Softw Syst Model (2011) 10:411–437
DOI 10.1007/s10270-010-0161-0

THEME SECTION

Beyond loop bounds: comparing annotation languages for worst-case execution time analysis

Raimund Kirner · Jens Knoop · Adrian Prantl · Markus Schordan · Albrecht Kadlec

Received: 15 January 2009 / Revised: 23 February 2010 / Accepted: 24 February 2010 / Published online: 9 April 2010

Abstract *Worst-case execution time* (WCET) analysis is concerned with computing a precise-as-possible bound for the maximum time the execution of a program can take. This information is indispensable for developing safety-critical real-time systems, e. g., in the avionics and automotive fields. Starting with the initial works of Chen, Mok, Puschner, Shaw, and others in the mid and late 1980s, WCET analysis turned into a well-established and vibrant field of research and development in academia and industry. The increasing number and diversity of hardware and software platforms and the ongoing rapid technological advancement became drivers for the development of a wide array of distinct methods and tools for WCET analysis. The precision, generality, and efficiency of these methods and tools depend much on the expressiveness and usability of the *annotation languages* that are used to describe feasible and infeasible program paths. In this article we survey the annotation languages which we consider formative for the field. By investigating and comparing their individual strengths and limitations with respect to a set of pivotal criteria, we provide a coherent overview of the state of the art. Identifying open issues, we encourage further research. This way, our approach is orthogonal and complementary to a recent approach of Wilhelm et al. who provide a thorough survey of WCET analysis methods and tools that have been developed and used in academia and industry.

Keywords Worst-case execution time (WCET) analysis · Annotation languages · Path-oriented, constraint-oriented, and hierarchy-oriented WCET annotation languages · WCET annotation language challenge

Communicated by Marko Bošković, Bernhard Schätz, Claus Pahl, and Dragan Gasevic.

This work was partially funded 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, by the ARTIST2 and ARTIST-Design Networks of Excellence (http://www.artist-embedded.org/), and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

R. Kirner (✉) · A. Kadlec
Institute of Computer Engineering,
Vienna University of Technology, Vienna, Austria
e-mail: raimund@vmars.tuwien.ac.at

A. Kadlec
e-mail: albrecht@vmars.tuwien.ac.at

J. Knoop · A. Prantl
Institute of Computer Languages,
Vienna University of Technology, Vienna, Austria
e-mail: knoop@complang.tuwien.ac.at

A. Prantl
e-mail: adrian@complang.tuwien.ac.at

M. Schordan
University of Applied Sciences Technikum Wien,
Vienna, Austria
e-mail: schordan@technikum-wien.at

1 Motivation

Computing the time that the execution of a program can take in the worst case is challenging. It requires to cope with analysis problems that are computationally intractable or undecidable. Tools and methods for *worst-case execution time analysis* (WCET) thus usually content themselves to computing an upper bound of the actual WCET. The precision, generality, and efficiency of these tools and methods depend much on the accurate description and identification of feasible and infeasible program paths. A program path is *feasible*,

if there is an actual program execution taking this path; it is *infeasible* otherwise. In practice, this information is computed by fully automatic program analyses, where possible; it is manually provided by application programmers otherwise. In both cases this requires a dedicated language in order to annotate this information and to pass it on to a subsequent WCET analysis. Such a language is commonly called an *annotation language*. Over the past two decades, an array of conceptually quite diverse annotation languages has been proposed to support the needs of different WCET analysis methods and tools.

In this article, we present a systematic and comprehensive comparison of the various approaches for WCET annotation languages. This acts on our suggestion of the *WCET annotation language challenge* [32] complementing the *WCET tool challenge* [17,22,62], and complements the recent survey of WCET analysis methods and tools by Wilhelm et al. [66].

Focusing on annotation languages for WCET analysis, we take a different angle, as Wilhelm et al. to add to characterize the state of the art in the field of WCET analysis. We believe that this provides a new stimulus for further research and development on WCET analysis methods and tools in general and annotation languages in particular.

It is worth noting that there are also annotation languages for modeling real-time properties at system level like response time, throughput, or jitter. Examples of such system modeling languages are the modeling and analysis suite for real-time applications called MAST [23] and the *UML Profile for Modeling and Analysis of Real-time and Embedded systems* (UML-MARTE) [49]. Such real-time modeling languages at system level, however, are beyond the scope of languages considered in this article, though, for example, UML-MARTE could be extended by user-defined properties to host annotation concepts specifically for WCET analysis.

The contribution of this article is thus twofold: first, to survey the annotation languages for WCET analysis which have been influential and formative for the field. Second, to identify open issues for further research which we consider essential for its further development and advancement.

To this end we introduce a set of criteria that are pivotal for the suitability and usefulness of an annotation language. This leads us to a clustering of the various annotation languages and allows us to investigate and discuss their relative strengths and limitations. Based on these findings we suggest open issues for further research towards novel, easy to use, and expressive annotation languages, which we believe will support the further advancement of methods and tools for WCET analysis. In the long run, this can also be seen a step towards a universal WCET annotation language for a wide range of WCET analysis tools. This would especially enhance the interoperability of different supportive program analysis tools for WCET analysis for their mutual benefit.

WCET annotation languages and WCET calculation methods are closely tied and cannot meaningfully be considered in complete isolation. We thus provide a brief summary of the fundamental WCET calculation methods and the fundamental kinds of flow information which are expressed by means of annotation languages in Sects. 2 and 3, respectively. This will set up the scene for our investigation and comparison of WCET annotation languages.

Figure 1 summarizes the annotation languages we consider, grouped with respect to the WCET calculation method the WCET analysis tools using them are based on. This illustrates the close ties between WCET calculation methods and WCET annotation languages.

2 WCET calculation: fundamental methods

2.1 Global versus scoped WCET calculation

The WCET of a program can be calculated for the whole program at once. We call this *global WCET calculation*. Alternatively, the WCET can be calculated in a hierarchical bottom-up fashion. We call this *scoped WCET calculation*.

Some of the WCET calculation techniques are inherently scoped methods, e. g., timing schema. But also WCET analysis performed intra-procedurally instead of inter-procedurally is a form of scoped WCET calculation.

It is worth noting that scoped WCET calculation methods are only compatible with flow information that addresses properties within the same scope, i. e., with flow information that only addresses program parts within the same scope. The degree to which scoped WCET calculation is used by a concrete WCET analysis tool is thus formative for the WCET annotation language to be used with this tool.

