WCET Analysis: The Annotation Language Challenge [*]

Raimund Kirner[†], Jens Knoop[‡], Adrian Prantl[‡], Markus Schordan[‡], Ingomar Wenzel[†]
Vienna University of Technology, Austria

Abstract

Worst-case execution time (WCET) analysis is indispensable for the successful design and development of systems, which, in addition to their functional constraints, have to satisfy hard real-time constraints. The expressiveness and usability of annotation languages, which are used by algorithms and tools for WCET analysis in order to separate feasible from infeasible program paths, have a crucial impact on the precision and performance of these algorithms and tools. In this paper, we thus propose to complement the WCET tool challenge*, which has recently successfully been launched, by a second closely related challenge: the* WCET annotation language challenge*. We believe that contributions towards mastering this challenge will be essential for the next major step of advancing the field of WCET analysis.*

***Keywords**: Worst-case execution time (WCET) analysis, annotation languages, WCET tool challenge, WCET annotation language challenge.*

1 Motivation

The precision and performance of worst-case execution time (WCET) analysis depends crucially on the identification and separation of feasible and infeasible program paths. This information can automatically be computed by appropriate tools or manually be provided by the application programmer. In both cases some dedicated language is necessary in order to annotate this information and make it amenable to a subsequent WCET analysis. Languages used for this purpose are commonly known as *annotation languages*. Over the past 15 years, an array of conceptually quite diverse proposals of annotation languages has been presented. Many of them have been used for the implementation of a WCET tool. A comprehensive survey of WCET tools and methods has been given by Wilhelm et al. [30]. Until now, however, there has been no systematic comparison of the various approaches proposed on annotation languages for WCET analysis.

We believe that this lack is not only a hurdle for students and researchers entering the field of WCET analysis, but that it also constrains the further progress and advancement of the field. In fact, after roughly two decades of vibrant and vigorous research we consider closing this gap a major step both for consolidating the state-of-the-art and for providing a new and strong stimulus for further advancing it.

The purpose and the contributions of this paper are thus three-fold: First, to identify an array of important universally valid criteria, in which the usefulness of annotation languages for WCET analysis becomes manifest. Second, to investigate and classify a selection of prototypical representatives of annotation languages used in practice along these criteria in order to shed light on the relative strengths and limitations of the different annotation concepts. And third, most important and directly based on these findings, to extend the invitation to researchers working in this field to contribute to the challenge of designing novel and superior annotation languages, which will allow the development of even more general and powerful algorithms and tools for WCET analysis.

In addition to providing a thorough, yet smooth and survey-like introduction to annotation languages used in WCET analysis for newcomers to this field, we hope that this endeavor will in fact significantly contribute to attracting the attention of researchers who are working in this field to the challenge, which we call the *WCET annotation language challenge*: the design of novel, elegant, and easy to use powerful annotation languages.

In spite of the tremendous success the research on

[*]This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract P18925-N13. This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/).

[†]Institut für Technische Informatik, Vienna University of Technology, Austria, {raimund,ingo}@vmars.tuwien.ac.at

[‡]Institut für Computersprachen, Vienna University of Technology, Austria, {knoop,adrian,markus}@complang.tuwien.ac.at

WCET analysis has had in the past, and the maturity and usefulness WCET tools have already achieved and proved in practice, we believe that the major next step to further advance their power and widespread dissemination in academia and industry depends crucially on the availability of more expressive and more easily useable annotation languages, which can truly seamlessly be integrated into the tool chain of WCET analysis. We consider the identification of the most important and useful features of annotation languages, the choice of a superior mix, the development and in the long-run the standardization of a language based thereon a major challenge for researchers working in the field of WCET analysis.

In this paper we present this challenge as the *WCET annotation language challenge* to the WCET research community. As pointed out, we believe that mastering it will be the key for further advancing the field of WCET analysis. But in fact, we also believe that mastering it will be essential in order to enable the recently successfully launched *WCET tool challenge*, which has attracted the attention of many WCET tool developers [9, 28], to unfold its strength and impact in full.

2 Assessment Criteria

In this section we introduce and discuss the criteria that we use throughout this paper to assess the merits of the annotation languages and mechanisms considered. We separate these criteria into the two groups of *language design* and *usability* criteria. While the characteristics of the criteria of the first group are under control when designing the language, the characteristics of the criteria of the second group are essentially an outcome of those of the first one. Additionally to these two groups of criteria, we consider a third and orthogonal issue: the existence of a *tool* using the annotation language. This is not directly related to a specific property or feature of an annotation language. In fact, the reasons why a tool has been developed, and vice versa, why not, are many-fold. They are not necessarily related to the language at all. Nonetheless, we consider the availability of a tool an indicator of the general usefulness and usability of an annotation language. Independently of this, it is also valuable as an information on its own. It is worth noting, however, that we do not assess the quality of these tools. This is indeed beyond the scope of this paper. Readers with a deeper interest in WCET tools are invited to refer to the (forthcoming) article by Wilhelm et al. presenting a survey of WCET methods and tools [30].

2.1 Language Design

Expressiveness: We consider this to be the most important criterion of all. Intuitively, expressiveness reflects the capability of an annotation language to describe control-flow paths. Especially important is here, which types of flow information can be described and which ones cannot. An interesting level of expressiveness is *completeness*. It requires that the annotation language allows to precisely describe all feasible paths of arbitrary terminating programs. Other important issues of expressiveness are the capability of an annotation language to cope with inter-procedural program flow or selected iteration ranges of loops.

Important setscrews, which allow a language designer to control the expressiveness of an annotation language, are the means and their capabilities for dealing with *loop bounds*, with *triangle loops*, and, more generally, with *context sensitivity* and the *execution order* of statements. We consider all these throughout this paper. See Section 3.1 for further details.

Annotation placement and abstraction level: The question of placement and abstraction level of annotations has an immediate impact on the usability of any annotation language. This becomes obvious when thinking in terms of the programmer's effort to use a language.

The first design decision that has to be made is concerning the location of annotations: Shall code annotations be directly placed at the locations of the source code they describe, or shall they be provided in a separate file? None of the two options is always superior and thus consistently preferable to the other. As a rule of thumb: a) if annotated manually, it is usually more convenient to annotate the source code, b) if annotated automatically by appropriate tools, the usage of separate files often turns out to be advantageous.

The second design decision concerns the issue of annotating the source code or the object code. From a (human-centered) usability perspective, source code annotations are generally preferable. This appears to be obvious, if code annotations are manually provided. But it also holds, if flow information is automatically computed by a tool. The reason is that it is often obligatory or at least desirable to verify these annotations manually, e.g., to verify that the correct execution context has been taken into account.

Closely related is the issue of establishing a mapping between source and object code: If an object code-based annotation language is to be used to express the behavior of constructs of the original programming language it is necessary to establish some correspondence between the two. One possibility is to define a set of constructs that can still be recognized after compilation, such as loops or procedure calls. We will call these constructs *anchors*.

Programming language: Restricting the features of the programming language that can be handled is an important option when designing an annotation language in order to control the expressiveness, precision, and efficiency of subsequent WCET analyses using the language. In effect, this means to restrict the programming language to a subset. Annotation languages, for example, can be limited to *reduceable* code. Programming languages, however, are often constrained in another way, too: by the WCET calculation methods which are compatible with the annotation language at hand. Similarly, also techniques for the automatic calculation of flow information impose often further restrictions on the programming language. Floating point operations, for example, might not be supported by an annotation language.

Besides this, it is another important issue if the annotation language supports path analysis of the object code. This is crucial because it imposes additional challenges compared to path analysis at the source code level. Different from object code, for example, source code typically uses high-level control-flow statements which allow for a simple calculation of the control-flow graph (CFG). For object code, additional annotations are necessary to reconstruct the CFG precisely.

