# WCET Annotation Languages Reconsidered: The Annotation Language Challenge [1]

Albrecht Kadlec, Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan and Ingomar Wenzel

Institute of Computer Engineering, Institute of Computer Languages
Vienna University of Technology, Austria
email: {albrecht,ingomar,raimund}@vmars.tuwien.ac.at
{adrian,markus,knoop}@complang.tuwien.ac.at

**Abstract** Worst-case execution time (WCET) analysis is a prerequisite for successfully designing and developing systems, which have to satisfy hard real-time constraints. Of key importance for the precision and performance of algorithms and tools for WCET analysis are the expressiveness and usability of annotation languages, which are routinely used by developers for providing WCET algorithms and tools with hints for separating feasible from infeasible program paths.

Reconsidering and assessing the strengths and limitations of current annotation languages, we believe that contributions towards further enhancing their power and towards a commonly accepted uniform annotation language will be essential for the next major step of advancing the field of WCET analysis. To foster this development we have recently proposed the *WCET annotation language challenge*. This challenge complements the already earlier successfully launched *WCET tool challenge*. In this paper we summarize the essential features of current annotation languages and recall the WCET annotation language challenge derived from their assessment.

## 1 Motivation

The precision and performance of *worst-case execution time* (*WCET*) analysis depends crucially on the identification and separation of feasible and infeasible

[1] An extended version of this paper has been published in the Proceedings of the *7th International Workshop on Worst-Case Execution Time Analysis* (*WCET'07*).
This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis*, `http://costa.tuwien.ac.at/`,
the ARTIST2 Network of Excellence, `http://www.artist-embedded.org/`
and research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

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 for annotating this information and making it available for 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. [27]. Until recently, however, there was no approach towards a systematic comparison of the various approaches proposed on annotation languages for WCET analysis [14].

The goal of our approach of [14] was three-fold: (1) To identify an array of important universally valid criteria, in which the usefulness of annotation languages for WCET analysis becomes manifest. (2) 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. (3) 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 support the development of enhanced WCET algorithms and tools which will outperform their current counterparts for WCET analysis: The so-called *WCET annotation language challenge*.

As pointed out in [14], we believe that mastering the WCET annotation language challenge will be the key for further advancing the field of WCET analysis. Moreover, we believe that it will also be essential in order to enable the recently successfully launched *WCET tool challenge*, which has attracted the attention of many WCET tool developers [6,26], to unfold its strength and impact in full.

In this paper we summarize the essential findings of the comparison of an array of prototypical annotation languages presented in [14] and the conclusions drawn from this comparison.

## 2 Assessment Criteria

The criteria we use in order to assess the strengths and limitations of WCET annotation languages can be divided into two groups of *language design* and *usability* criteria. While the characteristics of the language design criteria are essentially under control when designing the language, the characteristics of the usability criteria are essentially an outcome of the characteristics of the language design criteria. In addition we consider a singleton third criterion, which is orthogonal to the other criteria. This is the existence of a *tool* using the annotation language. It is worth noting that the availability of a tool need not directly be related to a specific property or feature of an annotation language. In fact, there may be manyfold reasons why a tool has been developed, and vice versa, why not. Actually, these reasons need not necessarily be related to the

language at all. Nonetheless, we consider the availability of a tool an important indicator of the general usefulness and usability of an annotation language. We thus report the existence of tools, however, it is beyond the scope of this paper to assess the quality of any such tool. Readers interested in this might refer to the article by Wilhelm et al. [27].

Here, we proceed with an overview of the assessment criteria of annotation languages we use and which we discuss in more detail subsequently.

1. Language Design
   (a) Expressiveness
   (b) Annotation placement and abstraction level
   (c) Programming language
2. Usability
3. Tool Support

*Expressiveness.* We consider expressiveness of an annotation language the most important criterion at all. Intuitively, expressiveness refers to the capability of an annotation language to describe control-flow paths. Key for the expressiveness of an annotation language is the type of flow information it allows to describe. We call an annotation language *complete*, if it allows to precisely describe all feasible paths of arbitrary terminating programs. The capability of an annotation language to cope with inter-procedural program flow or selected iteration ranges of loops are other important aspects of expressiveness. Important setscrews a language designer can use to control the expressiveness of an annotation language are the means and their capabilities to deal with *loop bounds*, *triangle loops*, and, more generally, the *context sensitivity* of loop iterations and procedure or function incarnations, and the *execution order* of statements.