2.2 Timing schema (hierarchy-oriented)

The *timing schema* approach is an efficient WCET calculation method that is also simple to implement [52,54,59]. Essentially, the timing schema consists of hierarchical WCET calculation rules for each node of the syntax tree representing elementary or composed statements. If $T(A)$ denotes the local WCET bound of a node 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 $T(A)+\max(T(B), T(C))$ and the local WCET bound of a loop `while` A `do` B `od;` with at most LB iterations is computed as $(LB + 1) \cdot T(A) + LB \cdot T(B)$ (LB shall remind to loop bound). Last but not least, if A represents an elementary statement, its local WCET $T(A)$ is simply the maximum execution time of A. Obviously, the timing schema can analogously be formulated to calculate the

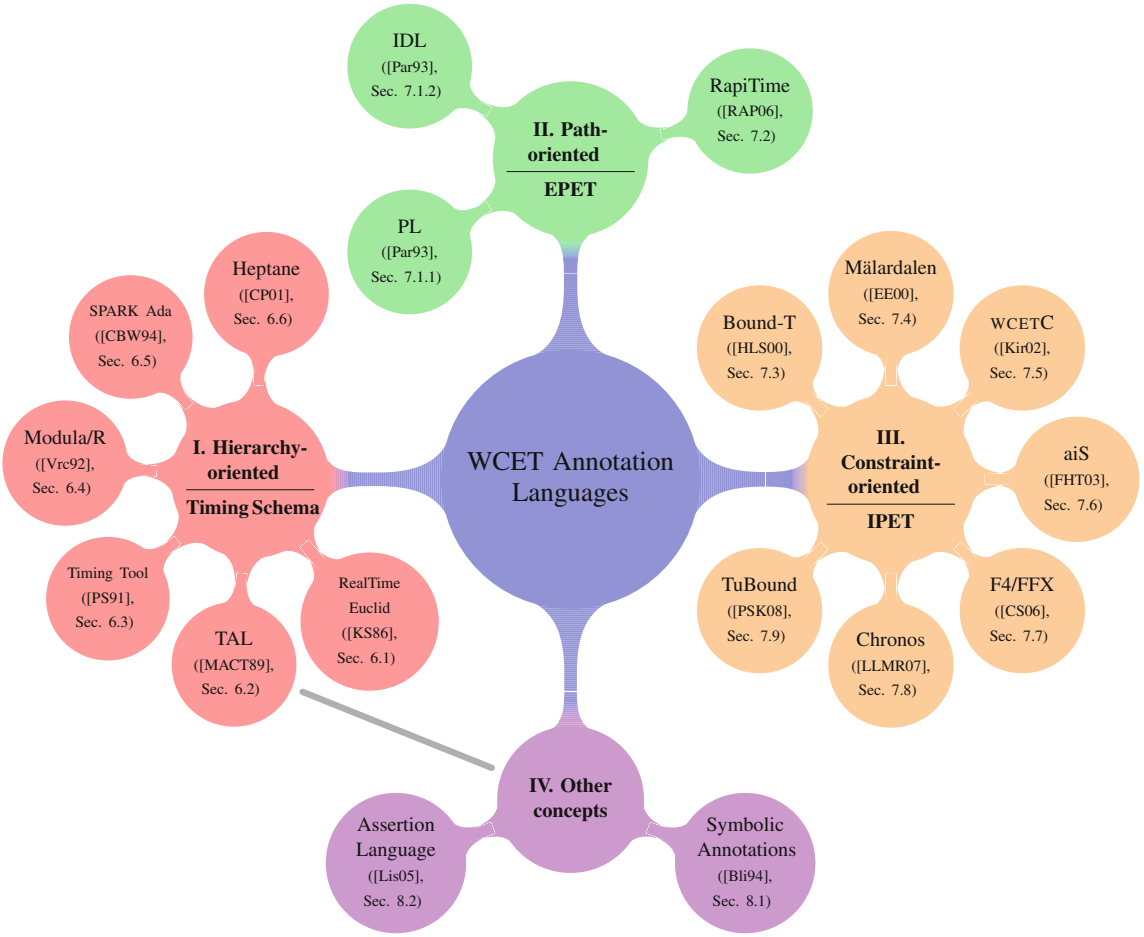


Fig. 1 Formative WCET annotation languages (note: the *gray line* indicates that the WCET calculation tool using TAL is hybrid, enjoying characteristics of *hierarchy-oriented* approaches as well as of unorthodox *other* ones). The various language aspects compared and discussed are listed systematically in Table 1

best-case execution time. Its computational complexity is linear with the program size. It thus scales well and can efficiently be applied to large programs.

The efficiency of this approach comes at the price of lacking support for global flow information, i. e., flow information that describes relations across the boundaries on individual timing schema rules. This is because the timing schema approach represents a fine-grained scoped WCET calculation technique that uses the syntactical elements of control-flow constructs as scopes. For some programs, this can result in a much higher WCET overestimation compared with other WCET calculation techniques. Further, the timing schema approach does not provide an interface to model the complex execution states of modern processors that have features like pipelines and caches. This imposes an inherently high pessimism on the WCET bound for such processors.

A refinement of the timing schema approach towards a more precise handling of nested loops has been presented by Colin and Puaut [10], cf. Sect. 6.6.

As shown in Fig. 1, all approaches that are based on a *hierarchy-oriented* annotation language use the timing schema for the WCET calculation.

2.3 EPET (path-oriented)

The *explicit path-enumeration technique* (EPET) searches the longest execution path by enumerating and comparing each program path. At a first glance, this seems a rather naive approach, and in fact, it does not scale (well) in the case of *global WCET calculation*. However, this approach has its merits when using *scoped* WCET calculation because the scoping allows to restrict the number of paths within each cluster such that their enumeration becomes feasible. For example, some path-oriented WCET calculation techniques use loop scopes as scopes for the analysis [18,58], which, however, implies that any flow information can address only program parts within the body of a single loop; the

body of nested loops or the scope outside the loop cannot be mixed in.

There are static WCET analysis approaches [18,58] and measurement-based timing analysis approaches [67,69] based on EPET.

Static WCET calculation based on EPET is well suited to analyze the effects of pipelines. It allows to model the impact of the pipeline on an instruction sequence longer than just basic blocks, and thus increases the precision of the WCET bound. However, EPET itself is inappropriate to take rather global timing effects into account, like cache behavior. Thus, the cache analysis has to be done in a separate step prior to the EPET step. The separate analysis of cache and pipeline effects, however, can be unsound for processors showing timing anomalies [34,44].

For measurement-based timing analysis using EPET a scoping is used to define the length of the paths for which the timing is obtained by execution-time measurements. Note that measurement-based timing analysis is a hybrid approach that uses execution time measurements as well as program analysis. The scoping used in measurement-based timing analysis can be of quite different granularity. For example, the *pWCET* tool of Bernat et al. [4] uses relatively fine-grained scoping, down to the basic-block level. In contrast, the philosophy of the *MTime* tool is to use a relatively coarse-grained scoping to reduce measurement error [67]. *MTime* is looking for a tradeoff between the maximum code length covered by sub-paths and the number of sub-paths to be measured.

As shown in Fig. 1, all approaches that are based on a *path-oriented* annotation language use *EPET* for the WCET calculation.

2.4 IPET (constraint-oriented)

The *implicit path enumeration technique* (IPET) has been pioneered by Li and Malik [42], as well as by Puschner and Schedl [55]. 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 with constraints. To reduce the complexity, typically only linear constraints are used, i. e., the program is represented as an *integer linear program* (ILP). Subsequent to this basic modeling, supplemental flow information is often included smoothly in terms of 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. As the IPET approach requires to solve a set of constraints describing the program behavior, it is per se a *global* WCET calculation method. It is also possible to use IPET as a scoped WCET calculation technique, as e. g. done by Ermedahl et al. [14].

