defined*	*pro* = **WCET-Analysis** (**Load**(*program_name*))

Table_Initial			True
pro.name =			*program_name*
pro.fp	*.status* =		"Unprocessed"
	.Loops	*.lstatus* =	"Undetermined"
		.Loops =	Null
	.Subss	*.sstatus* =	"Undetermined"
		.Subs =	Null
	.InfPaths	*.istatus* =	"Undetermined"
		.Paths =	Null
pro.fg =			Graphreader (*program_name*)
pro.tg =			Null
pro.wcet =			Null

Table_Load			True
pro.name =			*program_name*
pro.fp	*.statsus* =		*Load.status*
	.Loops	*.lstatus*=	*Load.lstatus*
		.Loops=	*Load.Loop*
	.Subss	*.sstatus*=	*Load.sstatus*
		.Subs=	*Load.Sub*
	.InfPaths	*.istatus*=	*Load..istatus*
		.Paths=	*Load..Inpath*
pro.fg =			*Load.fg*
pro.tg =			*Load.tg*
pro.wcet =			*Load.wcat*

Appendix A-2:

Requirements specification of **WCET-Analysis**

Input:

operation	Operation choice from users of the tool
pro	Program variable as defined in Appendix A-1

Output:

pro	Program variable

Table_WCET-Analysis :

operation ∈ {"View Display", "Flow Analysis", "Low-level Analysis", "WCET Calculation", "Quit" }
∧*pro.fp.status* ∈ {"Unprocessed", "Undetermined", "In process", "Confirmed"} ⇒

		operation =				
		"View Display"	"Flow Analysis"	"Low-level Analysis"	"WCET Calculation"	"Quit"
pro.fp.status = "Unprocessed"		DisplayViewer (*pro.name*)	*pro* = **FlowAnalysis** (*pro*)	Not defined	Not defined	Save(*pro*) ∧ Return
pro.fp.status ≠ "Unprocessed"	*pro.tg* = Null	DisplayViewer (*pro.name*)	*pro* = **FlowAnalysis** (*pro*)	*pro* = **TimingGraph** (*pro*)	Not defined	Save(*pro*) ∧ Return
	pro.tg ≠ Null	DisplayViewer (*pro.name*)	*pro* = **FlowAnalysis** (*pro*)	*pro* = **TimngGraph** (*pro*)	*pro* = **WCETCalculator** (*pro*)	Save(*pro*) ∧ Return

Appendix A-3:

Requirements specification of **Flow-Analysis**

Input:

operation	Operation choice from users of the tool
pro	Program variable as defined in Appendix A-1

Output:

pro.fp	Program's flow property variable
pro.fg	Program's flow graph

Table_Flow-Analysis :

operation ∈ {"View Display", "Flow Analysis", "Low-level Analysis", "WCET Calculation", "Quit" } ∧*pro.fp.status* ∈ {"Unprocessed", "Undetermined", "In process", "Confirmed"} ⇒

	operation =				
	"Subroutine Analysis"	"Loop Analysis"	"Infeasible Path Analysis"	"Refine Flow Graph"	"Quit"
pro.fp.status = "Unprocessed"	*pro.fp* = **SubAna** (*pro*)	*pro.fp* = **LoopAna** (*pro*)	*pro.fp* = **PathAna** (*pro*)	Not defined	Save(*pro*) ∧ Return
pro.fp.status ≠ "Unprocessed"	*pro.fp* = **SubAna** (*pro*)	*pro.fp* = **LoopAna** (*pro*)	*pro.fp* = **PathAna** (*pro*)	*pro.fg* = **GraphRefine** (*pro*)	Save(*pro*) ∧ Return

Table_SubAna:

Num = *pro.Subss.subs.size( )* ∧*SubOperation* ∈ {"Add", "Remove", "View", "Modify/Annotate", "Save", "Low-level Analysis", "View Display", "Quit"} ⇒

			True
SubOperation =	"Add"		*pro.Subss* = *pro.Subss.subs*.add(New(*Sub*))
	"Remove"		*pro.Subss* = $\exists i, 0 \le i < Num$, delete(*pro.Subss.subs*(*i*))
	"View"	N=*i*	$\exists i, 0 \le i < Num$, print(*pro.Subss.subs*(*i*))
		N=*all*	For *i* =0 to *Num*-1 : print(*pro.Subss.subs*(*i*))
	"Modify/Annotate"		*pro.Subss* = $\exists i, 0 \le i < Num$, annotate(*pro.Subss.subs*(*i*))
	"Save"		save(*pro.Subss*)
	"View Display"		**DisplayViewer** (*pro.name*)
	"Low-level Analysis"		**TimingGraph**(*pro*)
	"Quit"		save(*pro.Subss*) ∧ return

Table_LoopAna:

Num = *pro.Loops.loops.size( )* ∧ *LoopOperation* ∈ {"Add", "Remove", "View", "Modify/Annotate", "Save", "Low-level Analysis", "View Display", "Quit"} ⇒

			True
LoopOperation =	"Add"		*pro.Loops* = *pro.Loops.loops*.add(New(*Loop*))
	"Remove"		*pro.Loops* = ∃*i*, 0 ≤ *i* < *Num*, delete(*pro.Loops.loops*(*i*))
	"View"	N=*i*	∃*i*, 0 ≤ *i* < *Num*, print(*pro.Loops.loops*(*i*))
		N=*all*	For *i* =0 to *Num-1* : print(*pro.Loops.loops*(*i*))
	"Modify/ Annotate"		*pro.Loops* = ∃*i*, 0 ≤ *i* < *Num*, annotate(*pro.Loops.loops*(*i*))
	"Save"		save(*pro.Loops*)
	"View Display"		**DisplayViewer** (*pro.name*)
	"Low-level Analysis"		**TimingGraph**(*pro*)
	"Quit"		save(*pro.Loops*) ∧ return

Table_PathAna:

