A Unified Flow Information Language for WCET Analysis

Andreas Ermedahl†
IT-Dept. Uppsala University
Box 337, SE-751 05 Uppsala
Sweden
andreas.ermedahl@it.uu.se

Jakob Engblom†
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
jakob.engblom@iar.se

Friedhelm Stappert∗
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
friedhelm.stappert@c-lab.de

Abstract

In this paper we raise the question if it is possible to create a unified flow information language that all WCET research groups can agree upon, and that is independent of flow analysis and calculation methods.

We discuss desired characteristics of such a flow information language and describe the type of flows that it should be able to express. We present our previously published flow fact annotation language and discuss how it fulfils the desired language properties.

1. Introduction

A correct WCET calculation method must take into account the possible program flow, like loop iterations and function calls. For expressing program flows numerous annotation languages have been presented in the WCET literature. The expressiveness and the type of flows that can be handled by these languages mostly depend on the characteristics of flow analysis methods used, rather than being targeted for the potential WCET tool user.

To generate a WCET estimate, we consider a program to be processed through the phases of program flow analysis, low level analysis and calculation. Most WCET research groups make a similar division notationally, but sometimes integrate two or more of the phases into a single algorithm.

The program flow analysis phase determines possible program flows, and provides information about which functions get called, how many times loops iterate, if there are dependencies between if-statements, etc. The information can be obtained by manual annotations (integrated in the programming language [14] or provided separately [6, 9, 19]). The flow information can also be derived using automatic flow analysis methods [7, 10, 13, 22].

In the calculation phase a program WCET estimate is derived, combining the information derived in the program flow and low-level analysis phases. There are three main categories of calculation methods proposed in literature: tree-based, path-based, and IPET (Implicit Path Enumeration Technique).

In a tree-based approach the WCET is calculated in a bottom-up traversal of a tree generally corresponding to a syntactical parse tree of the program, using rules defined for each type of compound program statement (like a loop or an if-statement) to determine the execution time at each level of the tree [1, 2, 16, 20].

In a path-based approach the possible execution paths of a program or piece of a program are explored explicitly to find the longest path [10, 12, 22, 23]. The path-based approach is natural within a single loop iteration or function.

In IPET, program flow and low-level execution time are modeled using arithmetic constraints [6, 9, 15, 18, 21]. Each basic block and program flow edge in the program is given a time (t\textsubscript{entity}) and a count variable (x\textsubscript{entity}), and the goal is to maximize the sum \(\sum_{i\in\text{entities}} x_i \ast t_i\), subject to constraints reflecting the structure of the program and possible flows.

2. Representing Program Flow

The program flow phase can be further divided into three different subphases:

2. Flow representation: Representing the results of the flow analysis.
3. **Calculation:** Using the control flow information (as represented in the flow representation) in the final WCET calculation.

Some WCET methods integrate two or more of the phases. We believe that the separation of the flow analysis from the calculation reduces the complexity of each stage. Also, by keeping the flow analysis phase separate from the flow representation, results from several different flow analysis methods and manual annotations can be integrated and used together in the calculation phase.

When designing a language for expressing flow information there are a number of choices to be made:

- **Expressiveness:** What type of flows should be possible to express? What type of language constructs should be used?
- **Code relation:** How is the information related to different entities in the program code?
- **Calculation conversion:** How should the information be used in the final calculation phase?

### 2.1. Expressiveness

We first note, that a natural way to give flow information is by constraining the number of times different program entities, e.g. loops, statement, nodes or edges, can be taken. This can either be precise bounds, e.g. that a loop is iterated exactly ten times, or upper or lower bounds, e.g. that node A can’t be taken more than five times. It is also beneficial if we can relate the executions of different program entities, e.g. that node A and node B will always be executed together.

The language can consist of named special relations between entities (e.g. using constructs like Parks `samepath(A,B)` and `nopath(A,B)` [19]). An alternative is to use a more generic style based on math, like our flow fact language [6]. The benefit of a generic math-based language is that it can express flows that are hard to put in words and that there is no obvious limit to the types of flows that can be expressed. On the other hand, a special purpose language is easier to understand, but requires that new language constructs are invented in order to express new flows.