The IPET-based WCET calculation is currently the most popular WCET calculation technique. It allows to directly use rather flexible flow information like linear flow constraints. Linear flow constraints are supported by so many annotation languages such that we describe them in more detail in Sect. 4. As shown in Fig. 1, all approaches that are based on a *constraint-oriented* annotation language use *IPET* for the WCET calculation.

2.5 Other methods

Hierarchy, path, and constraint-oriented WCET calculation methods are currently most often used by WCET analysis tools. However, there are further approaches using different methods. For example, the WCET analysis method using the TAL annotation language uses the timing schema approach in combination with freely programmable algorithms. It is thus not a pure hierarchy-oriented approach, but a hybrid one. In Sect. 8 we consider methods falling in this group together with their annotation languages in more detail.

We now continue with recalling the fundamental kinds of flow information.

3 Flow information: fundamental kinds

Flow information describes aspects of the dynamic program behavior that are relevant for WCET computation. It is expressed by means of WCET annotation languages. In the following we summarize the fundamental kinds of flow information, distinguishing static and dynamic information.

3.1 Static control flow

The *abstract syntax tree* (AST) is a concise representation of a program which is stripped from information that is unnecessary for its compilation [2]. Primarily, the AST describes the syntactical structure of the program. Implicitly, it also describes its control flow. The *control flow graph* (CFG) is an explicit representation of a program's control flow [19]. Since branching constraints obeyed by actual program executions are not taken into account, it describes the *static control flow*. The nodes of the CFG represent the statements of the program, and the edges denote where execution might continue after executing a statement. Statements can be combined to basic blocks. A *basic block* is a sequence of maximum length of statements which can only be entered at the very first statement and left after the last one. A CFG can be assumed to have a unique start node representing the very first statement and an end node representing the last statement. The static control flow is then given by the set of paths leading from the start node to the end node. Programs with functions or procedures are represented by a *flow graph system*, where each

function and procedure is represented by an ordinary CFG. An *interprocedural control flow graph* or *super control flow graph* represents the control flow by function calls and returns explicitly by splitting each call site into a new call and return node, and introducing a call and a return edge connecting the call node with the start node of the called procedure or function and its end node with the return node of the corresponding call site [47]. This way the CFGs of the flow graph system are merged to a single graph. The set of paths in the interprocedural control flow graph from the start node to the end node of the program that respect the call/return-behavior of procedure and function calls (also known as *interprocedurally valid paths* [60]) denote the static control flow of a program with procedures and functions. Like for the AST, all branches are non-deterministically interpreted in a CFG and its interprocedural extensions in order to avoid undecidability issues. Hence, the static control flow of a program describes a superset of the program paths, for which there is an actual program execution, i. e., it describes a superset of the set of feasible program paths [5].

Another representation which is often used to describe the control flow induced by procedure and function calls is the call graph of a program [47]. A *call graph* contains a node for each procedure and function of a program. A directed edge leads from node m to node n in the call graph, if the procedure or function represented by m contains a call for the procedure or function represented by n. The call graph together with the flow graph system of a program can also be used to describe its static control flow.

3.2 Dynamic control flow

The dynamic control flow constrains the static control flow towards a more precise approximation of the set of feasible program paths. Typically, this is achieved by means of a control flow analysis, which conceptually can be thought of identifying a subset of the set of infeasible program paths. The precision of the approximation depends thus on the ability of the control flow analysis to identify infeasible program paths. To this end the control flow analysis can be guided by flow information to enhance its capabilities, where the flow information constrains and restricts the static control flow. This flow information is what is usually provided in terms of annotations of the program. Most important is here the provision of upper bounds for cycles in the static control flow representation of a program, which are due to loops and (mutually) recursive procedure and function calls. Generally, we distinguish the following kinds of dynamic control flow information:

Loop bounds Loop bounds are the minimum information required by WCET analyzers to come up with a WCET bound for a program. For *natural loops*, i. e., loops with a distinct loop header dominating all nodes in the loop body and a back edge pointing to the loop header [2], loop bounds specify the maximum iteration count of the loop body relative to the loop header. For non-natural loops, e. g., for irreducible loops, the specification of loop-bounds is less intuitive since there is no unique loop entry.

Recursion bounds Recursion bounds are similar in flavor to loop bounds. They provide an upper bound for the number of times a recursive function or procedure calls itself upon each invocation. Specifying bounds for mutually recursive procedure calls is less obvious and not directly possible.

Nested loops with non-rectangular iteration space If the range of an iteration variable ($i, j, k, ...$) of an inner loop depends on the current value of the iteration variable of an outer loop, the shape of the iteration space becomes non-rectangular [47]. Common examples are trapezoidal and triangular iteration spaces, the latter are also called *triangle loops*.

Call contexts Call-contexts provide information about the context, in which a procedure is called at a particular call site, e. g., the values of procedure arguments and global variables. Call contexts allow to analyze a procedure call site specifically. In general, this allows more precise analysis results. For example, the computation of loop bounds that depend on the values of input parameters or global variables at a call site, will usually be more precise if the analysis can refer to call-context information at call sites [38].

Loop contexts Loop-contexts are similar to call contexts but focus on loops instead of call sites. Often, the first and subsequent loop iterations behave differently, e. g., because of cache effects. The precision of analyzing such loops can be improved when, e. g., an annotation language allows for annotating the first and subsequent iterations of a loop separately.

Application contexts Application contexts provide information about the context an application is executed in, e. g., the possible states of the environment. An application context describes the environment of the program, while a call context describes the local context of a particular call site. Of course, a change in the application context can also cause a change of a particular call context. A program often behaves significantly different due to changes in the environment. The precision of analyses can thus be improved if annotation languages allow for the specification of specific application contexts.

Execution order The WCET of a program can depend on the concrete order in which its statements are executed, e. g., because of effects of instruction pipelines, instruction/data caches, or processor parallelism. Most WCET-calculation methods do not take this into account and

content themselves with estimates of the execution frequency of basic blocks. If, however, the method supports the modeling of a complex hardware architecture where the instruction timing depends on the execution history, the annotation language should support the specification of the execution order.

We conclude these introductory sections with discussing an extended example of an IPET-based WCET analysis. The IPET is the state-of-the-art WCET analysis technique in most application domains (cf. Sect. 2.4)[1] The example illustrates the interplay of a WCET calculation method and a supporting annotation language, and it shows how flow information is used to tighten the result of the WCET analysis.

4 Example: IPET and linear flow constraints

The IPET method supports the use of arbitrary constraints of the form $\sum_{j=1}^{n} f_{ij}(x_j) \leq b_i$ between the flow of different program locations (modeled by flow variables x_j) as flow information (index i). Though $f_{ij}()$ in general could also be non-linear, we assume the use of an *integer linear programming* (ILP) constraint solver for WCET calculation. This limits the flow information to linear flow constraints. We use the source code and the flow information of benchmark B_1 and B_2 as example. They are shown in Table 2 together with the corresponding CFG.

The universe of paths is defined by a general ILP problem. It consists of n decision variables $x_1, \ldots, x_n$, 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 calculate the WCET as an ILP problem, the execution costs of all program actions are to be expressed as the objective function to be maximized:

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

In this equation, E denotes the set of edges in the CFG, the constant $t_{i,j}$ the execution time of edge $\langle i, j \rangle \in E$ from node i to node j, and the variable $f_{i,j}$ the execution count of this edge.