2.2 Usability

The usability of an annotation language is possibly best reflected by the skills and the amount and the complexity of work it demands from a programmer when using it. It also refers to the knowledge that is required beyond the annotation language itself, e.g. about the WCET analysis expected to make use of it, maybe even of the implementation specifics of this technique as it might affect its performance. Similarly, this holds for the amount of work required to update a program annotation in response to an update of the program. Another issue referred to concerns the ability to cope with annotations that are automatically provided by a tool.

In principle, there are two potential classes of users that provide code annotations: a) programmers, who write manual code annotations, and b) tools that calculate annotations by means of code analysis.

Considering manual code annotations it is quite important that the program behavior can be described concisely and compactly. As an extreme case, the size of an annotation describing a specific program property, may grow exponentially with the program size. When using automatic techniques to calculate code annotations, it is important whether the techniques are capable to produce information in a format which is supported by the annotation language.

When post-processing the calculated WCET results, it is an important issue whether the WCET calculation methods compatible with the annotation language are able to provide the user with information explaining the WCET results. Standard use of ILP (cf. Section 3.4.3) with flow constraints, for example, can only provide information about the execution frequency of statements, but does not provide any information on the execution order.

The preceding discussion shows that usability is the outcome of the interplay of several factors, in particular, of the complex interplay of the annotation language and the possible support for applying this language that is provided by the (tool) environment it is used in. Assessing the usability of an annotation language thus implicitly amounts to an assessment of its usability with respect to a specific global environment, which might even change over time. This, however, is beyond the scope of this paper. For the purpose of this paper, we will thus additionally use a second term, which we call *intricacy* of an annotation language. In distinction to usability, which is the broader, the more general term, intricacy is the more specific one. We refer to intricacy to assess the language-inherent conceptual and technical complexity of an annotation language, detached from any environment or tool support of using it.

2.3 Tool Support

As mentioned above, the availability of a tool using a specific annotation language can be considered an indicator of the general usefulness and usability of this language. In particular, it is an information which we consider valuable on its own. In general, we believe that the efficiency of the known WCET calculation methods, which are compatible with an annotation language, is one of the most relevant factors driving the development of tools. It is also worth noting, however, that vice versa the efficiency of a specific WCET calculation method depends much on the specifics of the underlying annotation language. Obviously, this holds for annotation languages, which require the program structure to be unrolled in order to make the code annotation applicable. We would like to remind the reader that we are not aiming at assessing the quality of tools in this paper.

3 WCET Fundamentals

In this section we recall the essentials of flow information and of WCET calculation methods. This provides the foundation for reviewing the annotation languages we selected as prototypical representatives of the different annotation concepts. To enable this, we first divide the different kinds of flow-information into three types (Section 3.1), and then characterize the flow information which must be supported at a minimum by any reasonable annotation language (Section 3.2). The precision of flow information is limited by the expressiveness of the annotation language (Section 3.3). Subsequently, we describe the essence of the fundamental WCET calculation methods used in practice (Section 3.4).

3.1 Types of Flow Information

The static description of a program's control flow is given by its control-flow graph and its call graph. To calculate the WCET of a program also information about the dynamic control-flow behavior is needed. In WCET analysis, flow information about the dynamic control flow is typically used to partially describe the following program behavior:

Explicit execution frequency. This type of flow information describes the execution count of nodes or edges of the control-flow graph. Execution count information can be given, for example, as the absolute execution count of a code location, or as a relation between the execution count of one code location and another code location. In practice, information on execution frequency is formulated as linear equations between the execution count of different code locations.

Explicit execution order. This type of flow information is concerned with describing patterns of execution order of nodes or edges of the control-flow graph. The execution order of statements is significant on modern processors where the execution time of an instruction depends on the execution history.

Context-sensitive flow information. This refers to the control flow of instructions that may be executed multiple times within a program execution. In greater detail, we can distinguish two sources of context-sensitive flow information: instructions executed within a loop and instructions executed within a function which is called multiple times.

- *Loop-context sensitive* flow information describes the control-flow behavior of a loop body for a subrange of all possible loop iterations.
- *Call-context sensitive* flow information describes the control-flow behavior of a function for specific call sites.

3.2 Minimal Flow Information

Besides the control-flow description of a program, the only additional flow information mandatory to bound its WCET are boundaries of the execution frequency of cycles in the description of the syntactical control flow. For intra-procedural WCET analysis, the *control-flow graph* (CFG) is used as control-flow description; for inter-procedural WCET analysis the *super graph* is used, which is a combination of the call graph and the CFG of each subroutine.

In case of reducible loops [1] so-called *loop bounds* are used to describe the maximum iteration count of loops. Annotating other cyclic control-flow like recursive function calls or non-reducible loops is less intuitive.

3.3 Completeness of an Annotation Language

The *completeness* of an annotation language is concerned with the question of how precise the set of feasible control-flow paths of programs can be described by a flow annotation language. A path annotation language is *complete* if it is expressive enough to describe for arbitrary programs the *feasible paths* and the *infeasible paths* as two disjoint (non-overlapping) sets of paths, i.e., feasible and infeasible paths can be described precisely instead of having to use any approximations. For this definition of *completeness* it is sufficient to assume an implicit description of the infeasible paths by the inverse of the set of feasible paths. Given a flow annotation language which does not allow to describe the set of feasible paths precisely, one has to use over-approximations, representing a superset of the set of feasible paths. Lacking completeness in the flow description will generally result in an overestimation of the WCET.

3.4 WCET Calculation Methods

The type of interesting flow information depends much on the applied WCET calculation method. In the following we recall the three most important WCET calculation methods.

3.4.1 Timing Schema

The *timing schema* approach turned out to be an efficient WCET calculation method that is also very simple to implement [24, 26, 23]. Essentially, the *timing schema* consists of hierarchical WCET calculation rules for each node of the syntax tree representing elementary or composed statements. Denoting the local WCET bound of a node A by $T(A)$, the local WCET of the sequential composition $A;\ B$ of two nodes A and B is computed as $T(A)+T(B)$. Analogously, the local WCET of a conditional statement `if` A `then` B `else` C `fi;` is computed as $T(A)+max\,(T(B), T(C))$, while the local WCET bound of a loop `while` A `do` B `od;` with at most LB iterations (loop bound) is computed as $(LB+1)\cdot T(A) + LB \cdot T(B)$. Last but not least, if A represents an elementary statement, $T(A)$ is simply the maximum execution time of A. Of course, the *timing schema* can analogously be formulated to calculate the best-case execution time. The computational complexity of the *timing schema* is linear with the program size. It can thus be applied efficiently to large programs.

A refinement of the *timing schema* approach to handle nested loops more precisely has been presented by Colin and Puaut [6].

3.4.2 Path-Based WCET Calculation

Path-based WCET calculation [10, 27] is inspired by the naive approach of analyzing each program path and selecting the longest out of it as the WCET bound. Though this approach is infeasible for the whole program, it becomes realistic for local scopes of a program. Thus, the idea of path-based WCET calculation is to search for the longest path within each innermost scope. For example, each loop could form a scope. Once the longest path of a scope has been determined, the whole scope is treated as a single instruction with the execution time of the longest path assigned to it. This procedure is repeated till the whole program is analyzed.

Path-based WCET calculation has been developed to analyze the effects of pipelines. It allows to model the impact of the pipeline to an instruction sequence longer than just basic blocks, and thus increases the precision of the WCET bound. However, path-based WCET calculation is inappropriate to take rather global timing effects into account, like cache behavior.