Num = *pro.InfPaths.paths.size( )* ∧ *NumMDX* = *pro.fg.MDX.size( )* ∧ *PathOperation* ∈ {"Determine MDX", "Remove", "View MDX", "View Analysis Records" "Annotate other Infeasible Path", "Save", "Low-level Analysis", "View Display", "Quit"} ⇒

		True
PathOperation =	"Determine MDX"	*For i =0 to NumMDX* : findMDX(*i*) ∧ annotateMDX(*i*) ∧ *pro.InfPaths* = ∀ *j*, 0 ≤ *j* < *NumMDX* ∧ *MDX*(*j*).*feasiblestatus.change* = *True* add (*pro.InfPaths.paths.(MDX*(*j*)))
	"Remove"	*pro.InfPath* = ∃*i*, 0 ≤ *i* < *Num*, delete(*pro.InfPaths.paths*(*i*))
	"View MDX"	For *i* =0 to *NumMDX* : print(*pro.InfPaths.paths.MDX*(*i*))
	"View Analysis Records"	For *i* =0 to *Num-1* : print(*pro.InfPaths.paths*(*i*))
	"Annotate other Infeasible Path"	*pro.Loops* = ∃*i*, 0 ≤ *i* < *pro.Loops.Num*, annotate(*pro.InfPaths.paths*(*i*))
	"Save"	save(*pro.InfPaths*)
	"View Display"	**DisplayViewer** (*pro.name*)
	"Low-level Analysis"	**TimingGraph**(*pro*)
	"Quit"	save(*pro.InfPaths*) ∧ return

Appendix A-4

Requirements specification of **TimingGraph**

Input:

operation	Operation choice from users of the tool
pro	Program variable as defined in Appendix A-1

Output:

pro

pro.fg	Program's GXL flow graph
pro.tg	Program's timing graph

Table_TimingGraph:

pro.fg $\neq \phi \wedge$ *pro.fp.Status* $\neq$ "Unprocessed" $\wedge$ *pro.tg= pro.fg* $\wedge$
NumNode,i, NumEdge,NumSub, NumInfPath $\in$ *Int* $\wedge$
NumNode = pro.fg.graphNodes.size( ) $\wedge$ $0 \leq i <$ *NumNode* $\wedge$
NumEdge[i] = pro.fg.graphNodes[i].edges.size( ) $\wedge$
NumSub = pro.fg.Subss.subssize( ) $\wedge$ *NumInfPath = pro.fg.InfPaths.size( )* $\wedge$
operation $\in$ {"Refine Flow Graph", "Generate Timing Graph"} $\wedge$
refine-operation $\in$ {"Cluster Subroutine", "Remove Infeasible Paths"} $\wedge$
Con1 = (*pro.fp.Subss.sstatus* = "Undetermined") $\wedge$ *Con2* = (*pro.fp.InfPaths.istatus* = "Undetermined") $\Rightarrow$

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">operation = "Refine Flow Graph"</td><td rowspan="2">operation = "Generate Timing Graph"</td></tr>
<tr><td>refine-operation = "Cluster Subroutine"</td><td>refine-operation = "Remove Infeasible Paths"</td></tr>
<tr><td rowspan="2">Con1</td><td>Con2</td><td>Not defined</td><td>Not defined</td><td rowspan="3">pro.tg =
For i = 0 to NumNode-1
For j = 0 to NunEdge-1
pro.tg.node[i].edge[j].exeTime
= NodeTime
(pro.fg.node[i].edge[j]).</td></tr>
<tr><td>¬ Con2</td><td>Not defined</td><td>pro.fg =
For j=0 to NumInfPath-1
Remove(pro.fg.InfPaths.paths[j])</td></tr>
<tr><td rowspan="2">¬ Con1</td><td>Con2</td><td>pro.fg =
For j=0 to NumSub-1
Clsuter(pro.fg.Subss.subs[j])</td><td>Not defined</td></tr>
<tr><td>¬ Con2</td><td>pro.fg =
For j=0 to NumSub-1
Clsuter(pro.fg.Subss.subs[j])</td><td>pro.fg =
For j=0 to NumInfPath-1
Remove(pro.fg.InfPaths.paths[j])</td></tr>
</table>

Table_NodeTime :

branch_node, shift_node ∈ *flowGrapgNode* ∧ *N* ∈ Int ∧ *branch_node.Opercode* ∈ {BSI, BSC, MDX, XIO} ∧ *shift_node.Opercode* ∈ {SLA, SLT,SLCA,SLC,SRA, SRT, RET} ∧
Con1=*branch_node.Operand* ∧ *Con2*=(*N*=*shift_node.Operand*) ⇒