To make the maximization of the objective function a valid and precise WCET bound, the values of the ILP variables $f_{i,j}$ must be restricted by additional constraints:

1. Subjecting all ILP variables $f_{i,j}$ to integer solutions, since the execution count of any CFG edge can only be an integer value. For this we use the syntax `int` $f_{i,j}$.
2. Adding flow constraints that reflect the CFG structure: The incoming flow of any CFG node is equal to its outgoing flow. For CFG node 5 of our example, this equation is $f_{4,5} = f_{5,6} + f_{5,8}$.
3. Bounding the value of the flow variable that represents the program entry by one. This ensures the calculation of the WCET for a single program execution: $f_{1,3}$ `= 1`.
4. Adding constraints that reflect the flow information of the benchmarks B_1 and B_2. For example, the upper loop bound of the only loop in the example is 100. This can be modeled by the additional constraint: $f_{12,4} \leq 100 f_{3,4}$ where edge $\langle 3, 4 \rangle$ is the entry and $\langle 12, 4 \rangle$ the back edge of the `while` loop.

The resulting ILP problem for WCET calculation is shown in Fig. 2. Note that for simplicity we assume that the execution time of each action in the program is one time unit, i. e., $\forall \langle i, j \rangle \in E.\ t_{i,j} = 1$.

There exist many off-the-shelf ILP solvers to solve the ILP problem of Fig. 2, some of which are available for free, e. g. *lp_solve* [43]. Figure 3 shows the solution of the ILP problem. The first line shows the value of the objective function: The WCET bound is 653 time units. The remaining lines show the execution count of all flow variables for the particular solution. It is worth noting that in general it is not possible to reconstruct a unique control-flow path from the given execution-count values of the flow variables. For example, the given solution can be instantiated to multiple control-flow paths, depending on the order of the control-flow edges $\langle 5, 6 \rangle$, $\langle 5, 8 \rangle$, $\langle 10, 12 \rangle$, and $\langle 10, 11 \rangle$.

This example clearly demonstrates that flow information is indispensable for providing a limit on the number of iterations of loops, and that the side constraints given in benchmarks B_1 and B_2 (the last two constraints in Fig. 2) help to tighten the WCET bound. If these two constraints were omitted, the calculated WCET bound would rise from 653 time units to 703 time units.

5 Comparison criteria

We compare WCET annotation languages with respect to a set of *language design* characteristics, with respect to *intuitiveness* and the *availability* of a WCET analysis tool using

[1] Hierarchy-based WCET calculation is useful in domains where WCET computation underlies very tight time constraints.

Table 1 Comparison and assessment of annotation languages

Category	Language	Expressiveness	Loop-bounds	Non-rectangular loops	Call-contexts	Loop-contexts	Application-contexts	Execution Order	Annotation placement	Abstraction level	Programming language	Intuitiveness	WCET Tool	Annot. Producer	Consumer
Hierarchical															
	Real-Time Euclid	Loop-bounds	✓	·	·	·	·	·	source constructs	source	Euclid	high	no	·	✓
	TAL	Timing schema	✓	·	(✓)	·	·	·	external script	object	C/Asm.	low	academic prototype	✓	✓
	Timing Tool Language	Timing schema	✓	·	·	·	·	·	interactive	source	C	high	academic prototype	·	✓
	Modula/R	Linear flow constraints	✓	(✓)	·	·	·	·	source constructs	source	Modula	medium	academic prototype	·	✓
	SPARK Ada	Loop-bounds	✓	·	·	·	✓	·	source code comments	source	Ada	high	academic prototype	·	✓
	Heptane Language	Loop-bounds	✓	✓	·	✓	·	·	source constructs, external file	source / object	C	medium	academic (open src)	·	✓
Paths															
	PL and IDL	Regular expressions	✓	·	·	·	·	✓	–	source	C	medium	no	·	·
Constraints															
	Bound-T Language	Linear flow constraints	✓	·	(✓)	·	·	·	external file	object	C, Ada	medium	commercial	✓	✓
	Mälardalen Language	Linear flow constraints	✓	(✓)	·	✓	·	·	external file	source	C	medium	academic prototype	✓	✓
	WCETC	Linear flow constraints	✓	(✓)	·	·	·	·	source constructs	source	C	medium	academic prototype	·	✓

Table 1 continued

Category	Language	Expressiveness	Loop-bounds	Non-rectangular loops	Call-contexts	Loop-contexts	Application-contexts	Execution Order	Annotation placement	Abstraction level	Programming language	Intuitiveness	WCET Tool	Annot. Producer	Consumer
	aiS (aiT)	Linear flow constraints	✓	(✓)	(✓)	(✓)	✓	·	external file + some source code	source / object	C/Asm.	medium	commercial	(✓)	✓
	F4/FFX (Otawa)	Linear flow constraints	✓	(✓)	✓	·	·	·	external file	source / object	C	medium	academic (open src)	✓	✓
	Chronos Language	Linear flow constraints	(✓)	(✓)	·	·	·	·	GUI / external	source	C	medium	academic (open src)	·	✓
	TuBound Language	Linear flow constraints	✓	(✓)	·	·	·	·	#pragma statements	source	C++ subset	medium	academic prototype	✓	✓
Others															
	Symbolic Annot. in Ada	Loop annotations	✓	(✓)	·	·	·	·	source constructs	source	Ada	medium to high	academic prototype	·	✓
	Assertion Language	Arbitrary algorithms	(✓)	(✓)	·	(✓)	(✓)	(✓)	source code, external file	source	any	low	no	·	·

✓…specific annotation construct available (✓)…supported indirectly by generic annotation constructs

```
max: 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;

  f1_3 = 1;   /* Input flow constraint */

  f1_3 = f3_4; /* Structural constraints */
  f3_4 + f12_4 = f4_5 + f4_13;
  f4_5 = f5_6 + f5_8;
  f5_6 = f6_9;
  f5_8 = f8_9;
  f6_9 + f8_9 = f9_10;
  f9_1 = f10_12 + f10_11;
  f10_11 = f11_12;
  f10_12 + f11_12 = f12_4;

  f12_4 ≤ 100 f3_4; /* Loop bound */

  f5_6 = f10_11; /* Constraint B1 */
  f5_6 ≤ f5_8; /* Constraint B2 */

int f1_3,f3_4,f4_5,f5_6,f5_8,f6_9,f8_9,f9_10,
   f10_11,f11_12,f10_12,f12_4,f4_13;
```

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

```
Actual values of the variables:
f1_3                    1
f3_4                    1
f4_5                  100
f5_6                   50
f5_8                   50
f6_9                   50
f8_9                   50
f9_10                 100
f10_11                 50
f11_12                 50
f10_12                 50
f12_4                 100
f4_13                   1
```

Fig. 3 Solution of the ILP problem given in Fig. 2

them. The language design characteristics are under control of the language designer. Intuitiveness is mostly an outcome of the language design characteristics. Tool availability is an indicator for the general suitability and usefulness of an annotation language. Though it does not refer to a specific property or feature of an annotation language, we consider tool availability a valuable information on its own. Our focus, however, is on the annotation languages. We thus do not compare or assess the quality of the WCET analysis tools. Readers interested in this are referred to [66].