3.4.3 IPET-Based WCET Calculation

The *implicit path enumeration technique* (*IPET*) has been introduced by Li and Malik [17], as well as by Puschner and Schedl [25]. In contrast to path-based WCET calculation where paths are explicitly enumerated, IPET performs an implicit longest path search.

The basic idea is to model the control flow of the program by constraints. To reduce the complexity, typically only linear constraints are used, i.e., the program is represented as an *integer linear program* (*ILP*). Subsequently to this basic modelling, supplemental flow information can be included smoothly as additional constraints of the ILP problem. The finally formulated ILP problem is passed to an ILP solver that computes the desired WCET bound. Due to the broad availability of commercial and open-source ILP solvers, such ILP problems can be solved conveniently.

4 WCET Annotation Languages

Together with Section 5 and Section 6, Section 4 represents the core of this paper. In this section we reconsider a selection of prototypical representatives of the different annotation languages (Section 4.1 to 4.7). The findings of this reconsideration will then be the basis of our conceptional comparison of these languages in Section 5.

4.1 TAL - Equations with Event Markers

Mok et al. describe the *Timing Analysis Language* (*TAL*) [20]. This is an integral part of the timing analysis system developed at the University of Texas. The timing analysis system uses the *timing schema* approach and consists of several tools retrieving information that is to be used as input for the *timetool*, which eventually performs the calculation of the execution time of the analyzed program.

While *timetool* itself works only on assembler code, the tool set also contains a modified C compiler to translate annotated C programs to annotated assembler programs. The annotations of the C code are automatically generated by the *annotate* tool that fills in default assumptions about the program's behavior. The compiler generates the annotations of the assembler code in form of a TAL script. Usually, this script is not yet useful for the analysis since it contains too conservative estimates. It has thus to be refined by a more powerful tool or by hand to get better results. To aid the user with this task, a graphical user interface is provided.

Finally, the script is interpreted by *timetool* to calculate the execution time of the program. A very detailed description of the language can be found in [5].

Figure 1 shows a simple C-program taken from [20] that will serve us as an example. The automatically

```
main() {                    // -v-L1:
  int i=0, j=0; 3
  while (i < 100) {         // -v-L3:
    if (i < 10) j++;
    i++;
  }                         // -^-L6:
}                           // -^-L7:
```

Figure 1. Example C-Source

generated TAL-script is displayed in Figure 2. The script contains references to labels that occur in the assembler output of the compiler. We have also inserted the locations of these labels into the C-source in Figure 1.

```
func TAL_main() {
  block blk1;
  loop lp1;

  blk1#begin = "-v-L1";
  blk#end   = "-^-L7";

  lp1#begin = "-v-L3";
  lp1#count = MAXINT;
  lp1#end   = "-^-L6";

  return(blk1#time);
}
```

Figure 2. Autogenerated TAL-Script for the program in Figure 1

As can be seen in Figure 2, the language offers the following data types for timing purposes: A *loop* describes a loop construct where the execution frequency depends on the data being processed. A *block* is a program fragment that may contain loops, but the execution time of the block must be fixed. The language then also defines an *action*, which is any larger program fragment whose execution time is of interest. TAL distinguishes primitive and composite actions.

Each object is associated a set of attributes, such as the `time` and (loop-)`count` expressions. The syntax of assigning an attribute is `object#attribute = expression`.

In the example, there are two obvious modifications a programmer can be expected to make to the autogenerated script: First, replacing `MAXINT` as loop count attribute of `lp1` by a more accurate value.

```
9   lp1#count = 100;
```

Second, parameterizing function definition and changing the calculation formula in the last line of the script to reflect the fact that the inner `if`-statement is executed only ten times:

```
1 func TAL_main(if_count)
```

and

```
12   return (lp1#count - if_count)*blk#2time;
```

It is an interesting feature of the annotation language that it allows to specify nearly perfect execution time bounds, since the formula may contain almost any expressions. This creates new responsibilities for the programmers, who have to devise the correct calculations on their own.

4.2 Path Language and IDL

Park and Shaw proposed a WCET analysis for a subset of the C language, compiled by the GCC compiler for the MC68010 processor [26, 23, 21, 22]. They also developed the *timing schema* recalled in Section 3.4.1 for calculating the WCET of a program.

4.2.1 Path Language (PL)

Park and Shaw took much care in order to allow the specification of (in)feasible program paths. They developed a so-called *path language* (in the following called *PL*) based on regular expressions, which is shown in Figure 3. The basic idea is to label instructions interesting for path characterization with labels. Using PL one can describe path patterns, representing a set of paths.

`path` ::

  a regular expression of `symbols`

`symbols` ::

  alphabets(Σ) : a set of code labels

  operators : $+$, $\cdot$, $\star$, $\cap$, $\neg$

  parenthesis : (,)

  empty set : $\emptyset$

  wild cards :

    $*$ . . . arbitrary string of labels: $(\Sigma)\star$

    _ . . . any string of code labels not containing

      its surrounding labels,

      i.e., $x\_y = x(\Sigma - \{x, y\}) \star y$

      '_' may be also used as unary operator:

      $\_y = (\Sigma - \{y\}) \star y$, $x\_ = x(\Sigma - \{x\})\star$

Note the difference between the Kleene star '$\star$' and the wildcard '$*$'!

Figure 3. Path language based on regular expressions [22]

Multiple occurrences of a pattern can be abbreviated, e.g., A^{2-4} is a short hand for $AA+AAA+AAAA$.

Using this convention, it can be easily expressed, for example, that a loop, where the beginning of the body is labelled LB, has an iteration count of at most 10: $\_(LB\_)^{0-10}$.

A key feature of PL is that it allows to describe patterns of explicit execution order of labeled statements. In fact, it is complete, i.e., it allows to describe all paths of terminating programs. To specify the PL expression of a given program, one has to instantiate the possible shape of input data, i.e., one has to take into account the possible valuations of input data.

A drawback of PL is that even common path patterns can result in very long expressions. For example, linear flow constraints like $f_i < f_j$ (i.e., control-flow edge f_i is executed less frequently than edge f_j) can only be described by explicitly enumerating all possible path combinations containing f_i and f_j.

Path analysis based on regular expressions can be rather computation-intensive. Every path information I_i represents a set of paths IP_i. The path analysis is done by intersecting the set of syntactically possible paths AP with the set of paths described by all the path informations $IP = \bigcap_i IP_i$. The set of feasible paths XP is then calculated as

$$XP = AP \cap IP$$

The problem with this approach is that the central path processing operations $\neg$ and $\cap$ require exponential time in general [19]. Park and Shaw thus found that PL in its generic form results in expressions too complex for path processing. Therefore, they complemented PL with a higher level *information description language*, which we are going to recall next.

4.2.2 Information Description Language (IDL)

In order to overcome the deficiencies of PL, Park et al. developed the *information description language* (*IDL*), which can be translated into a structured subset of PL [22]. For example, the information that label A and B can only be executed together is expressed in IDL as samepath(A,B). This is translated to the equivalent low-level PL-expression $(*A*) \cap (*B*) + \neg(*A*) \cap \neg(*B*)$. As another example, a loop of scope A with constant iteration count K is written as loop A K times. This is translated into the low-level expression $\neg(*A*) + (\_A.entry\_A.body(\_A.body)^K) \star \_$. This transformation of loop information also illustrates the difficulty of getting descriptions using low-level regular expressions right. In fact, the original transformation given in [22] is faulty, as it does not take care of the case that a loop may be nested within another loop. The original translation was like $\neg(*A*) + \_A.entry\_A.body(\_A.body)^K\_$, which in case of nested loops would erroneously exclude paths with multiple executions of the loop.