The language must reflect the flows found in real-world programs. Researchers have investigated embedded software [4], the RTEMS operating system [3] and common signal-processing algorithms [8]. The results are not in complete agreement on the properties and flows typical for embedded software, showing that more research and knowledge is needed here.

One observation is that flow information is mostly local in its nature, specifying something valid for a small part of a program or a particular invocation of a function. Thus, it is not always suitable to specify flow information once for each entity in the program. E.g. we would like to be able to specify that some node A can’t be executed during the first five iterations of a loop or give a loop bound valid for just some particular executions of a loop. A language should allow for such local flow information to be expressed.

### 2.2. Code Relation

First we note that it is natural to express flow information in relation to the entities available in the program code. Flow information can be provided in relation to the source code, intermediate code in a compiler, or the object code. If provided on source code level, the information must be mapped to the object code to be used in the WCET calculation. In the presence of optimizing compilers, this problem is non-trivial [5, 17].

Automatic flow analysis is probably easier to perform at the source code or intermediate code, since variables and other entities of interest are harder to identify in optimized object code. Also, for the potential WCET-tool end-user manual annotations are typically easier to provide at the source-code level.

Another issue is if the flow information should be included as a part of the programming language or provided outside the program. The benefit of language inclusion is that it forces the programmer to write code in an analysable manner. However, this requires compiler support and makes it harder to try different scenarios.

Specifying the flow information outside the program source allows it to free itself from the static structure of the program. For example, by using a call-graph representation, we can differ between invocations of the same function when called from different places in the code. An example of the extended version is our scope graph representation [6].

A good language should provide stability in that program changes not related to annotated code should not force the annotations to change. For example, a problem with expressing flow information on the object code level is that the information might need to be regenerated every time the program code changes.

An important issue is the ability to handle unstructured code, e.g. due to uses of `goto` and jumps into loops. An optimizing compiler might produce unstructured object code from structured source code, and automatic code for state machines also tends to be unstructured. A general purpose flow information language must be general enough to express flows over such unstructured code.

### 2.3. Calculation Conversion

Regardless of the flow information language used the extracted flow information must be "compiled" or
"adapted" to the calculation method used. The adaptation must be safe: never exclude execution paths which are considered possible by the flow information, and tight: including as few extra execution paths compared to the provided flow information. Figure 1 gives example code showing that not all calculation methods can take advantage of all types of flow information.

The tree-based method [1, 2, 16, 20] is conceptually simple and computationally cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependencies between statements. For example, the code and flow information in Figure 1(a) causes problems in a tree-based calculation method since the timing of the first if-statement will be calculated in isolation from the second if-statement.

The path-based approach is natural within a single loop iteration or other executions of one loop [11, 23]. The method has problems with flow information stretching over loop borders and/or flow information on the total number of times entities are taken. For example, the path-based method has problems handling the “triangular” loop dependency in Figure 1(b). If WCET calculation is performed locally, the WCET calculation for the inner loop will assume 10 iterations, and the WCET calculation for the outer loop will use 10 executions of the inner loop, leading to the body of the inner loop being counted 100 times, when it is actually never executed more than 55 times.

For IPET very complex flows can be expressed using constraints, but all flow information needs to be given on a global program level [6, 9, 15, 18, 21]. This contradicts the need to specify flow information in a local context. As shown in [6], local flows can be handled by unrolling the program and lifting the information to a global level. Since flow information is given as relations over count variables some type of flow implications are problematic to express. E.g. Figure 1(c) shows an example of code where we would like to express an implication dependency like: “if F is taken once then (and only then) G can be taken several times, but if F is not taken then G can not be taken either”.

3. Our Flow Fact Language

This chapter describes our previously published flow fact annotation language [6] and discusses how it fulfills the desired language properties.
iterations, node 0 must be taken exactly three times.

Compared to the criteria given above, we note that the flow facts language uses the math-based style and allows us to give local information. The information is given outside the code and uses an expanded version of the call graph (and thus the control flow graph). In its current version, it cannot handle all types of unstructured code due to the need for a header, and since it relates to the object code, it is very sensitive to program changes.

It has been used to perform both IPET- and path-based calculations [6, 23], but not all facts could be used in the path-based approach. It is interesting that the path-based calculation recognized certain types of facts as meaning “samepath” or “not samepath”, and exploited these by rewriting the graph.

References