			flowgrapg.node.Format = Null		*flowgrapg.node.Format* ≠ Null	
			flowgrapg.node. Flag = *Null*	*flowgrapg.node. Flag* ≠ *Null*	*flowgrapg.node. Flag* = *Null*	*flowgrapg.node. Flag* ≠ *Null*
flowgrapg.node. Opcode =	LD		*time* = 3.75	*time* =3.75	*time* = 6	*time* = 5.75
	STO		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 5.75
	LDD		*time* = 5.75	*time* = 5.75	*time* = 8	*time* = 7.75
	STD		*time* = 5.75	*time* = 5.75	*time* = 8	*time* = 7.75
	A		*time* = 3.5	*time* = 3.5	*time* = 5.75	*time* = 5.5
	S		*time* = 3.5	*time* = 3.5	*time* = 5.75	*time* = 5.5
	AD		*time* = 5.25	*time* = 5.25	*time* = 7.5	*time* = 7.25
	SD		*time* = 5.25	*time* = 5.25	*time* = 7.5	*time* = 7.25
	M		*time* = 14.75	*time* = 14.75	*time* = 17	*time* = 16.75
	D		*time* = 41.25	*time* = 41.25	*time* = 44	*time* = 43.5
	AND		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 5.75
	OR		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 5.75
	SOR		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 5.75
	BSI	*Con1*	*time* = 1.75	*time* =1.75	*time* = 2	*time* = 1.75
		¬ *Con1*	*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 5.75
	BSC	*Con1*	*time* = 2	*time* = 2	*time* = 2	*time* = 1.75
		¬ *Con1*	*time* = 2	*time* = 2	*time* = 4	*time* = 3.75
	SLA	*Con2*	*time* = 2+N/4	*time* = 2+N/4	*Not defined*	*Not defined*
	SLT	*Con2*	*time* = 2+N/4	*time* = 2+N/4	*Not defined*	*Not defined*
	SLCA	*Con2*	*time* = 2+N/4	*time* = 1.5+N/4	*Not defined*	*Not defined*
	SLC	*Con2*	*time* = 2+N/4	*time* = 1.5+N/4	*Not defined*	*Not defined*
	SRA	*Con2*	*time* = 2+N/4	*time* = 2+N/4	*Not defined*	*Not defined*
	SRT	*Con2*	*time* = 2+N/4	*time* = 2+N/4	*Not defined*	*Not defined*
	RET	*Con2*	*time* = 2+N/4	*time* = 2+N/4	*Not defined*	*Not defined*
	WAIT		*time* = 2	*time* = 2	*time* = 2	*time* = 2
	XIO	*Con1*	*time* = 5.75	*time* = 5.75	*time* = 8	*time* = 7.75
		¬ *Con1*	*time* = 7.75	*time* = 7.75	*time* = 10	*time* = 9.75
	LDX		*time* = 1.75	*time* = 1.75	*time* = 3.75	*time* = 3.75
	STX		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 6
	MDX		*time* = 1.5	*time* = 1.5	*time* = 9.75	*time* = 3.25
	LDS		*time* = 2	*time* = 2	*Not defined*	*Not defined*
	STS		*time* = 3.75	*time* = 3.75	*time* = 6	*time* = 6.75
	CMP		*time* = 3.5	*time* = 3.5	*time* = 5.75	*time* = 5.5
	DCM		*time* = 7.25	*time* = 5.25	*time* = 7.5	*time* = 7.25

Appendix A-5

Requirements specification of **WCETCalculator**

Input:

operation	Operation choice from users of the tool
pro	Program variable
path_set	The possible execution path set of the given program
path_time_set	The execution time set of the possible execution paths of the given program
nonfun_path_set	The non-functional execution path set of the given program

Output:

pro	Program variable and changed elements including
pro.fg	Program's GXL flow graph
pro.tg	Program's timing graph
pro.wcet	Program's wcet solution
lpath	The longest execution path in the timing graph
lpath.path	The path, node sequence representation
lpath.type	The path type of the longest execution path in the timing graph
lpath.time	The execution time the longest execution path in the timing graph

Table_WCETCalculator :

operation ∈ {"Process Subroutine", "Process Loop", "Search Longest Path", "Check Path Type", "Determine WCET", "WCET Report", "Quit"} ∧ $pro.tg \neq \phi$ ∧
$path\_set \cap nonfun\_path\_set = \phi$ ∧ $path\_set \cup nonfun\_path\_set \neq \phi$ ∧
$path\_set \cup nonfun\_path\_set = \{ \forall p \mid p \in \text{Path} \wedge p \in pro.tg \}$ ∧
$pro.fp.Subss.sstatus \neq$ "Undetermined" ∧ $pro.fp.Loops.lstatus \neq$ "Undetermined" ⇒

		True
operation =	"Process Subroutine"	*pro.tg* = For *i* = *0* to *pro.fp.Subss.subs.size( )* Combine(*pro.fp.Subss.subs[i]*) ∨ AnnoateTime(*pro.fp.Subss.subs[i]*)
	"Process Loop"	*pro.tg* = For *i* = *0* to *pro.fp.Loops.loops.size( )* Bound(*pro.fp.Loops.loops[i]*) ∨ AnnoateCount(*pro.fp.Loops.loops[i]*)
	"Search Longest Path"	*lpath.path* = LogestPath(*pro.tg*) ∧ *lpath.time* = *max*(*path_time_set*)
	"Check Path Type"	*lpath.type* = PathCheck (*lpath*) ∧ *lpath.type* ∈ {"functional" ,"nonfunctional"} ∧ *pro.tg* = AdjustTG(*lpath*) ∧ *path_set*=AdjustPS(*lpath*) ∧ *nonfun_path_set*=AdjustNPS(*lpath*)
	"Determine WCET"	*pro.wcet* = *lpath.time* ∧ *lpath.time* = *max*(*path_time_set*) ∧ *lpath.type* = "functional" ∧ *lpath.path* ∈ *pro.tg* ∧ *lpath.path* ∈ *path_set*
	"WCET Report"	Print (*pro.wcet*)
	"Quit"	Save (*pro, lpath*) ∧ EXIT

Appendix A-6

Requirements specification of **ExtendedDisplay**

Input:

operation	Operation choice from users of the tool
pro	Program variable
new_list	The name list of programs that have not been analyzed
analysis_list	The name list of programs that have been processed or in process
display	The display of a given program

Output:

display	The display of a given program

Table_ExtendedDisplay :

$new\_list \cap analysis\_list = \phi \wedge new\_list \cup analysis\_list \neq \phi \wedge pro \neq \phi \wedge display \neq \phi \wedge$ *pro.name* $\in$ { new_list $\cup$ analysis_list } $\wedge$ *operation* $\in$ {"View Display", "Extend Display", "Update Display", "Specify Time", "Quit"} $\Rightarrow$

	operation =				
	View Display	Extend Display	Update Display	Specify Time	Quit
pro.name $\in$ new_list	DisplayViewer (*pro.name*)	*display* = *display*. Supplement (*pro.fp, pro.fg*)	Not Defined	Not defined	Save(*new_list, analysis_list, display*) $\wedge$ Return
pro.name analysis_list	DisplayViewer (*pro.name*)	Not Defined	*display* = *display*. Update (*pro.fp, pro.fg*)	*display* = *display*. AddTime (*pro.tg, pro.wcet*)	

$\wedge NC(pro, analysis\_list, new\_list)$

Appendix B: WAT Tool Specifications

Appendix B-1 WAT Flow Analysis Tool

CLASS	SECRETS	RESPONSIBILITES
FlowAnalysis (L3-01)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) perform "master control" including subroutine, loop and infeasible path analysis 2) manage data files used store original, intermediate and final data in the analysis phases. 3) provide interactive panels for user to determine and modify related information.