5.1 Language design

Expressiveness We use expressiveness to denote the capability of an annotation language to describe (sets of) control-flow paths as precisely as possible. Essentially, this is an outcome of the types of flow information that can be annotated. An annotation language, which is able to accurately describe all feasible control-flow paths of arbitrary terminating programs, is called *path-complete*. Annotation languages lacking this property enforce the usage of over-approximations of the set of feasible program paths for WCET analysis, which generally leads to an overestimation of the WCET of a program. In practice, it is most important if an annotation language is capable of coping with interprocedural control flow or with selected iteration ranges of loops.

The most important setscrews for tuning the expressiveness of an annotation language are the means for dealing with *loop bounds*, specific loop-types such as *triangle loops*, *context sensitivity*, and the *execution order* of statements.

Annotation placement and abstraction level These involve pivotal design decisions: (1) *Annotation placement*: Where to keep annotations? As part of the code, or separately in another file? (2) *Abstraction level*: Which code to annotate? High-level source code or low-level machine code?

Thinking in terms of the user's effort of using an annotation language it is obvious that these design options have a strong impact on its usability.

Regarding the first design decision, none of the two design options is consistently superior to the other. As a rule of thumb: if annotations must be manually provided, it is usually considered less complex and error-prone to annotate the source code. If annotations are computed automatically, it is often advantageous to keep the annotations separate from the program code as it simplifies multiple uses of them within a tool chain.

For the second design decision, an other rule of thumb might be helpful: for a user high-level source code annotations are usually easier to understand and cope with than low-level machine code annotations. This holds both for user-provided (where it seems obvious) and automatically computed annotations. Often it is necessary to manually verify annotations, e. g., that the correct execution context has been taken into account. If machine code has been annotated, this makes it necessary to maintain a mapping between source code and machine code, which is typically non-trivial. This also holds if the behavior of high-level source code language constructs shall be described in terms of object code annotations. Often such a mapping is realized by defining a set of language constructs, the so-called *anchors*, that will still be recognized after compilation, such as loops or procedure calls.

If an annotation language supports object code path analysis, this imposes additional challenges compared with source code path analysis. One of the reasons is the construction of the CFG. Whereas in high-level source code the control flow is usually defined in terms of structured control-flow statements, which allows for a simple computation of the CFG, reconstructing it from object code requires usually additional annotations.

Programming language The complexity of WCET analysis depends much on the features of the programming language, in which application programs are written. Restricting the programming language to a subset used for real-time application programming is thus an important option which also allows for less complex annotation languages. One such example is to exclude language constructs, for example, the `goto` statement in ANSI C, which can be the source of irreducible code [47]. Irreducible code is problematic for hierarchical WCET analysis like the timing schema approach. In particular, it is non-intuitive to annotate loop bounds for such code. Alternatively, programming languages can implicitly be constrained for real-time applications. Typical reasons for this are compatibility constraints between the WCET calculation method and a specific annotation language, or the limited language coverage of program analyses for the computation of flow information. Floating point operations, for example, might not be supported by an annotation language or a program analysis.

5.2 Intuitiveness

For a qualitative assessment of intuitiveness, we consider the skills, the learning curve, the amount of work, and the complexity imposed on a user as important characteristics. A higher amount of implicitly assumed knowledge going beyond the annotation language itself, e. g., about the subsequent WCET analysis, or of specifics of its implementation diminishes intuitiveness. The amount and complexity of work that is required to update program annotations in response to a program update is another important characteristic. In the following, we use the term *intuitiveness* to refer to this set of characteristics; awarding annotation languages one of the three grades *high*, *medium*, or *low*.

Concerning tool support, another issue that is related to intuitiveness but not part of our comparison is how the results of the WCET analysis are presented. A reporting tool of the WCET analyzer could provide the user with information in terms of annotations which explain the computed WCET. Flow constraints as used by WCET analyzers based on ILP solvers (cf. Sect. 2.4), e.g., could provide information about the execution frequency of statements, but not about the execution order.

5.3 Tool availability

As mentioned earlier, the availability of a WCET analysis tool using an annotation language is an indicator for the suitability and usefulness of the annotation language in general. We thus report the availability of a WCET analysis tool in our comparison of WCET annotation languages, together with the information if it is of academic or industrial origin. More detailed, we additionally report if these languages are purely consumers of annotations, or also producers, the latter possibly be realized by means of a separate tool as part of a tool chain.

Retrospectively, it can be observed that the design of WCET annotation languages went typically hand in hand with the development of WCET analysis tools using them. In the following, we use this observation in order to present and discuss the various annotation languages in an evident order. We group the annotation languages according to the particular fundamental WCET calculation method applied by the WCET analysis tool(s) using them. The annotation languages of each group are then discussed in the order of their advent. Fig. 1 presents a comprehensive overview.

6 Hierarchy-oriented annotation languages

6.1 Real-Time Euclid

Real-Time Euclid is one of the first real-time programming languages that features annotations for timing analysis [37]. It is structurally restricted in order to ensure the analyzability of programs. Recursive functions and dynamic data structures are not permitted. Instantiation and activation of tasks can directly be specified, based on a period or on the occurrence of an event.

The specification of loop bounds is the only kind of flow information which is supported by Real-Time Euclid. Loop bounds can be specified in terms of the maximum number of loop iterations or of the maximum amount of real-time units the execution of a loop will take. In the latter case, the compiler deduces an upper bound for the number of loop iterations from the given amount of real-time units. This requires, besides knowing the execution time of the loop body, information about the absolute time per real-time unit. The compiler can be provided with this information using the `realTimeUnit` construct: `realTimeUnit :=` *timeInSeconds*.

6.2 TAL: equations with event markers

The *Timing Analysis Language* (TAL) has been developed by Mok et al. [45]. A detailed description of TAL can be found in [9]. TAL is an integral part of the *Timing Analysis*

```
1 main() {                  // -v-L1:
2   int i=0, j=0;
3   while (i < 100) {       // -v-L3:
4     if (i < 10) j++;      // IF_1:
5     i++;                  // -^-L5:
6   }                       // -^-L6:
7 }                         // -^-L7:
```

Fig. 4 C language program

```
1 func TAL_main() {
2   block blk1;
3   loop lp1;
4
5   blk1#begin = "-v-L1";
6   blk1#end = "-^-L7";
7
8   lp1#begin = "-v-L3";
9   lp1#count = MAXINT;
10  lp1#end = "-^-L6";
11
12  return(blk1#time);
13 }
```

Fig. 5 Automatically generated TAL-Script skeleton

System (TAS). This is a representative of the *timing schema* approach. TAS not only consists of multiple tools, but also requires user-assistance. The general work flow is a four-stage process:

1. The *annotate* tool analyzes the C language argument program and produces C code that is annotated with default assumptions about the dynamic program behavior.
2. A modified C compiler translates the annotated C program into annotated assembler code. Using the source code annotations, the compiler produces a TAL-script skeleton.
3. User assistance is required to manually refine the TAL-script skeleton of the previous step. A graphical user interface is provided to aid the user.
4. The *timetool* tool, finally, uses the refined TAL-script to perform the actual WCET calculation on the assembler code of the argument program.