The strength of IDL (as well as of PL) is that they both allow to describe path patterns of explicit execution order. However, IDL inherits a significant weakness from PL: information about relative execution frequencies of code can only be expressed by explicitly enumerating all possible path patterns, which can be of exponential length. An example illustrating this phenomenon is shown in Column 4 of Table 2 (Benchmark B2).

4.3 Linear Flow Constraints

Linear flow constraints are used in the context of IPET WCET calculation methods. The general ILP problem representing the program execution consists of n decision variables x_1, ..., x_n, an objective function $Z = \sum_{i=i}^{n} c_i \cdot x_i$ that has to be maximized, m functional constraints $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ for all $i \in [1, m]$ with a_{ij} being integer constants, and the non-negativity constraints $x_i \geq 0$.

To model the WCET calculation as an ILP problem, the static program structure is reflected by the control-flow graph $G = (V, E)$, having a unique start node $s \in V$ and a unique termination node $t \in V$. The execution time of each edge $\langle i, j \rangle \in E$ is denoted by $t_{i,j}$. Denoting the execution frequency of edge $\langle i, j \rangle \in E$ as $f_{i,j}$, the WCET of a program P is given by the following objective function to be maximized:

$$wcet(P) = max \sum_{\langle i,j \rangle \in E} f_{i,j} \cdot t_{i,j}$$

The key idea to map the WCET calculation problem onto the general ILP problem is formulating the CFG structure as flow equations. For that purpose, the structure of the CFG is represented as functional constraints in the ILP problem. The CFG resulting from the source code of benchmark B_1 and B_2 (Table 2) is used as example CFG within this section. For each node exactly one flow equation is generated stating that the sum of the execution frequencies of incoming control-flow edges equals the sum of the execution frequencies of the outgoing edges. For instance, for node 5 this equation is $f_{4,5} = f_{5,6} + f_{5,8}$. To model that the program is executed exactly one time we set the frequency of the back edge $\langle 14, 1 \rangle$ to one, i.e., $f_{14,1} = 1$.

To get the WCET bound, an ILP solver is used to calculate the length of the longest possible path through the CFG. However, in the CFG the length of this path is not bounded due to the cycle introduced by the back edge $\langle 12, 4 \rangle \in E$ of the loop. Thus, it is required to add a constraint limiting the iteration count (and thus the frequency $f_{12,4}$). This is accomplished by adding an additional constraint of the form $f_{12,4} \leq LOOP\_BOUND \cdot f_{2,4}$. This so-called *loop bound* is a mandatory flow fact to calculate a WCET bound.

After this step, the obtained model can be solved by an ILP solver. There exist many implementations of such solvers, for instance the *GNU Linear Programming Kit* (*GLPK*). Figure 4 shows the resulting ILP problem of benchmark B_1 and B_2 (Table 2). The corresponding $t_{i,j}$ represent the execution times of the edges $\langle i, j \rangle \in E$ and are the coefficients of the respective $f_{i,j}$ within the objective function (in this example, for all edges $\langle i, j \rangle \in E$ it holds that $t_{i,j} = 1$).

```
Maximize

etime: 1 f1_3  + 1 f3_4 + 1 f4_5 + 1 f5_6  + 1 f5_8  +
       1 f6_9  + 1 f8_9 + 1 f9_10 + 1 f10_11 + 1 f11_12 +
       1 f10_12 + 1 f12_4 + 1 f4_13

Subject To
  f13_1  - f1_3                   = 0
  f1_3   - f3_4                   = 0
  f3_4   + f12_4  - f4_5 - f4_13 = 0
  f4_5   - f5_6   - f5_8          = 0
  f5_6   - f6_9                   = 0
  f5_8   - f8_9                   = 0
  f6_9   + f8_9   - f9_10         = 0
  f9_10  - f10_12 - f10_11        = 0
  f10_11 - f11_12                 = 0
  f10_12 + f11_12 - f12_4         = 0
  f4_13  - f13_1                  = 0

  f13_1 = 1  \* Artificial back edge *\
  f12_4 - 100 f3_4 ≤ 0 \* Loop bound *\

  f5_6 - f10_11 = 0 \* Constraint B1 *\
  f5_6 - f5_8 ≤ 0 \* Constraint B2 *\
End
```

Figure 4. ILP problem for the CFG of benchmark B_1 and B_2 in Table 2.

This example illustrates that flow facts are indispensable for providing a limit on the number of the loop iterations whenever loops are present. An example for an annotation language that allows to express linear flow constraints is WCETC [15]. Another annotation language, being used within a tool that automatically extracts control-flow information and constraints from a program, has been proposed by Engblom et al. [7]. This annotation language is discussed next.

4.3.1 Modeling Contexts within Flow Constraints

The annotation language developed by Engblom and Ermedahl allows to represent flow facts over all iterations of a loop as well as over some specific iterations [7]. In particular, it also allows to specify flow facts for irreducible control flow. For WCET analysis the flow facts are converted to a format suitable for a WCET calculation method based on the *implicit path enumeration technique* (*IPET*) (cf. Section 3.4.3). Engblom and Ermedahl assume that flow-analysis is performed prior to low-level analysis, meaning that flow analysis does not have access to information about the execution time of code. The outcome of the flow analysis is a set of statically feasible paths. The WCET calculation uses information about the execution time of each piece of code to find the paths in the set of statically feasible paths that correspond to the actual worst-case execution times.

To represent the dynamic behavior of a program Engblom and Ermedahl introduce the concept of a scope. A scope has a header node that dominates all nodes in the scope and corresponds to a certain repeating execution environment, such as a function call or a loop. All scopes are supposed to be looping, even if they just iterate zero or one time. Each scope is represented by a set of nodes and edges. Scopes are connected by edges according to the control flow in the program. Every scope has a set of associated flow information facts. A flow information fact consists of three parts: i) the name of the scope, where the fact is defined, ii) a context specifier, and iii) a constraint expression. A context specifier allows to specify the iterations of the scope in which the constraint expression must be valid. The specifiers are defined using two dimensions of type and iteration space. The type allows to specify that the fact is considered a sum over all iterations, or for each single iteration separately. The iteration space is the set of iterations of the scope it is valid for. This can either be all iterations or some specified range. The flow information specified by annotations is converted to a form appropriate for IPET by mapping the scope-local semantics to execution-global semantics.

4.4 Data Value Assertions used in SPARK Ada

Chapman et al. described a WCET analysis for SPARK Ada [3, 4], the programming language used in the *Spark Proof and Timing System* (*SPATS*). The SPARK[1] language is a subset of Ada83 that is extended

[1]SPARK is an acronym for *SPADE Ada Kernel*, where SPADE is a short hand for *Southampton Program Analysis Development Environment*.

by a special kind of comments. The annotations are used for both program proof and timing analysis. Like the program proof framework, the WCET calculation in SPARK Ada is based on *symbolic execution*.

The edges of the control-flow graph of the input program are provided with weights that describe the execution time of the corresponding instructions. To keep flexibility, the weights in the CFG are given in the form of symbolic expressions instead of specific timing values; this representation has the advantage of being independent of the target hardware.

The static semantics of the SPARK language ensures that the programmer places at least one assertion before every loop statement as well as preconditions and postconditions to each function. These assertions are called *cut points*. Thus, the control-flow graph can be decomposed into a set of cut points and *basic paths* connecting them.