*Annotation placement and abstraction level.* The question of where to place annotations and at which level of abstraction has a strong impact on the usability of an annotation language because it directly affects the demands on a programmer when using a language.

First, it has to be decided if annotations shall be placed at the location of the source code statements they describe, or in a separate file? None of the two options is always superior over its counterpart. As a rule of thumb we have: If annotations are provided manually, it is usually more convenient to directly annotate the code. If annotations are computed automatically, it is often pragmatically advantageous to provide annotations in separate files.

Second, it has to be decided if the source code or the object code shall be annotated. Taking a (human-centered) usability perspective, annotating the source code appears generally preferable. This might be obvious, if code annotations are manually provided. However, it also holds, if flow information is automatically computed because it is often obligatory or at least desirable to verify automatically computed annotations manually, e.g., to verify that the correct execution context has been taken into account.

Closely related to this is the issue of establishing a mapping between source code and object code: If an object code-based annotation language is used to express the behavior of constructs of the original programming language it is necessary to establish a correspondence between the object code and the source code. This can be achieved e.g. by defining a set of so-called *anchors*, special language constructs, which can be recognized after compilation, such as loops or procedure calls.

*Programming language.* Restricting a programming language to a well chosen sublanguage and tailoring an annotation language towards this sublanguage is an important means to control the expressiveness, precision, and efficiency of a WCET analysis using this language. For example, an annotation language can be limited to *reducible* code. Also the WCET calculation methods which are compatible with an annotation language can constrain the features of a programming languages, which can meaningfully be handled. Another source, which can impose restrictions on the programming language, are the techniques for the automatic calculation of flow information. For example, a technique might not support floating point operations.

It is also an important feature of an annotation language if it supports path analysis of the object code. This is crucial because compared to path analysis at the source code level this imposes additional challenges at the object code level. For example, source code typically makes use of high-level control-flow statements which simplifies the construction of the control-flow graph (CFG) of a program. For object code, a precise (re-) construction of the CFG requires usually additional annotations.

*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 is also reflected by the knowledge which 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: Programmers writing manual code annotations, and tools automatically computing annotations by means of some code analysis.

For code annotations which are to be provided manually it is most 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. For code annotations which are automatically computed, it is important that the underlying techniques are able to deliver their information in a format which is supported by the annotation language.

It is also an important issue if a WCET calculation method which is compatible with an annotation language can provide the user with adequate information explaining its results. For example, *Integer Linear Programming* (*ILP*) with flow constraints as very often used in practice can only provide information about the execution frequency of statements, but not on their execution order.

All this shows that usability is the outcome of the interplay of several factors, in particular, of the complex interaction of an annotation language and the possible support for applying this language which 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. In addition to usability, we thus introduce a second more specific term, which we call the *intricacy* of an annotation language. We refer to this term in order to assess the language-inherent conceptual and technical complexity of an annotation language, detached from any environment or tool support of using it.

*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. We therefore report the availability of tools but we do not aim at assessing their quality.

## 3 WCET Fundamentals

We consider the general typology of current WCET calculation methods and the types of flow information they rely on as WCET fundamentals which we recall next.

*Types of flow information.* Intuitively, flow information provides a WCET calculation method with information about the dynamic behavior of a program. Typically, the (interprocedural) control-flow graph of a program is used to provide this information. The various kinds of flow information can roughly be classified as follows:

1. Explicit execution frequency
2. Explicit execution order
3. Context-sensitive flow information
   (a) Loop-context sensitive flow information
   (b) Call-context sensitive flow information

*Explicit execution frequency* information describes the execution count of nodes or edges of the control-flow graph. In principle, this information can be given as absolute execution count of a code location or as a relation between the execution count of one code location and another one. In practice, this kind of information is usually provided in terms of linear equations between the execution count of different code locations.

*Explicit execution order* information describes patterns of execution order of nodes or edges of the control-flow graph of a program. This information allows WCET calculation methods to cope with the intricacies of advanced modern processors, where the execution time of an instruction can depend on the execution history.