Figure 4 shows a simple ANSI C program given in [45]. Figure 5 shows the corresponding automatically generated TAL-script skeleton. The script contains references to labels that occur in the assembler code generated by the compiler. In Fig. 4 these references are made explicit by inserting their locations into the C-source.

Figure 5 shows various examples of timing-analysis relevant program constructs which can be addressed by TAL: *loop* describes a loop construct where the execution frequency depends on the data being processed; *block* denotes a program fragment which may contain loops; however, the execution time of the block must be fixed. TAL defines also the notion of an *action*, that is, any larger program fragment whose execution time is of interest. TAL distinguishes primitive and composite actions. Each object is associated with a set of attributes, such as the `time` and (loop-)`count` expressions. The syntax for assigning attributes is `object#attribute = expression`.

Considering the automatically generated TAL-script skeleton of Fig. 5, a programmer is likely to make two changes to the script in the third stage: first, replacing `MAXINT` as loop count attribute of `lp1` by a more accurate value:

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

Second, parameterizing the function definition and changing the calculation formula in line 12 of the script to express the fact that the inner `if`-statement is executed only ten times:

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

and

```
12  blk2#begin = "IF_1";
13  blk2#end = "^-L5";
14
15  return (blk1#time - (lp1#count -
        if_count)*blk#2time);
```

A particular strength of TAL is its expressiveness. TAL-scripts may contain arbitrary calculations. In principle, this allows users to specify formulæ to compute nearly perfect execution time bounds. However, this is extremely challenging, since users have to devise the complex formulæ on their own.

6.3 Interactive annotations with the timing tool

The *Timing Tool* (TT) has been developed by Park and Shaw [54,59]. It is one of the first implementations of a timing schema approach. It supports a subset of the C language and addresses MC68010-based SUN workstations as target platform. TT allows the user to interactively specify "software timing property"-annotations at the source code level. In a typical TT session, the user is shown the source code and requested by the tool to specify both upper and lower bounds for loops. An example is shown in Fig. 6. The numbers in square brackets denote the estimate of minimum and maximum clock cycles the execution of last statement would take on the target hardware.

```
fvalue = -1;/* all data are positive */
                                [ 20, 20 ]
while (low ≤ up)
   *** WHILE statement ***
         Input LOOP-BOUNDS [ low up ]: 1 4
{
  mid = (low + up) >> 1;    [ 52, 52 ]
```

Fig. 6 Interactive session with the Timing Tool (from [54])

6.4 Modula/R

The Modula/R language has been developed by Vrchoticky [63,64]. It comes together with a compiler system that provides support for timing analysis. Modula/R is derived from Modula-2 with extensions tailored to timing and memory consumption analysis. It supports the following annotations for WCET analysis:

- `impure`: a flag that has to be given for the definition/call of each procedure that has side-effects. This is also useful for improving other static program analyses.
- `max` *ConstExpr* `times`: for each loop a fixed loop bound has to be specified.
- `loopsequence` Ident `max` *ConstExpr* `times` Stmts `endsequence` Ident: The *loopsequence* constant is the bound (*ConstExpr*) of the overall iteration count of multiple loops. Each loop that belongs to a loopsequence with symbolic identifier *Ident* has to be prefixed with "`in` *Ident*".
- `scope` *Ident LoopStmt* `endscope` *Ident*: a scope can enclose a loop statement to express information about the possible control flow inside the loop. The information about the control flow is expressed by markers of the form "`in` *Ident* `max` *ConstExpr* `times`" where *Ident* is the symbolic name of the corresponding scope. Markers are similar to loop bounds, but besides loops they can also be attached to the `then` and `else` branches of `if`-statements.

The Modula/R compiler allows to optionally check the validity of the annotations at runtime. Modula/R, however, does not support arbitrary linear flow constraints. It has been developed before IPET-based WCET tools became available.

6.5 SPARK Ada: data value assertions

SPARK Ada is the programming language used in the *Spark Proof and Timing System* (SPATS).[2] Chapman et al. [7,8] adopted SPARK Ada for WCET analysis. The SPARK language is a subset of Ada83 that is extended by a special kind of comments serving as annotations. The annotations are used for both program verification and timing analysis. Like the program verification framework, the WCET calculation in SPARK Ada is based on *symbolic execution*.

The edges of the CFG of the input program are attached 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 makes the approach independent of the target hardware.

The static semantics of SPARK Ada requires at least one assertion to be placed at *cut points*. Cut points are at the entry of every loop statement and before and after each function body. Thus, the CFG can be decomposed into a set of cut points and the so-called *basic paths* connecting them.

The problem of WCET computation is then equivalent to finding the longest path of the weighted CFG. This can be solved by a simplified version of Tarjan's algorithm [61]: applying a set of transformation rules, an acyclic directed graph is mapped to a regular expression that is used to find the shortest path. For SPARK Ada, the dual problem of finding the longest path is considered. To handle loops, a special bounded iteration operator is included in the regular expression syntax. Chapman et al. [7] give three graph rewriting rules to collapse alternatives, inner loops, and outer loops to a simplified graph containing fewer edges, but more complex regular expressions as weights. SPARK Ada expects the programmer to manually supply loop bounds.

A distinct feature of SPARK Ada is the provision of *modes*. This 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.

For illustration, consider the example of Fig. 7, which is taken from [8]. It shows a program that implements the power function.

6.6 The annotation language of heptane

The Heptane WCET analysis tool accepts two different input formats, which are used by different WCET calculation methods [20]: C source code, which is used by a tree-based WCET analysis, and machine code, which is used by an IPET-based WCET analysis. The concepts underlying Heptane are described in [10]. A description of the annotation language of Heptane can be found on the tool web page [20].

Annotations of the C source code are placed as additional constructs inside the source code. Heptane supports two kinds of source code annotations: loop bounds and absolute

[2] SPARK is an acronym for *SPADE Ada Kernel*; SPADE is an acronym for *Southampton Program Analysis Development Environment*.

```
1 --# proof function pow(FLOAT,INTEGER) return FLOAT;
2 function POWER(BASE: in FLOAT;
3                EXPONENT: in INTEGER) return FLOAT
4 --# pre true;
5 --# mode A (EXPONENT >=0);
6 --# mode B (EXPONENT < 0);
7 --# post (POWER = pow(BASE,EXPONENT));
8 is
9   ONE: constant FLOAT := 1.0;
10  EXCHANGE: BOOLEAN;
11  L_RES: FLOAT;
12  L_EXP: INTEGER;
13  RESULT: FLOAT;
14 begin
15  L_RES := ONE;
16  if EXPONENT ≥ 0 then
17    EXCHANGE := FALSE;
18    L_EXP := EXPONENT;
19  else
20    L_EXP := -EXPONENT;
21    EXCHANGE := TRUE;
22  end if;
23  --# loopcount(L_EXP);
24  loop
25    --# assert
26    --# ((not EXCHANGE) -> (L_RES = pow(BASE,(EXPONENT
        - L_EXP)))) in A;
27    --# & (EXCHANGE -> (L_RES = pow(BASE,(-EXPONENT -
        L_EXP)))) in B;
28    exit when L_EXP = 0;
29    L_RES := L_RES*BASE;
30    L_EXP := L_EXP-1;
31  end loop;
32  if EXCHANGE = TRUE then RESULT := ONE / L_RES;
33  else RESULT := L_RES; end if;
34  return RESULT;
35 end POWER;
```

Fig. 7 Annotated SPARK Ada program (from [8])

time bounds for procedures, if the source code is not available.