The problem of finding the WCET is equivalent to finding the longest path of the extended control-flow graph, which can be solved by a simplified version of the algorithm described by Tarjan [29]: through a set of transformation rules, an acyclic directed graph is mapped to a regular expression that is used to find the shortest path. The dual problem is considered in SPARK. To handle loops, a special bounded iteration operator is included in the regular expression syntax. Chapman gives three graph rewriting rules [3] to collapse alternatives, inner loops and outer loops to a simplified graph containing fewer edges, but more complex regular expression as weights. SPARK Ada expects the programmer to supply annotations for the loop bounds.

Figure 5 shows an example taken from [4] of a program calculating the power function. A distinct feature of the language is the inclusion of *modes*. SPARK Ada allows the user to specify multiple behaviors for a function that may be called from different contexts or with different input values. For each mode, the user can specify a distinct set of annotations; thereby enabling a more precise analysis.

Due to the nature of the annotations, however, it is not possible to specify tight bounds for nested loops, where the iteration space of the inner loop depends on the state of the outer loop.

```
--# proof function pow(FLOAT,INTEGER) return FLOAT;
function POWER(BASE: in FLOAT;
               EXPONENT: in INTEGER) return FLOAT
--# pre true;
--# mode A (EXPONENT >=0);
--# mode B (EXPONENT < 0);
--# post (POWER = pow(BASE,EXPONENT));
is
  ONE: constant FLOAT := 1.0;
  EXCHANGE: BOOLEAN;
  L_RES: FLOAT;
  L_EXP: INTEGER;
  RESULT: FLOAT;
begin
  L_RES := ONE;
  if EXPONENT ≥ 0 then
    EXCHANGE := FALSE;
    L_EXP := EXPONENT;
  else
    L_EXP := -EXPONENT;
    EXCHANGE := TRUE;
  end if;
  --# loopcount(L_EXP);
  loop
    --# assert
    --# ((not EXCHANGE) -> (L_RES = pow(BASE,(EXPONENT - L_EXP)))) in A;
    --# & (EXCHANGE -> (L_RES = pow(BASE,(-EXPONENT - L_EXP)))) in B;
    exit when L_EXP = 0;
    L_RES := L_RES*BASE;
    L_EXP := L_EXP-1;
  end loop;
  if EXCHANGE = TRUE then RESULT := ONE / L_RES;
  else RESULT := L_RES; end if;
  return RESULT;
end POWER;
```

Figure 5. An example of an annotated SPARK Ada program as given in [4]

4.5 Symbolic Annotations

Blieberger proposed an approach, which combines aspects of a pure annotation language with those of a programming language extension [2]. The clue of this approach is the invention of so-called *discrete loops*. Discrete loops can be considered a generalized kind of for-loops. Discrete loops allow a very flexible update of the loop-variable, much more flexible as for a for-loop. Nonetheless, like for for-loops, also for discrete loops the loop bounds can often automatically be computed by means of reasonably simple mathematical reasoning. Particularly well-suited for this purpose are methods for symbolic analysis. We thus coin the term *symbolic annotation* for this approach here.

The following program fragment illustrates the essence underlying the concept of discrete loops:

```
k:= ...;
discrete h := k in 1..N/2
        new h := 2*h | 2*h+1 loop
<loop body>
end loop
```

Marked by the new key word `discrete` the expression following the initialization of the loop variable `h` specifies the range both the initial value of `h` as well as all other values of `h` during subsequent iterations of the loop must be inside. Once the value is outside of this range, the loop terminates. This captures the language extension portion of this concept. The annotation language portion is captured by the term following the keyword `new`. This term specifies the set of legal values of the loop variable of immediately adjacent loop iterations. In the example above, the new value must be either the result of doubling the old value (`2*h`), or the increment of this value (`2*h+1`). The semantics given to discrete loops requires that these constraints are validated at compile-time, or checked at run-time, if the former fails.

A very appealing feature of this approach is the seamless integration of the annotation and the program source text. This elegance, however, comes at the cost that algorithms, whose textbook version may often make deliberate use of arbitrary loops, have to be adopted or replaced by newly invented algorithms which comply with the programming discipline imposed by discrete loops. Depending on the algorithmic problem, this can be natural and easy, but sometimes also difficult and challenging, or impossible at all.

4.6 The Annotation Language of Bound-T

In [13], Holsti et al. introduce *Bound-T*, an industrially available WCET tool originally developed by Space Systems Finland Ltd and currently marketed by Tidorum Ltd. Bound-T operates on the object-code level and relies on debug information and additional assertions provided by the programmer.

The analysis is performed in three distinct phases. First, a control-flow analysis is performed to construct the call-graph of the program. The WCET calculation is then performed bottom-up on the call-graph. Bound-T cannot handle cyclic call graphs.

In the next step, iteration bounds for the loop constructs in the program are calculated. In some cases these bounds can be found by the data flow analysis that is implemented in Bound-T. In this step, the semantics of the loop body is expressed as the functional composition of the effect of the individual statements, which are expressed in Presburger arithmetic, a decidable subset of integer arithmetic. On this basis, loop increments can be found and thus can be the bounds for all counter-based loops. If Bound-T is unable to bound a loop automatically, the user is prompted to provide an *assertion* containing the loop bound. The tool will emit a warning for each instance, together with the context of the loop in question. The assertions are placed in an additional file. The decision for the external annotation is motivated by the need to support multiple execution contexts for each function. Once the call-graph has been constructed and the loop bounds have been found, the actual worst-case execution path is searched for.

On modern processors, the execution time of a particular instruction depends on the history of instructions that have previously been issued. Bound-T handles this by simulating the processor pipeline. It does not, however, model any cache behavior. The calculation is performed by transforming the analysis data into an ILP problem which is then passed to the `lp_solve` tool.

The assertion language was conceived to be flexible enough to be used with programs written in both high-level languages and assembler. Assertions are stated for a specific *scope* (= subprogram, loop or call) which are identified through their respective name or - in the case of loops - their syntactical structure. This allows for characterizations such as loops being nested inside other loops and loops calling a particular subprogram. An example of such a characterization is given in Figure 6.

```
loop that
   is in (loop that calls "Foo")
   and contains (loop that not calls "Bar"
   and calls "Fee")
   and not contains (loop that calls "Fee2")
repeats 10 times end loop
```

Figure 6. An example of a Bound-T annotation as given in [13]

While the annotations are conceptually driven by the sources, they are referentially depending on the object code. Through this design decision, Bound-T theoretically gains language and compiler independence by using the object code as a basis for its calculations, but there are also limitations that arise with this approach: Because of the optimization steps performed by the compiler, the annotations are restricted to reference only program *features* that can still be recognized after compilation. We will call these features (such as calls and loops, but not if-then-else statements) anchors. A detailed description can be found in [14]. In [12], the author of Bound-T also stated that a better mapping between source and object code is planned for a future revision of the language.

4.7 The Annotation Language of aiT

Like Bound-T, the aiT WCET tool is a commercially available tool for WCET analysis. It is developed by AbsInt Angewandte Informatik GmbH, Germany, and is available for different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555. The aiT tool reads binary files as input programs to be analyzed. To make this more effective, the tool supports a special kind of object code annotations to reconstruct the control-flow graph from the object code [8, 11]. They allow, for example, the user to annotate the possible targets of a jump instruction in order to guide the object-code parser when reconstructing the flow graph.

aiT is not included in Table 2 since the focus of this paper is on the annotation of feasible paths in general. It is worth noting that the aiT tool also includes a value analysis to automatically calculate some flow information.

5 Discussion

In order to evaluate the annotation languages reconsidered in Section 4 we created benchmarks (Table 2) covering the criteria developed in Section 2. These benchmarks highlight the features and restrictions of the respective languages.

Each benchmark B_i consists of a source program and additional flow information that is specified informally. The first column of Table 2 describes for each benchmark the flow information to be annotated. The original source codes subject to annotation are given in the second column of Table 2, their control-flow graphs in the third column. The annotated examples for each annotation language are presented in the subsequent columns.