*Context-sensitive flow information* is relevant for reliably capturing the effect of instructions which may be executed multiple times within a program execution. In principle, two major sources of context-sensitive flow information can be distinguished: Instructions executed within a loop and instructions executed within a possibly recursive function or procedure which is called multiple times.

Two examples of concrete flow information are *loop bounds* and *recursion bounds*. Such bounds information is mandatory for any WCET calculation method. It can thus be considered the minimal flow information necessary for WCET analysis.

*WCET calculation methods.* WCET calculation methods can roughly be divided into dynamic and static techniques. Intuitively, dynamic methods are measurement-based and run the program to figure out the worst case execution time, whereas static methods are analysis-based and compute a bound for the worst case execution time of a program without running it. In this paper, we concentrate on static methods. The static methods can roughly be classified as follows:

1. Timing Schema Approaches
2. Path-based Approaches
3. Implicit Path Enumeration Technique (IPET) Approaches

*Timing schema* approaches operate on the *abstract syntax tree* (*AST*) of a program. Intuitively, each leaf of the tree representing elementary operations is assigned an execution time, each inner node an operation allowing to compute its execution time as a function on the execution time of its successor nodes. This directly induces a hierarchical approach for computing the worst case exection time of a program. Historistically, timing schema based approaches were among the first WCET calculation methods used in practice [22,24,20]. Refinements of these approaches e.g. towards an improved handling of nested loops have been proposed more recently [3]. The popularity of the timing schema approaches is in part due to their conceptual simplicity, which simplifies their implementation.

Unlike timing schema approaches, *path-based* approaches decompose a program into fragments. For each of the fragments they determine a program path with maximum execution time [7,25]. These times are then combined to the worst case execution time of the program. Path-based approaches have been developed for capturing the effects of pipelines, however, they are less appropriate for taking global timing effects into account, like cache behavior.

*Implicit path enumeration technique* (*IPET*) approaches perform an implicit search for the longest path of a program without enumerating paths explicitly [16,23]. This distinguishes them from path-based approaches. Intuitively, IPET

approaches model the control flow of a program by constraints. Typically, only linear constraints are used in order to reduce the complexity of solving the resulting constraint problem. This leads to an *integer linear program* (*ILP*), which can be solved by off-the-shelf open source or commercial ILP solvers.

## 4 WCET Annotation Languages

In this section we recall the essential features of the seven annotation languages, which we selected as prototypical representatives for our conceptual comparison of WCET annotation languages.

1. The Timing Analysis Language TAL
2. The Path Language PL and Information Description Language IDL
3. Linear Flow Constraints
4. SPARK Ada
5. Symbolic Annotations
6. The Annotation Language of Bound-T
7. The Annotation Language of aiT

In the following, we focus on the most relevant key facts concerning these languages. A more detailed description of these languages and the calculation methods and tools using them can be found in [14].

The *Timing Analysis Language* (*TAL*) has been developed by Mok et al. [18]. It is a timing schema approach. The TAL language is an integral part of the timing analysis system developed at the University of Texas and is used by the tool *timetool* [2].

The *Path Language* (*PL*) has been developed by Park and Shaw [24,20,21,19]. It is a path-based approach, which describes feasible and infeasible paths of a program by means of regular expressions. Later on Park developed a more high-level variant of PL called *Information Description Language* (*IDL*) [19], which is easier to use than the more low-level PL.

*Linear Flow Constraints* are typically used by IPET approaches [4,13] as already discussed in the previous section. We thus proceed with *SPARK Ada*. This is a subset of Ada83 which is extended by a special kind of comments which are used for both program proof and timing analysis. Spark Ada programs can be analyzed by the *Spark Proof and Timing System* (*SPATS*), which is based on symbolic execution.

*Symbolic Annotations* is a term which we coined to denote an approach proposed by Blieberger [1]. This approach combines aspects of a pure annotation language with those of a programming language extension. The clue of this approach is the invention of so-called *discrete loops*. These can be considered a generalized and more flexible kind of for-loops. Exploiting the structural properties of discrete loops, however, loop bounds can often automatically be computed by simple mathematical reasoning.