The following C source code annotations are available to bound loops:

- `[ Expr ]`: upper loop bound information. *Expr* may use any existing function of Maxima or Maple [46,68].
- `[ Expr`$_1$`, Expr`$_2$ `]`: upper loop bound information. *Expr*$_1$ is the upper loop bound and *Expr*$_2$ is a symbolic expression that describes the value of the loop induction variable (using a special variable *i* as a representative of the current iteration count starting with zero).
- `NLast(P, Expr )`: is a function that can be used within a loop bound expression. It yields the current values of loop induction variables in outer loops. The value 1 for *Expr* addresses the immediate outer loop. P denotes the immediate outer loop (i. e., "parent loop").

The following C source code annotations are available to analyze function calls where the source code of the callee may not be available:

- `[HEPTANE_EXTERNAL_ASM]`: specifies within a function declaration that the assembly code for the function is available.
- `[HEPTANE_INLINE_TIMING,` *IntValue*`]`: placed inside a function, this specifies an absolute time of *IntValue* clock cycles to be used as WCET for this function.

An example of how to use the loop annotations of Heptane is given in Fig. 8. The example shows the implementation of a Fast Fourier Transformation, after slicing away all computational straight-line code.

Heptane machine code annotations are given in a separate XML file. In principle, the IPET-based WCET analysis approach could support arbitrary linear flow constraints.

Fig. 8 ANSI C program with loop annotations for Heptane

```
1  #define NSAMPLES 1024
2
3  void fft()
4  {
5    unsigned a, n, bs, be=1;
6
7    for (bs=2; bs ≤ NSAMPLES; bs=bs<<1)[10, pow(2,i
       +1)]
8    {
9      for (a=0; a<NSAMPLES; a+=bs) [NSAMPLES/nlast(P,1)]
10     {
11       for (n=a; n < be; n++) [nlast(P,2)/2]
12       {
13         /* code ... */
14       }
15     }
16     be = bs;
17   }
18 }
```

`path` ::
a regular expression of `symbols`
`symbols` ::
alphabets(Σ) : a set of code labels
operators : $+, \cdot, \star, \cap, \neg$ ($\star$ ... Kleene star)
parenthesis : (,)
empty set : $\emptyset$
wild cards :
$*$ ... arbitrary strings of code labels, i. e., $* = \Sigma\star$
... all strings 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 wild-card '$*$'!

Fig. 9 PL-based on regular expressions [51]

However, the machine code annotations are restricted to resolve dynamic branch targets and loop bounds.

7 Path-oriented annotation languages

7.1 PL and IDL

The *Path Language* (PL) and the *Information Description* Language (IDL) have also been developed by Park and Shaw [50,51].

7.1.1 Path language

PL has specifically been designed to provide advanced support for the specification of (in)feasible program paths of programs of high-level languages like ANSI C. PL is based on regular expressions as shown in Fig. 9. Recursive procedure calls are thus not allowed (this would require pushdown automata instead of regular expressions). The basic idea of PL is to annotate instructions which are interesting to path characterization with labels. PL can describe path patterns representing a set of paths.

Multiple occurrences of a pattern can be abbreviated, e. g., A^{2-4} is an short-hand version of $AA+AAA+AAAA$. Using this convention, it can be easily expressed, for example, that a loop whose body is assumed to be labeled LB, has an iteration count of at most 10: $\_(LB\_)^{0-10}$. This convention could be extended to also describe feasible paths of bounded tail-recursive function calls; however, such an extension is not considered in [50,51].

A particular strength of PL is its outstanding expressiveness. It allows to describe patterns of explicit execution order of labeled statements. PL is in fact complete, i. e., it allows to describe all paths of arbitrary terminating programs. To specify the PL expression describing the set of feasible paths of a given program one has to instantiate the possible shape of input data, i. e., one has to take the possible valuations of input data into account.

A significant drawback of PL is that even common path patterns can result in very long expressions. For example, linear flow constraints like $f_i < f_j$ expressing that 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 Path analysis based on regular expressions, e. g., for figuring out the path with the maximum execution time, can be computationally expensive.

$\Sigma\star$ represents the set of all possible paths to be constructed with the code labels Σ. Park and Shaw use A_P to denote the set of all paths that are syntactically possible due to the structure of the CFG ($A_P \subseteq \Sigma\star$). Every path information I_i represents a set of feasible or infeasible paths $I_P^i \subseteq \Sigma\star$. Since each path information I_i is an additional constraint that does not include the syntactical structure of the CFG, it typically holds that $I_P^i \setminus A_P \neq \emptyset$. The path analysis approximates the set of feasible paths (denoted by X_P by Park and Shaw) by intersecting A_P with the set of paths I_P described by the conjunction of all path information I_i: $I_P = \bigcap_i I_P^i$. The set of feasible paths X_P is then approximated by $X_P \supseteq X'_P$, which is calculated as follows:

$$X'_P = A_P \cap I_P$$

Unfortunately, the computational costs of the central path processing operations $\neg$ and $\cap$ are exponential in the worst case [48]. In its generic form PL allows thus a user to specify expressions which are too complex for path processing. Therefore, Park and Shaw complemented PL with a more restrictive higher level *information description language*, which we recall next.

7.1.2 Information description language

Information description language (IDL) is designed to address and overcome the shortcomings of PL. The meaning of high-level IDL expressions is given by a structured subset of low-level PL expressions [51]. Park and Shaw provide a transformation that transforms high-level IDL expressions into low-level PL expressions. The advantage of IDL is that the resulting PL expressions are only a subset of all possible low-level PL expressions, resulting in a more efficient path processing.

For example, the flow information that labels A and B can only be executed together is expressed by the IDL-expression `samepath(A,B)`. Its meaning is defined by the corresponding low-level PL-expression $(*A*) \cap (*B*) + \neg(*A*) \cap \neg(*B*)$. As a second example, a loop of scope A with constant iteration count K is specified by the IDL-expression `loop AK times`. Its meaning is given by the low-level PL-expression $\neg(*A*) + (\_A.entry\_A.body(\_A.body)^K) \star \_$.

The second example illustrates how difficult it is to get descriptions of path sets using low-level regular expressions right. The original low-level representation given in [51] is indeed faulty. It does not take care of the case that a loop may be nested within another loop. The original low-level representation was like $\neg(*A*) + \_A.entry\_A.body(\_A.body)^K\_$, which in case of nested loops erroneously excludes paths with multiple executions of the loop.

The strength of IDL (as well as of PL) is its ability to express path patterns of explicit execution order.

Nonetheless, IDL still suffers from a weakness it inherits from PL: information about relative execution frequencies of code can only be expressed by explicitly enumerating all possible path patterns. This can be of exponential length. The example of Column 4 of Part 2 of Table 2, Benchmark B2, illustrates this phenomenon.