ASSUMPTION LIST

1. *FlowAnalysis* has no constraints on the order of performing (1) subroutine (2) loop and (3) infeasible path analyses, and they also can be performed individually. However, any data update and status change caused by an individual component should be noticed to other ones, and proper decisions should be made for whether or not invoking further manipulations.
2. Each flow analysis case is for a program provided a *display*. Its invocated programs are seen as specific nodes, subroutine nodes, and their behaviors are known.
3. User program will be provided proper messages when *WCET Analysis* meets unexpected events or required information is not available

Access function list

Function Name	Parameter Type	Parameter Information
FlowAnalysis	N : String : I [S : SubAna, L:LoopAna, P : Path : I];	The name of program to be analyzed Optional flow property variables
AnalyzeFlow	N : String: I [S : Sub, L : Loop, P : Path, st : String] : I/O P : Panel : O G : FlowGraph : O	Program name Optional flow property variables Interactive panel used to perform control The returned flow graph
ReadFlowData	N : String : I	The name of a flow property data file
GetFlowGraph	N : String : I G : FlowGraph : O	Program name The returned flow graph
GetFlowProperty	N : String : I [S : Sub, L : Loop, P : Path, st : String] : O	Program name Flow property variables
GetStatus	N : String : I S : String : O	The name of a program which is being processed Analysis status

Class Variables:

String name
String fp
SubAna s
LoopAna l
Path p

CLASS	SECRETS	RESPONSIBILITES
SubAna (L2-01)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) Describe subroutine properties of a given program; 2) Create, modify subroutine properties for each invoked programs. 3) Cluster "subroutine nodes" for flow graph. 4) Update extended displays

ASSUMPTION LIST

1. *SubAna* class decomposes subroutines based on the decomposition solution specified in Display. To determine each subroutine variable, user can refer to function tables in the displays.

Access function list

Function Name	Parameter Type	Parameter Information
SubAna	N : String : I	Program name
DefienSub	D : Display : I S : Sub [ ] : I/O P : Panel : O	Program Display in which subroutine decomposition is specified. Subroutine list of a program Interactive panel for program control
GetSubs	N : String : I; S : Sub [ ] : I	Program name. Subroutines invoked by the given program.
ReadSubs	N : String : I S : Sub [ ] : O	The file name of a subroutine data file. Subroutine variable list.
GetStatus	N : String : I S : string : O	Program name; The analysis status.
ModifySub	S : Sub : I/O	A subroutine variable to be modified
ModifyExtDisplay	D : ExtDisplay : I/O G : FlowGraph : I S : Sub [ ] : I	A extended display. Updated flow graph; Sub variable/s.
ClusterGraph	G : FlowGraph : I/O S : Sub [ ] : I	The flow graph to be clustered; Subroutines;

Class Variables:

String name;
Sub [ ] subs
String status

CLASS	SECRETS	RESPONSIBILITES
LoopAna (L2-01)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) Describe Loop properties of a given program 2) Create, modify Loop properties for each invoked program 3) Cluster "Loop nodes" for flow graph 4) Update extended displays

ASSUMPTION LIST

1. Each loop terminates in a finite number of iterations.
2. Loops are identified by checking backward edges specified in the flow graph, and their properties are determined through function specifications.

Access function list

Function Name	Parameter Type	Parameter Information
LoopAna	N : String : I	Program name
DefienLoop	D : Display : I L : Loop [ ] : I/O P : Panel : O	Program display in which Loop decomposition is specified Loop list of a program Interactive panel
GetLoops	N : String : I; L : Loop [ ] : I	Program name Loops invoked by the given program
ReadLoops	N : String : I L : Loop [ ] : O	The file name of a Loop data file Loop variable list
GetStatus	N : String : I L : string : O	Program name The analysis status
ModifyLoop	L : Loop : I/O	A Loop variable to be modified
ModifyExtDisplay	D : ExtDisplay : I/O G : FlowGraph : I L : Loop [ ] : I	A extended display Updated flow graph Loop variable/s
SliceLoopInGraph	G : FlowGraph : I/O L : Loop [ ] : I	A flow graph Loops identified in the program

Class Variables

String name;
Loop [ ] loops
String status

CLASS	SECRETS	RESPONSIBILITES
PathAna (L1-03)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) identify infeasible paths in flow graph. 2) determine the feasibilities of MDX statements. 3) find paths indicated by the function table 4) check whether a path is indicated by the function table. 5) refine flow graph by removing identified infeasible paths

ASSUMPTION LIST

1. PathAna class is used to analysis feasibilities of program execution paths in a given flow graph.
2. Condition *Headers* in Program function tables indicate execution paths that can be extracted. Statements used to implement conditions are seen as key clue to determine such paths.