*Bound-T* is a commercial WCET tool originally distributed by Space Systems Finland Ltd. It has been developed by Holsti et al. [10,11,9] and is currently

marketed by Tidorum Ltd. A specialty of the annotation language of Bound-T is that it is designed to be usable both within high-level languages programs and assembler programs.

The *Annotation Language of aiT*, finally, is used by the ait WCET tool, a commercial tool developed by AbsInt Angewandte Informatik GmbH, Germany. This tool targets different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555 [5,8]. A specialty of this tool and its annotation language is to start from binary files as input to be analyzed.

## 5 Main Results

Table 1 summarizes the major findings of our comparison of the seven languages we selected for assessing and highlighting the strengths and limitations of current WCET annotation languages.

Most of the criteria listed in the leftmost column of Table 1 are self-explaining or have been discussed before, except of triangle loops and the various kinds of context-related information.

Intuitively, *triangle loops* are nested loops which meet a triangular pattern in the iteration space $(i, j)$. The two IPET-based methods in our comparison, linear flow constraints and Bound-T, allow a precise description of the behaviour of triangle loops by allowing the use of equalities and inequalities in the specification of constraints.

Often the timing behaviour of the first iteration of a loop is different from that of subsequent iterations, e.g. because of cache effects. *Loop context-sensitive* annotations allow to make such differences explicit. Similarly, the bounds of loops inside of procedures and functions depend often on the values of their input parameters. *Context-sensitive* annotations of the calling context allow to differentiate between the various calling scenarios and thus to obtain more precise analysis results.

*Application context-sensitive* annotations, finally, are a specialty used in SPARK Ada. It refers to a feature called *modes* which allow to describe multiple annotations for a function depending on different input parameters. This resembles the scenario of calling context-sensitivity without being exactly the same. We thus introduced the term *application context sensitivity* for this feature of SPARK Ada.

Table 1 illustrates that none of the seven prototypical annotation languages selected for our comparison uniformly outperforms its competitors. They all 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 [15] or the consideration of application domains of annotation languages which go beyond pure WCET analysis as e.g. recently proposed by Lisper [17].

| 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 | Asm/C | C | - | C, Ada | Asm/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 |

**Table 1.** Assessment summary

## 6 Conclusions

The summary of Table 1 demonstrates that the languages proposed and used so far for WCET analysis all 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" must be made more precise in order to be practically useful. We argue that the underlying notion of the relation "bet-

ter" 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 example on the notion of the relation "better" as discussed above. The challenges above thus represent a full array of more fine-grained challenges rather than exactly three individual challenges.

In a companion paper published in the present proceedings [12], too, we make a proposal towards such a new annotation language by highlighting ingredients, which we consider essential for an annotation language that can serve as a commonly accepted uniform WCET annotation language in the future.

## References


1. Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.
2. Moyer Chen. *A Timing Analysis Language - (TAL)*. Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual.
3. Antoine Colin and Isabelle 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.
4. Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
5. Christian Ferdinand, Reinhold Heckmann, and Henrik 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.
6. Jan 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.
7. Christopher A. Healy, Robert D. Arnold, Frank Mueller, David Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.
8. Reinhold Heckmann and Christian Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.
9. Niklas Holsti. Bound-t assertion language: Planned extensions. Technical report, 2005.
10. Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000.

11. Niklas Holsti, Thomas Långbacka, and Sami Saarinen. *Bound-T timing analysis tool User Manual*. Tidorum Ltd, 2005.
12. Albrecht Kadlec, Raimund Kirner, Peter Puschner, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a common WCET annotation language: Essential ingredients. In *Proceedings of the 25th Annual Workshop of the GI-FG 2.1.4 "Programmiersprachen und Rechenkonzepte", Bad Honnef, Germany*, 2008.
13. Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.
14. Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In *Post-Workshop Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET 2007) Analysis*, pages 83 – 99, 2007.
15. Raimund Kirner and Peter 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.
16. Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
17. Björn Lisper. Ideas for annotation language(s). Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen, 2005.
18. Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtorn 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.
19. Chang Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.
20. Chang Y. Park and Alan C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.
21. Chang Yun Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02.
22. Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.
23. Peter Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.
24. Alan C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
25. Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.
26. Lili Tan and Klaus 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.
27. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckman, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, April 2008.