In the following, the most interesting results of applying the annotation languages of Table 2 to these benchmarks are presented. Table 1 provides a summary of our findings.

5.1 Expressiveness

In order to assess the expressive power of the annotation language, it is necessary to understand the calculation method that is implemented by the tool. We thus begin with an overview about the different methods and will then assess how these methods apply to the modelling of different kinds of flow information:

Calculation method. Timing schema, as implemented by TAL, were the first approach to WCET analysis and provide little more than a unified framework for the programmer to specify timing calculations with.

This approach is refined by the graph-rewriting technique used in SPARK Ada. It must be noted, however, that the expressiveness of SPARK Ada is limited, since it restricts the permitted kinds of flow information to loop-bounds.

IPET-based methods that use linear flow constraints, are widely regarded as state-of-the-art and allow a more versatile constraint-based specification of flow facts. These constraints are then used as input for an ILP solver. IPET-based tools still allow the specification of loop-bounds, which can be transformed into constraints easily.

As a unique feature, PL and IDL model execution order naturally.

Loop-bounds. The minimal information necessary to perform WCET analysis is an upper bound for every loop construct; all discussed languages support this.

Triangle-loops. The IPET-based methods (linear flow constraints and Bound-T) allow to specify inequalities as further constraints in addition to loop-bounds. With this method, so-called *triangle-loops* – these are nested loops that follow a triangular pattern in the iteration space (i, j) – can be described precisely. In the case of Bound-T, annotations of triangle-loops are only possible when they contain an anchor to identify them, such as a call. These anchors are necessary to identify program fragments in the object code.

Calling context-sensitive annotations. If loop bounds depend on input parameters, the precision will benefit from a tailored annotation for each calling context. The parametrized calculation schema of TAL supports this through a functional abstraction. While Bound-T does not expose context-sensitive information to the annotation language, it is aware of context information during the automatic computation of loop bounds.

Loop context-sensitive annotations. As described in Section 4.3.1, in the presence of caches it is often beneficial to distinguish between the first and subsequent loop iterations when formulating annotations. Currently none of the surveyed languages directly supports this feature.

Application context-sensitive annotations. SPARK Ada has a unique feature called *modes* to describe multiple annotations for a function depending on different input parameters. While this is in some ways related to calling context-sensitivity, it is not entirely the same concept. We will thus call this feature *application context sensitivity.*

Execution order. Most WCET-calculation methods content themselves with estimates of the execution

Criteria	Annotation Language							
	TAL	*PL and IDL*	*Linear Flow Constraints*	*Bound-T*	*aiT*	*SPARK Ada*	*Symbolic Annotations*	*Challenge*
Expressiveness	Timing schema	Regular expressions	Constraint-based	Constraint-based	Constraint-based	Loop-bounds	Loop-annotations	
Loop-bounds	yes	yes	yes	yes	yes	yes	yes	yes
Triangle-loops	yes	no	yes	some	yes	no	yes	yes
Calling context	yes	no	possible	implicit	no	explicit	no	yes
Loop context	no	no	possible	no	no	no	no	yes
Appl. context	no	no	no	no	yes	yes	no	yes
Execution order	no	yes	no	no	no	no	no	yes
Intricacy of Annotations	high	medium to high	medium	medium	medium	low	low to medium	as low as possible
Annot. placement	External TAL-script	Ideally inside the source code	Ideally inside the source code	External file	External file; partially inside source code	Source code comments	Integral part of the source language	– Design Decisions –
Abstraction level								
Source code	no	yes	yes	no	yes	yes	yes	
Object code	yes	no	yes	yes	yes	no	no	
Program. language								
Implementation	As'mbler/C	C	-	C, Ada	As'mbler/C	Ada	Ada	
General Scope	-	Any structured language	Any structured language	Any structured language	Any structured language	-	Any structured language	
Tool available	yes	no	yes	commercial	commercial	yes	prototype	yes

$\underbrace{\hspace{6cm}}_{\text{See also Table 2.}}$

Table 1. Assessment summary

frequency of basic blocks. If the method supports the modelling of a complex hardware architecture[2], the execution order is equally important. In contrast to the PL and IDL, the IPET-based methods currently cannot be used to describe the execution order.

5.2 Annotation Placement and Abstraction Level

Placing annotations directly inside the source code is more convenient for the programmer, but may affect the readability of the program, especially in the case of library functions, which usually have many different call sites. One argument against an integration into the sources is that in an production setting, any modification of the source code may require a repeated audit.

The decision of the annotation placement is closely tied to that of the abstraction level. For obvious reasons, all surveyed tools that operate on the object code level (TAL and Bound-T) also choose to place the annotations in a separate file.

In general, the following three choices are available:

Abstraction	Placement	
Source Code:	inside	external file
Object Code:	(not practicable)	external file

Bound-T and TAL follow the approach to annotate the program at the object code level. This low-level representation gains independence from the compiler, but complicates the development phase where the source code is frequently changing. The interaction with the compiler is an important issue, as optimizations that change the control flow may invalidate the annotations.

5.3 Programming Language

If the WCET annotation is based entirely on the object code, the annotation language is theoretically independent of the original programming language. This advantage, however, is hardly exploited. For practical reasons, many of the surveyed implementations focus on subsets of the C language.

5.4 Intricacy of Annotations

As mentioned above, there is a trade-off between the expressiveness and the complexity of annotations.

[2] In complex hardware architectures the execution time of instructions depends on the execution history. Typical reasons for this behavior are instruction pipelines, instruction/data caches and processor parallelism.

TAL, for example, leaves many aspects of the WCET calculation to the user, who may specify almost arbitrary formulae within the script. While this approach would guarantee the highest precision, it also demands a high effort from the programmer, who has to devise the correct calculation schema with care. The annotation languages of aiT and Bound-T reduce the intricacy and support language constructs tailored to different kinds of flow information. At the other end of the spectrum, there is SPARK Ada, which is restricted to only loop-bounds that are annotated directly into the source code.

5.5 Availability of Tools

Most of the surveyed annotation mechanisms stem from an academic background, with aiT and Bound-T being notable exceptions; they are currently being marketed as commercial products. According to Praxis High Integrity Systems, we may still see a future release[3] of a SPARK Ada-based source code annotation language.

In closing of our discussion, the findings summarized in Table 1 illustrate that none of the annotation languages we considered uniformly outperforms its competitors, but instead have their own individual strengths and limitations. This became the more apparent, if we were to take further criteria into account, e.g., the possibility and ease of reconstructing the control-flow graph on the object-code level such that it precisely reflects its counterpart on the source-code level [16] or the consideration of application domains of annotation languages which go beyond pure WCET analysis. An approach for the latter has e.g. recently been proposed by Lisper [18]. Compared to the languages we considered in this paper, the language he proposes has a more state-oriented flavor. By its execution counters the language especially allows to express execution frequencies, similar to the linear flow constraints described above. In principle, the language could also be used to describe explicit execution orders, however, the resulting expressions will often be very complex.

6 The WCET Annotation Language Challenge

Reconsidering the annotation languages proposed and used so far for WCET analysis and opposing their key characteristics as summarized in Table 1 demonstrate that all these languages have their own specific profile of strengths and limitations. The demand for an annotation language, which combines the individual strengths of the known annotation languages, while simultaneously avoiding their limitations, is thus apparent. In Table 1 this demand is reflected by the right-most column denoted by "*Annotation Language Challenge*." It grasps the summarized strengths of the different annotation concepts. Developing a language (or an annotation concept), which enjoys this profile is the central challenge, which we derive from our investigation, and which we would like to present to the research community.