Access function list

Function Name	Parameter Type	Parameter Information
PathAna	N : String :I	Program name
ReadInfPaths	N : String : I	The name of infeasible path data file name
GetInfPaths	N : String : I P : Path [ ] : O	The name of infeasible path data file name The obtained path
FindMDX	C : String [ ] : I M : MDX [ ] : O	Program source code scope determined in Display MDX statements code address list
DetermineMDX	M : MDX [ ] : I/O D : ExtDisplay : I P : Penal : O	MDX statement list to be determined Display for information reference Interactive panel
FindPathInTable	D : ExtDisplay : I P : Path [ ] : O	 Path listed extracted from function tables
InfeasiblePath	P : Path [ ] : O D : ExtDisplay : I M : MDX [ ] : I	Infeasible path list Identified infeasible paths caused by interpreted MDXes
CheckPathType	P : Path [ ] : I T : String : O	Path list Type concluded from the path
CheckPathInTable	P : Path : I D : Display : I B : Boolean : O	A path to be checked Display of the given program Returns "true" when the path is specified in the display function table, otherwise return "false"
RefineGraph	P : Path [ ] : I G : FlowGraph : I/O	Identified infeasible paths The target flow graph/

Class Variables

String	name
Path [ ][2]	infpath; type
String	status

CLASS	SECRETS	RESPONSIBILITES
FlowGraph (L0–02)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions	1) read original control flow graph data into program variables 2) provide graph, graph nodes/edges and their item information to other modules 3) Perform graph modifications on both graph structure and node context

ASSUMPTION LIST

(1) Original program control flow graph is generated by *Graph Generation Module* , and the graph is described in GXL or XML
(2) *Sub-graph tools* are available for extracting the flow graph for the target program
(3) Proper messages will be prompted when data is not available or other unexpected events occur

Function Name	Parameter Type	Parameter Information
FlowGraph	n: String: I	Flow graph name
ReadGraph	n : String : I	The name of a flow graph data file
GetGraph	n : String : I g: Vector<Node> : O	Flow graph name A vector stores flow graph nodes
GetStatus	n : String : I s :String : O	Flow graph name Flow graph process status
SaveGraph	n: String: I	Flow-graph name
GetNode	A : String: I, n : Node : O	Node core address, and the returned graph node
SetNode	A : String : I, n : Node : O	Node core address, and the returned graph node
DelNode	a: String: I	Core address of the node
AddEdge	a1, a2: String: I	The core addresses of source and sink nodes
RemoveEdge	a1, a2: String: I	The core addresses of source and sink nodes
GetEdge	a1, a2: String: I e : Edge : O.	The core addresses of source and sink nodes A flow graph edge
GetTime	e : Edge : I, t : real : O	Requested edge, and the assigned timing data
SetTime	e : Edge : I, t : real : I	Graph edge, and the execution time of the source node
GetNum	n : Node : I, num : real : O	Graph node, and the number of times the node executed
SetNum	n : Node : I, num : real : O	Graph node, and the number of times the node executed
ClusterNodes	a1, a2 : String : I	Start and end addresses for a code block to be clustered.

Class Variables :

Vector <Node> nodes
String status

CLASS	SECRETS	RESPONSIBILITES
Subroutine (L0-03)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions e.g. interactive panels	1) describe program's subroutine properties 2) provide subroutine property information 3) provide interactive interfaces for users to determine and annotate related information 4) search and tag possible sub-subroutine/s

ASSUMPTION LIST

1. Subroutine decomposition is identical with decomposition solutions in provided program displays.
2. A subroutine may invoke sub-subroutines and they may call other subprograms.
3. A subroutine may consists more than one sequential code slices, and its code scope does NOT include the source code of its subroutines;
4. Features will not be changed including:
   - data structure, and basic access functions specified below.
5. Following features may been changed :
   - subroutine identify functions, such as: MDX, BSI/DC/BSI invocation identification and etc.
   - panel formats used for users to input/modify data.

Access function list

Function Name	Parameter Type	Parameter Information
Subroutine	N : String : I	The name (label) of a subroutine
ReadSub	N : String : I	The name of a subroutine data file
GetSub	N : String : I S : Subroutine : O	The name a subroutine A subroutine variable
GetScope	S : Vector<String>: O	The returned subroutine slice scopes
GetsubSubs	subs : Vector<Subroutines>: O	The returned sub-subroutines
GetExeB	exeB : int: O	Upper bound of the number a subroutine been executed.
GetExeN	exeN : int : O	The number of subroutine been executed
GetTime	time : real : O	Execution time of a subroutine
GetWCET	wcet : real : O	WCET of a subroutine
GetStatus	s : String : O	The analysis status of a subroutine
ModifySub	N : String I subs : Vector<Subroutines>: O P : Panel : O	The name of a subroutine to be modified The returned sub-subroutines Interactive panel for modifying data
RemovSub	n : String : O subs : Vector<Subroutines>: O	The name of a subroutine to be removed The returned sub-subroutines

Class Variables :

String name
Vector <String> scope ; {{s1,e1},{s2,e2},.....,{sn,en}}
Vector <String> subsubs
int exeBound, exeNum
float timeBound, time
String status

CLASS	SECRETS	RESPONSIBILITES
Loop (L0-04)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) describe program's loop properties 2) provide loop property information 3) provide interactive interfaces for users to determine and annotate related information *4) search and tag possible loop/s

ASSUMPTION LIST

1. Loops are identified through tools in other modules, and their information is saved in *loop data files*
2. A loop may be nested with subroutines or loops
3. A loop may consists more than one sequential code slices, and its code scope does NOT include the source code of its subroutines
4. Features will not be changed including: data structure, and basic access functions specified below
5. Following features may been changed : panel formats used for users to input/modify data

Access function list

Function Name	Parameter Type	Parameter Information
Loop	N : String : I	The name (label) of a loop
ReadLoop	N : String : I	The name of a subroutine data file
GetLoop	N : String : I L : Loop : O	The name a subroutine A loop property variable
GetScope	S : Vector<String>: O	The returned subroutine slice scopes
Getloops	loops : Vector<String>: O	A name list of subroutines nested in a loop
GetLoop	Ls : Vector<Loop> : O	A list of loops nested in a loop
GetExeB	exeB : int: O	Loop upper bound
GetExeN	exeN : int : O	The number of loop body been executed
GetTime	time : real : O	Execution time of a loop
GetWCET	wcet : real : O	WCET of a loop
GetStatus	s : String : O	The analysis status of a loop
ModifyLoop	n : String : I P : Panel : O	The name of a loop The interactive panel for modifying data
RemoveLoop	n : String : I Ls : Vector<Loop> : O	The name of a loop A list of loops nested in a loop
SaveLoop	n : String : I	The name of data file name

Class Variables :

String name
String entrance
Vector <String> loopbody ; {{s1,e1},{s2,e2},.....,{sn,en}}
String exit
Vector <String> subs
Vector <String> subloops
int exeBound, exeNum
float timeBound, time
String status

CLASS	SECRETS	RESPONSIBILITES
Path (L0-05)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) describe execution path's properties 2) provide information of identified path/s 3) provide interactive interfaces for users to determine and annotate path related information 4) compose a program execution path.

ASSUMPTION LIST

1. Class ***Path*** is to help user identify and determine *infeasible paths* exist in the generated flow graph, rather than analyze the program
2. This class is also designed to analyze the flow paths indicated by the program function table
3. Features will not be changed including: data structure, and basic access functions specified below
4. Following features may been changed: panel formats used for users to input/modify data

Access function list

Function Name	Parameter Type	Parameter Information
Path	A : String [ ] : I P : String [ ] : O	The core addresses of nodes to construct a path The address sequence of a path
ReadPaths	F : String : I	The name of a infeasible path data file
GetPath	A1, A2 : String : I P : String [ ][4] : O	The core addresses represent start and end nodes of a path Information of an infeasible path
ModifyPath	A1, A2: String : I	The start/end addresses of a path
AppendPath	P1 : String [ ] : I, P2 : String [ ] : I P : String [ ] : O	Two paths to be merged. Core addresses of a merged path
RemovePath	P : String [ ] : I	The core addresses of nodes in a path.
CheckPath	P : String [ ] : I T : String : O	A path. Specified type of a given path.
GetStatus	S : String : O	The path analysis status.
SavePath	N : String : O	The name of data file

Class Variables :

Vector <String> Path; {{s0,e0, t0, c0}, ..., {sn, en, tn, cn}}

CLASS	SECRETS	RESPONSIBILITES
MDX (L0-06)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) describes MDX statements' properties 2) finds suspect branch MDXs in the slices consists the program 3) determine

ASSUMPTION LIST

1. Class ***MDX*** is to help user identify and determine *infeasible paths* caused by MDX statement (IMB 1800 assembly language) in the generated flow graph,
2. Guess-and-Verify method is used to determine the feasibility of MDX statements which were interpreted as alternative branch statements
3. Function table specifies all of the variable value changes which can used for MDX analysis
4. This class can be removed when the flow graph generator supplemented with MDX process module, and it will not affect other modules in the WCET tool
5. Features will not be changed including: data structure, and basic access functions specified below
6. Following features may been changed: panel formats used for users to input/modify data
7. Proper interactive panels are provided for related methods

Access function list

Function Name	Parameter Type	Parameter Information
MDX	A : String : I	The core address of a MDX statement.
ReadMDXs	F : String [ ] : I	The name of a MDX data file.
FindMDXs	S: String [ ] : I M : String [ ] : O	Program scope (start/end core addresses of program slices). MDX statement (core address) list.
ModifyMDX	A: String : I	Core addresses of a MDX statement.
RemoveMDX	A: String I	The core address of a MDX statement.
CheckMDX	A: String: I F : Boolean : O	The core addresses of a MDX. Returns "true" if the MDX is a conditional branch statement, otherwise returns "false".
GetStatus	S : String : O	The MDX analysis status.

Class Variables :

Vector <String> MDX; {{a0, t0, s0, c0}, ..., {an, tn, sn, cn}} // address, type, status, comments

Appendix B-2 WAT Low-level Analysis Tool

CLASS	SECRETS	RESPONSIBILITES
TimingGraph (L1-02)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) Generate timing graph 2) Present timing graph information. 3) Provide interfaces for users to modify related information. 4) Process nested relationships, such as loops in subroutines and subroutines in loops 5) Process hardware effects

ASSUMPTION LIST

1. *Timing graph* (TG) class specifies a kind of graphs represented in GXL/XML scripts
2. TG can be refined if and only if at least one of its items is modified

Access function list

Function Name	Parameter Type	Parameter Information
TimingGraph	FG : FlowGraph : I S : Subroutine [ ] : I L : Loop [ ] : I H : HDFeature : I T : TimeTable : I TG : TimingGraph : O	A flow graph to be processed All the Subroutine variables for a program All the Loop variables for a program Identified Hardware Features The Statement execution time table A timing graph to be generated
ReadTG	N : String : I TG : TimingGraph : O	The name of a timing graph data file A timing graph
GetTG	N : String : I TG : TimingGraph : O	The name of a timing graph The returned timing graph
HidePath	P : Path : I TG : TimingGraph : O	A path to be hidden The timing graph in which one or more path are hidden
UnHidePath	P : Path : I TG : TimingGraph : O	A path which was hidden before The timing graph in which one or more path are unhidden
SubInLoop	L : Loop : I S : Subroutine [ ] : I TG : TimingGraph : O	A loop in which one or more subroutines are invoked Subroutines nested in a loop A timing graph
LoopInSub	S : Subroutine : I L : Loop [ ] : I TG : TimingGraph : O	A subroutine contains one or more loops Loops nested in a subroutine A timing graph
RefineTG	TG : TimingGraph : I S : Subroutine [ ] : I L : Loop [ ] : I H : HDFeature : I T : TimeTable : I TG : TimingGraph : O	A flow graph to be processed All the Subroutine variables for a program All the Loop variables for a program Identified Hardware Features The Statement execution time table A timing graph to be generated

CLASS	SECRETS	RESPONSIBILITES
HDFeature (L0-07)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) describe hardware features affect program execution time