This challenge, however, is not the only challenge, which is suggested by the findings of our investigation. It is obvious that an annotation language and a methodology for computing the WCET of a program based on annotations given in this language are highly intertwined. Expressiveness delivered by an annotation language, which cannot be exploited by a WCET computation methodology, is in vain. Vice versa, the power of a WCET computation methodology cannot be evolved if the annotation language is too weak to express the needed information. This mutual dependence of annotation languages and WCET computation methodologies suggests two further challenges. Which annotation language serves a given WCET computation methodology best? And vice versa: which WCET computation methodology makes the best use of a given annotation language?

Of course, the meaning of "best" has to be made more precise to be practically useful. We argue that the underlying notion of the relation "better" has several dimensions, each of these leading to possibly different solutions. Besides parameters like ease of use, we consider the parameters of power and performance and the trade-off between the two most important.

Summing up, this results in the following *challenges*:

1. Finding an annotation language, which enjoys the individual strengths of the known annotation languages while avoiding their limitations.

2. Finding an annotation language, which serves a given WCET computation methodology best.

3. Finding a WCET computation methodology, which makes the best use of a given annotation language.

It is worth noting that these challenges can be considered on various levels of refinement, depending for instance on the notion of the relation "better" as discussed above. Thus, the challenges above represent a full array of more fine-grained challenges rather than exactly three individual challenges.

[3] http://www.praxis-his.com/sparkada/examiner.asp

1: Benchmark B_i | *2: Control flow graph* | *3: TAL*

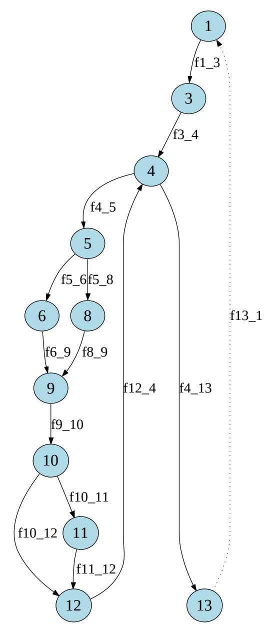
B_1: Explicit execution order

Side constraints: The conditions of the two if-statements at line 5 and 10 evaluate both to `false` or both to `true` within each iteration of the while-loop.

B_2: Explicit execution frequency

Side constraints: The execution frequency of the statement at line 6 is less or equal to that of the statement at line 8.

```
1 void cond(int a[],int b[])
2 {
3   int i=0, j=0;
4   while (i < 100) {
5     if (a[i] < 10)
6       j++;
7     else
8       a[i]=10;
9     i++;
10    if (b[i] < 10)
11      j++;
12  }
13 }
```



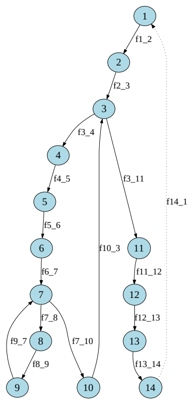
```
func TAL_cond(A_COUNT, B_COUNT) {
  block blk1, blk6, blk8, blk11;
  loop lp1;
  blk1#begin= "-v-LA_1";
  blk1#end  = "-^-LA_13";
  lp1#begin = "-v-LA_4";
  lp1#count = 100;
  lp1#end   = "-^-LA_12";
  blk6#begin= "-v-LA_6";
  blk6#end  = "-^-LA_7";
  blk8#begin= "-v-LA_8";
  blk8#end  = "-^-LA_9";
  blk11#begin = "-v-LA_11";
  blk11#end   = "-^-LA_12";
  return(blk1#time
   -(min(0,100-A_COUNT)*blk6#time
   -(min(100,A_COUNT) *blk8#time
   -(min(0,100-B_COUNT)*blk11#time);
}
```

Note: The side constraints for $\mathbf{B}_1$ and $\mathbf{B}_2$ are not directly expressible in TAL; the above script represents a best effort for both $\mathbf{B}_1$ and $\mathbf{B}_2$.

B_3: Subranges of loop iterations

In this benchmark, the challenge lies in expressing a triangular iteration pattern. (The program computes the sum $\sum_{k=1}^{n} k$)

```
1 int compute_sum(int n) {
2   int a=0, b=0, i=n, j, y;
3   while (i>0) {
4     a=a+1;
5     i=i-1;
6     j=i;
7     while (j>0) {
8       b=b+1;
9       j=j-1;
10    }
11  }
12  y=a+b;
13  return y;
14 }
```

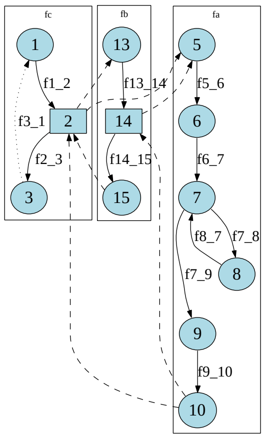


```
func TAL_compute_sum(N) {
  block blk1; loop lp3, lp7;
  blk1#begin= "-v-LA_1";
  blk1#end  = "-^-LA_14";
  lp1#begin = "-v-LA_3";
  lp1#count = N;
  lp1#end   = "-^-LA_11";
  lp2#begin = "-v-LA_7";
  lp2#count = N-1;
  lp2#end   = "-^-LA_10";
  blk7#begin= "-v-LA_7";
  blk7#end  = "-^-LA_10";
  return(blk1#time -
        N*(N-1)/2 * blk7#time);
}
```

B_4: Call-context sensitive flow information

Side constraints: The loop bound of the loop starting at line 7 is 10 when `fa()` is called from `fc()` and 7 when it is called from `fb()`.

```
1 int fc(int m, int n) {
2   return fa(m) + fb(n);
3 }
4
5 int fa(int i) {
6   int j=0;
7   while (j<i) {
8     j++;
9   }
10  return j;
11 }
12
13 int fb(int i) {
14  return 8 + fa(i);
15 }
```



```
func TAL_fc() {
  return TAL_fa(10) +
         TAL_fb(7);
}
/* To be called as TAL_fa(10); */
func TAL_fa(I_COUNT) {
  block blk5; loop lp7;
  blk5#begin = "-v-LA_5";
  blk5#end   = "-^-LA_11";
  lp7#begin = "-v-LA_7";
  lp7#count = I_COUNT;
  lp7#end   = "-^-LA_9";
  return(blk5#time);
}
func TAL_fb(I_COUNT) {
  return TAL_fa(I_COUNT);
}
```