ASSUMPTION LIST

1. ***HDFeature*** is designed to specify identified hardware features that may affect program execution time
2. In this class, timing effects are represented as *delay* or *speed up* functions for related code slices
3. This class is extensible, i.e., new identified features can be added, and be contracted with exist ones

Access function list

Function Name	Parameter Type	Parameter Information
HWFeature	N, T: String : I D : Float : I S : String [ ] : I	The name and type of a hardware feature Delay or speed time value/function of a HW feature The scope that the HW feature affects
LoadFeatures	F : String : I	Hardware feature file name
GetFeatures	N : String [ ] : I E : Vector <Effects>: O	The name a hardware feature A or a list of features
AddFeature	N, T: String : I D : Float : I S : String [ ] : I	The name and type of a hardware feature Delay or speed time value/function of a HW feature The scope that the HW feature affects
ModifyFeature	N : String : I	The name of a hardware feature

Class Variables :

String : name;
String : type;
Float : delay;
String [ ] : scope.

CLASS	SECRETS	RESPONSIBILITES
TimeTable (L0-08)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) describe IMB 1800 statement time data 2) provide statement timing information 3) modify statement timing data

ASSUMPTION LIST

1. The basic statement average execution time information is obtained from the hardware manual provided through hardware vendors, and such information is collected in a *statement timing table*
2. The execution time of a statement is represented as a sum of its average statement time with a known delay, or a subtraction with its speedup value.

Access function list

Function Name	Parameter Type	Parameter Information
TimeTable	F : String : I	The file name of the average statement timing data file
AdjustTiming	S : String [ ][3] : I D : float [ ] : I	A list of statement names, formats, tags A or a list of real number (delay/speedup data)
GetTime	S, F, T: String : I T : float : O	A statement name, format and tag A real number (execution time)
GetTimeTable	T : float [ ][4] : O S : String [ ] : O	A complete time table of all statements A complete name list of all statements

Class Variables :

```
String name;   //statement name
Char F;        //format
Char T;        //tag
Float t;       // time
```

Appendix B-3 WAT WCET Calculator

CLASS	SECRETS	RESPONSIBILITES
Calculator (L2-04)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) calculate the WCET for given timing graph and other information 2) translate GXL/XML graph to real graph data structure for graph operations 3) specify timing solution in extended displays

ASSUMPTION LIST

1. *Calculator* is to compute the WCET for a program based on its given timing graph. How to use the timing information of its invoked programs should be determined and annotated
2. *Calculator* will translate provided timing graph (stored in script or data bases formats) into math graph for further processing
3. User program will be provided proper messages when *WCETAnalysis* meets unexpected evens or required information is not available

Access function list

Function Name	Parameter Type	Parameter Information
Calculator	N : String : I	Name of the program
GraphGenerator	TG : TimingGraph : I S : ProSub : I G : Graph : O	The timing graph generated from other components Subroutine invocation information The math graph will be used for longest path searching
GetWCET	D : ExtDisplay : I TG : TimingGraph : I S : ProSub : I G : Graph : I W : WCET : O	The display can be used as information reference Timing graph to perform timing calculation and path search Subroutine process variable Graph used to perform longest path search WCET solution variable
UpdateDisplay	D : ExtDisplay : I/O W : WCET : I	Display of the program WCET solution

CLASS	SECRETS	RESPONSIBILITES
ProSub (L1-03)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) determine how to use subroutine's the timing information for WCET calculation. 2) determine relationships of subroutine/loop nest activities. 3) process subroutine node in a given graph

ASSUMPTION LIST

1. An invoked subroutine may take different time in different invocation circumstances. To calculate the WCET of its caller program, each invocation has to be determined an unique time value.
2. The number of loop iterations also needs to be determined when subroutines nested in loops.

Access function list

Function Name	Parameter Type	Parameter Information
ProSub	S : Subroutine : I	A subroutine to be specified execution time
SubTime	S : Subroutine : I T : float : O	A subroutine variable The time estimate for a subroutine
SubInLoop	S : Subroutine : I L : Loop : I T : float : O	A subroutine which in a loop A loop The subroutine execution time considered loop iteration
GraphSubNode	G : Graph : I S : Subroutine : I RG : Graph : O	The graph for the longest path searching The subroutine node to be processed A graph in which subroutine node is processed

Class Variables :

```
Sting      program_name
String [ ] sub_names
Float  [ ] sub_time
Sting  [ ] sub_use
```

CLASS	SECRETS	RESPONSIBILITES
LongestPath (L1-04)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) search the path takes the longest execution time in a timing graph. 2) check the type of the longest path 3) check whether a found path is indicated in the function table. 4) compute the execution time for a given path

ASSUMPTION LIST

1. The longest path searching could be performed either in the timing graph, or in a scope of paths indicated in the function table or specified by the user
2. The execution time is represented in real number for if provided sufficient information about program's flow properties and related timing data
3. When the program contains one or more segments that their execution time cannot be determined, a path set is provided in which paths have been determined that can not be the longest path will not be included

Access function list

Function Name	Parameter Type	Parameter Information
LongestPath-G	G : Graph : I P : Path [ ]: O	A graph for WCET calculation A core address list represents an execution path
LongestPath-T	P : String [ ] [ ]: I P : Path [ ] : O	A path set for WCET calculation A core address list represents an execution path
CheckPath	P : Path : I T : String : O	A execution path represented as a code address list The type of the path belongs to
GetTablePaths	D : Display: I P : String [ ][ ] : O	The extended display which provides the function table Path set indicated by a given table
PathTime	P : Path : I; T : float : O	The path for time estimation The execution time for a path

Class Variables :

String longest_path_name
String longest_path_type
Float longest_path_time
String [ ] longest_path_unknownslices

CLASS	SECRETS	RESPONSIBILITES
Graph (L0-09)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) present GXL/XML graph in graph data structure 2) tag paths in a graph 3) check properties of a given graph