Table 2: Flow Information Benchmarks and Annotation Examples, Part I

4: PL and IDL	*5: Linear Flow Constraints (*WCETC*)*	*6: Bound-T annotation*
PL: Loop-bound: $\neg(*L5*) + (\_L2(\_L5)^{100})\star\_$ **B$_1$:** $(*L6*)\cap(*L11*)+\neg(*L6*)\cap\neg(*L11*)$ **B$_2$:** *The flow relation between L6 and L8 would need full path enumeration!* **IDL:** Loop-bound: `loop` $L4$ `100 times` **B$_1$:** `samepath(`$L6, L11$`)` **B$_2$:** *The flow relation between L6 and L8 is not expressible!* Note: Lx means a reference to line x of the original code	Note: The side constraints for **B**$_1$ are not directly expressible in WCETC. **B$_2$:** `void cond (int a[], int b[]) {` `int i=0, j=0;` `WCET_SCOPE(s1) {` `while (i < 100) WCET_LOOP_BOUND(100) {` `if (a[i] < 10) {` `j++;` `WCET_MARKER(M1);` `}` `else {` `a[i]=10;` `WCET_MARKER(M2);` `}` `i++;` `if (b[i] < 10)` `j++;` `}` `WCET_RESTRICTION(M1 ≤ M2);` `} /* scope s1 */` `}`	**B$_1$:** `subprogram "cond"` `loop` `repeats 100 times;` `end loop;` `end "cond";` Note: Due to the lack of *anchor* features in the original program, a finer granularity is not possible. The side constraints for **B**$_2$ are also not directly expressible in Bound-T.
PL: $\neg(*L4*) + (\_L2(\_L4)^{0-n})\star\_$ $\neg(*L8*) + (\_L6(\_L8)^{0-(n-1)})\star\_$ $\neg(*L2\_L8*) + (\_L2(\_L8)^{\frac{n(n-1)}{2}})\star\_$ **IDL:** `loop` $L3$ n `times` *The inner loop has a variable loop bound, which is not expressible!* `execute` $L8$ $\frac{n(n-1)}{2}$ `times inside` $L3$`;`	`#define N 100 /* max. value of n */` `int compute_sum(int n) {` `int a=0, b=0, i=n, j, y;` `WCET_SCOPE(s1) {` `while (i>0) WCET_LOOPBOUND(N) {` `a=a+1;` `i=i-1;` `j=i;` `while (j>0) WCET_LOOPBOUND(N-1) {` `b=b+1;` `j=j-1;` `WCET_MARKER(M);` `}` `}` `WCET_RESTRICTION(M ≤ (N*(N-1)/2));` `} /* scope s1 */` `y=a+b;` `return y;` `}`	`subprogram "compute_sum"` `loop that contains (loop)` `repeats N_MAX times;` `end loop;` `loop that is in (loop)` `repeats N_MAX-1 times;` `end loop;` `end "compute_sum";` Note: This assumes that `N_MAX` is a known constant
PL: $\neg(*fb\_fa*) + (\_fb\_fa(\_L8)^{7})\star\_$ $\neg(*fc\_fa*) + (\_fc\_fa(\_L8)^{10})\star\_$ **IDL:** `loop` $L7$ `7 times inside` fb`;` `loop` $L7$ `10 times inside` fc`;`	`int fc (int m, int n) {` `return fa(m) + fb(n);` `}` `int fa (int i) {` `int j=0;` `/* specific loop bound for call context fb(fa()) is not supported */` `while (j<i) WCET_LOOP_BOUND(10) {` `j++;` `}` `return j;` `}` `int fb (int i) {` `return 8 + fa(i);` `}`	`subprogram "fa"` `loop` `repeats ≤ 10 times` `end loop;` `end "fa";` Note: According to [14], Bound-T is able to perform context sensitive analysis when loop bounds depend on function parameters. It is not possible to annotate context-sensitive information directly in Bound-T.

Table 2: Flow Information Benchmarks and Annotation Examples, Part II

In order to foster research on these challenges and to assess success, we consider the reference to a collection of benchmark programs which reflect the intricacies of annotating programs for WCET analysis, and of the interaction of annotation languages and WCET computation methodologies, most valuable. Ideally, these programs should be taken from real world applications, but stripped off from unnecessary detail; focusing on just the very essence to demonstrate where current annotation languages appear insufficient or inadequate to cope with. We are planning to set up a web page to host such a library of programs. In the long run we hope that this results in a research community maintained and accepted library of benchmark programs for assessing and evaluating the relative merits of annotation languages and WCET computation methodologies and combinations thereof. In spirit this is similar to the collection of benchmark programs proposed by the organizers of the WCET Tool Challenge [9, 28]. In fact, we consider it desirable to host such libraries in close relationship to each other.

7 Conclusions and Perspectives

The power, the generality, and the ease of use of tools for WCET analysis depend strongly on the kind and the expressiveness of the annotation language used to feed the tool with program-specific WCET information. The choice of the annotation language is the most crucial decision in the early stages of designing a WCET analysis tool. This choice is not trivial. The many conflicting properties an annotation language is desired to enjoy, e.g. expressiveness vs. ease of usage and analyzability, make the choice of a “good” language indeed a challenge of its own. It is thus by no means surprising that annotation languages attracted so much attention by researchers working on WCET analysis and that so many different approaches of annotation languages have been proposed and used so far for the implementation of WCET analysis tools.

In this paper we systematically reconsidered an array of prototypical approaches which we consider pathbreaking or especially successful and important for the advancement of the new and still fast developing field of WCET analysis. The evaluation of these approaches gives indeed evidence to our thesis that the definition of a “good” annotation language is a challenge. According to our findings, which are summarized in Table 1, none of the annotation languages turns out to be uniformly superior to its competitors, let alone to be without deficiency. As discussed in Section 5, this becomes the more apparent, if further criteria are taken into account such as the possibility and ease of reconstructing the control-flow graph on the object-code level (cf. [16]) or the consideration of application domains of annotation languages beyond pure WCET analysis (cf. [18]).

In spite of the indisputably successful use of so many conceptually diverse annotation languages for WCET analysis, all this indicates that the annotation languages proposed so far are still challenged in one way or the other. It is this observation, which yields the slogan and the invitation extended by this paper:

Contributing to

> overcoming the *challenged annotation languages*
> by mastering the *annotation language challenge.*

We consider the invention of an annotation language, which enjoys the profile outlined in the rightmost column of Table 1 entitled “Annotation Language Challenge”, as a milestone indicating the (partially) successful mastering of this challenge (and its variants). Particularly important for this will be advancements allowing a refined handling of contexts, execution orders, and interprocedural control-flow.

We believe that contributions towards mastering this new challenge will be essential for the next major step towards the further advancement of the field as a whole. The annotation language challenge complements the recently launched challenge for WCET tools [9, 28]. In fact, it is motivated by it in part. We believe that contributions towards mastering the annotation language challenge will also be a major step towards enabling the delivery of the prospects related to the tool challenge. Otherwise, the incompatibility of the annotation languages and the tools using them might soon turn out to be a significant obstacle for truly meaningful and in-depth comparisons of WCET tools.

Acknowledgments. The authors greatfully acknowledge the helpful comments of the anonymous referees and the feedback of the participants at the WCET’07 workshop. Especially, they would like to thank Niklas Holsti and Albrecht Kadlec for their many and very detailed comments, which helped to clarify and improve the presentation of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, June 1997. ISBN 0-201-10088-6.

[2] J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[3] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of spark ada. In *Proc. ACM Workshop on Language, Compiler and Tool Support for Real-time Systems*, pages K1–K11, June 1994.

[4] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[5] M. Chen. *A Timing Analysis Language - (TAL)*. Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual.

[6] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, June 2001. Technical University of Delft.

[7] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[8] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003.

[9] J. Gustafson. The WCET tool challenge 2006. In *Preliminary Proc. 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.

[10] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[11] R. Heckmann and C. Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.

[12] N. Holsti. Bound-t assertion language: Planned extensions. Technical report, Tidorum Ltd, 2005.

[13] N. Holsti, T. Långbacka, and S. Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000.

[14] N. Holsti, T. Långbacka, and S. Saarinen. *Bound-T timing analysis tool User Manual.* Tidorum Ltd, 2005.

[15] R. Kirner. The programming language wcetC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[16] R. Kirner and P. Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In *Proc. 5th International Workshop on Worst-Case Execution Time Analysis*, Palma, Spain, July 2005.

[17] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[18] B. Lisper. Ideas for annotation language(s). Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen, 2005.

[19] R. MacNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(39-47), 1960.

[20] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Worksop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.

[21] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs.* PhD thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02.

[22] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[23] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.

[24] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[25] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[26] A. C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[27] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[28] L. Tan and K. Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006. 13 pages.

[29] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[30] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, (Accepted January 2007).