ASSUMPTION LIST

1. In the WCET tool, *program control flow* graph and *timing graph* are represented in GXL or XML formats. The *Graph* class is designed to store such data in real graph data structure for further graph operations
2. It is extensible for this class to add other access functions to read and translate data base information into graph data structure

Access function list

Function Name	Parameter Type	Parameter Information
Graph	F : String : I	A GXL or XML script file name
CheckGraph	N : String [ ] : I P : String [ ] : I S : Boolean [ ] : O	A name of a graph A property name list for graph checking A list of checking Boolean solutions. Note, for each checking the solution is "true" if all required properties are satisfied, otherwise, is "false".
GetGraph	N : String : I G : Graph : O	Name of a graph The returned graph
GetStatus	S : String : O	The graph process status
TagPath	P : String [ ] : I G : Graph : O	The core addresses consist a execution path A Graph

CLASS	SECRETS	RESPONSIBILITES
WCET (L0-10)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) calculate the WCET for a graph 2) calculate the WCET for a set of paths, e.g. paths identified in the function table 3) report WCET analysis solutions

ASSUMPTION LIST

1. In the case of knowing the flow and timing information of all segments of a program, the WCET is calculated and reported
2. When exist unknown segments, paths that can not be the longest path will be picked out, and remainder paths will be reported their timing solution including which segments are unknown and the execution time of known slices

Access function list

Function Name	Parameter Type	Parameter Information
WCET_G	G : Graph : I W : Float : O	A timing graph The WCET value
WCET_P	P : String [ ] [ ] : I W : Float : O	A path set The WCET value
WCET_Reprot	P : String [ ] [ ] : O T : float [ ] : O	The path set which contains unknown segment/s The execution time of known segments

Class Variables :

Float wcet
String [ ] wcet_report
String status

Appendix B-4 WAT Display Manager

CLASS	SECRETS	RESPONSIBILITES
ExtDisplay (L1-01)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) Describes extended displays 2) Supplement graph and other variables into conventional displays 3) Provide interfaces for users to modify related information

ASSUMPTION LIST

1. *Extended-display* class takes items (function table, source code, and invoked program function tables) from the conventional *displays* and is supplemented with (1) original and refined flow graph, (2) flow property variables and (3) timing variables
2. Above supplemented items can be modified by the modules in WCET tool, but others can not
3. All of the *Information View Actions* provide one or more visible pictures when required items are available

Access function list

Function Name	Parameter Type	Parameter Information
ExtDisplay	D : Display : I	A conventional display to be extended
ViewExtDisplay	N : String : I	Display (program) name
ViewItem	N : String : I NI, T : String : I F : String : O	The name of a extended display The name and type of the item to be viewed File name which is used to invoke a visible viewer
AddGraph	N : String : I; G : FlowGraph : I	Display name The flow-graph to be supplemented into an extended display
AddVariable	N, V, T : String: I	The name of an extended display, and the name and type of a variable to be supplemented into the display

Class Variables :

Display display
FlowGraph original_fg, refined_fg
String [ ] flow_property
Float [ ] time

CLASS	SECRETS	RESPONSIBILITES
Display (L0-01)	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions	1) represent a display of a given program, including its function table, program source code, and tabular specification of invoked program/s 2) provide visible viewer for user to view program function table, and source code slices

ASSUMPTION LIST

(1) Display construction is performed first, and program Displays are NOT allowed to be modified during the WCET analysis
(2) Features will not be changed including:
- Class data structure, access functions' interfaces

(3) Features may be changed:
- Display data format, note, current Displays are in PNG image format, and future Displays may be presented in HTML format

(4) Proper messages will be prompted when data is not available or other unexpected events occur

Access Function Table

Function Name	Parameter Type	Parameter Information
Display	name : String : I	Display (program) name
ViewDisplay	name : String : I	Display (program) name
GetTable	file : String : O	The file name of a function HTML/XML table data file
GetCode	file : String : O	The file name of a source LST code data file
GetSubs	subs: Vector <String> : O	The file names of a list of subroutine names
ViewTable	name: String: I	The name of a function table
ViewCode	name : Vector <String> : I	The name list of a set of code slices

Class Variables:

String : table-ID should be either file name of HTML links
String [ ]: code-segments
Vector <String> sub-tables

Appendix B-5 WAT master control class WCET-Analysis

CLASS (L4-01)	SECRETS	RESPONSIBILITES
WCETAnalysis	1) data structure 2) process sequences/algorithms of access functions 3) other internal process functions, e.g. interactive panels	1) performs "master control" including program flow, low-level analysis, and WCET calculation. 2) manage data files used store original, intermediate and final data in the analysis phases. 3) provide interactive panels for user to determine and modify related information, or choose proper operations.

ASSUMPTION LIST

1. The first execution of WCET analysis of a given program should be performed in the order of (1) flow analysis (2) low-level analysis and (3) WCET calculation. Any data update and status change in early steps should be reported to later ones, and those information can be used to decide whether invoke further manipulations or not.
2. Each WCET analysis case is focus on a program provided a *display*, in which the program and invocated programs' behavior specifications are correct/accurate.
3. User program will be provided proper messages when *WCETAnalysis* meets unexpected evens or required information is not available.

Access function list

Function Name	Parameter Type	Parameter Information
WCETAnalysis	N : String : I[	The name of program to be analyzed.
AnalyzeWCET	N : String: I F : FlowAnalysis : I/O T : TimingGraph : I/O C : Calculator: I/O P : Panel : O	Program name Three optional components of the WCET analysis: program flow analysis, low-level analysis (timing graph generation) and WCET calculator. Interactive panel for WCET analysis
ReadWCETData	N : String : I	The name of a WCET data file.
GetWCET	N : String : I W : WCET : O	Program name. The returned wcet for a program.
GetStatus	N : String : I S : String : O	The name of a program which is being processed Analysis status

Class Variables:

String name
FlowAnalysis fp
TimingGraph
WCET wcet
ExtDispaly d