7.2 The annotation language of RapiTime

RapiTime is a so-called measurement-based timing analysis tool, i. e., it combines execution-time measurements and program analysis to obtain the WCET estimate [57].

The annotation language of RapiTime focuses on placement of annotation points describing where to instrument the program for execution time measurements.

In general, measurement-based timing analysis may also use flow information to specify the set of feasible paths. However, RapiTime currently does not use such annotations and instead assumes that the user-given data trigger all worst-case control flow, for example, the maximum iteration count of loops.

Because the annotations supported by RapiTime are not directly bound to WCET analysis, RapiTime is not included in the comparison of annotation languages given in Table 1.

7.3 The Bound-T annotation language

Bound-T is a commercial WCET analysis tool. Originally developed by Space Systems Finland Ltd, it is now marketed and developed by Tidorum Ltd [24]. Bound-T operates on the object-code and proceeds in four stages, where it relies on debug information and in part on user-assistance for the provision of additional assertions.

1. Call graph construction
2. Automatic loop bound computation
3. User-assistance for specifying missing loop bounds
4. WCET calculation

The first stage performs a control-flow analysis of the argument program and constructs its call graph.

The second stage applies a data-flow analysis to automatically compute loop bounds, where possible. In this stage, each loop body is analyzed by rewriting individual statements into Presburger arithmetic, a decidable subset of integer arithmetic. By expressing each loop body as composition of decidable formulæ, it is possible to compute the increment values of loop counters and based thereon bounds for all counter-based loops.

The third stage involves user-assistance for loops which could not automatically be bounded in the second stage. Bound-T emits a warning for each missing bound, together with the context of the loop in question. For such loops, the user is prompted to provide an *assertion* which specifies the missing bound.

The fourth stage, finally, computes the WCET bound for the program under consideration. This WCET computation proceeds bottom-up on the call graph, which has to be acyclic. The WCET calculation is then performed by transforming the flow information and the program structure into an ILP problem which is subsequently passed to the *lp_solve* tool [43].

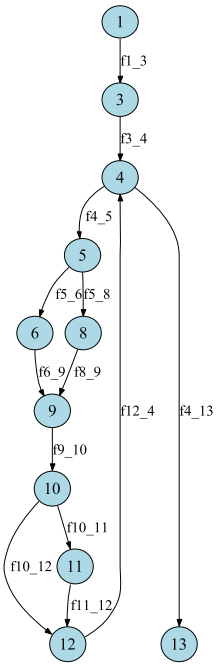
On modern processors, the execution time of a particular instruction depends on the history of instructions that have previously been issued. Bound-T handles this by simulating the processor pipeline. It does not, however, model any cache behavior.

Bound-T stores assertions computed in the second stage or user-provided in the third stage in a separate file. This simplifies to support multiple execution contexts for each function. The assertion language itself is designed for flexibility and generality in order to allow for annotations of both assembler code and high-level languages programs. Assertions are stated for a specific *scope* (= subprogram, loop or call). A scope is identified by its name or—in the case of loops—its nesting level. Debug information is used to locate entry points of functions in the object code. This allows for sophisticated specifications, e. g., loops which are nested inside other loops, or loops which call a particular subprogram. For illustration, consider the example of Fig. 10.

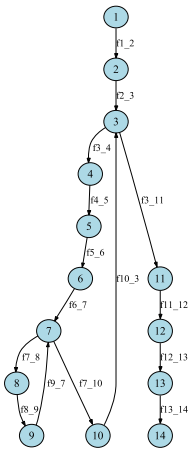
Conceptually, the annotations are driven by the structure of the high-level language sources but they are closely tied to the object code. Bound-T uses the object code as the basis for its calculations. In principle, it thus gains language and compiler independence. A drawback, however, is that annotations are limited to anchors, i./,e., to program constructs that can be recognized after the compilation and possibly applied

Table 2 Flow information benchmarks and annotation examples, part II

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 to the same boolean value within each iteration of the while-loop.
	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 }
			1 func TAL_cond(A_COUNT, B_COUNT) {
			2 block blk1, blk6, blk8, blk11;
			3 loop lp1;
			4 blk1#begin= "-v-LA_1";
			5 blk1#end = "-^-LA_13";
			6 lp1#begin = "-v-LA_4";
			7 lp1#count = 100;
			8 lp1#end = "-^-LA_12";
			9 blk6#begin= "-v-LA_6";
			10 blk6#end = "-^-LA_7";
			11 blk8#begin= "-v-LA_8";
			12 blk8#end = "-^-LA_9";
			13 blk11#begin = "-v-LA_11";
			14 blk11#end = "-^-LA_12";
			15 return(blk1#time
			16 -(min(0,100-A_COUNT)*blk6#time
			17 -(min(100,A_COUNT) *blk8#time
			18 -(min(0,100-B_COUNT)*blk11#time);
			19 }
			
			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 to approximate $\mathbf{B}_1$ and $\mathbf{B}_2$.

B_2: Explicit execution frequency *Side constraints:* The execution frequency of the statement at line 6 is less or equal to the one at line 8.			

B_3: Sub-ranges of loop iterations
 	
In this benchmark, the challenge lies in expressing the triangle loop (The program computes the sum $\sum_{k=1}^{n} k$).
 	
Side constraints:
None.
	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 }
			1 func TAL_compute_sum(N) {
			2 block blk1; loop lp3, lp7;
			3 blk1#begin= "-v-LA_1";
			4 blk1#end = "-^-LA_14";
			5 lp1#begin = "-v-LA_3";
			6 lp1#count = N;
			7 lp1#end = "-^-LA_11";
			8 lp2#begin = "-v-LA_7";
			9 lp2#count = N-1;
			10 lp2#end = "-^-LA_10";
			11 blk7#begin= "-v-LA_7";
			12 blk7#end = "-^-LA_10";
			13 return(blk1#time -
			14 N*(N-1)/2 * blk7#time);
			15 }

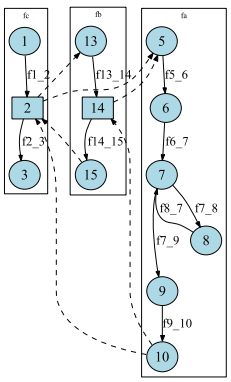
B_4: Call-context sensitive flow information
 	
Side constraints:
The upper bound of the loop starting at line 7 is 10 when `fa()` is called from `fc()` and 7 when it is called from `fb()`.
	1 int fc(int m, int n) {
	2 return fa(m) + fb(n);
	3 }
	4
	5 int fa(int i) {
	6 int j=0;
	7 while (j<i) {
	8 j++;
	9 }
	10 return j;
	11 }
	12
	13 int fb(int i) {
	14 return 8 + fa(i);
	15 }
			1 func TAL_fc() {
			2 return TAL_fa(10) +
			3 TAL_fb(7);
			4 }
			5 /* To be called as TAL_fa(10); */
			6 func TAL_fa(I_COUNT) {
			7 block blk5; loop lp7;
			8 blk5#begin = "-v-LA_5";
			9 blk5#end = "-^-LA_11";
			10 lp7#begin = "-v-LA_7";
			11 lp7#count = I_COUNT;
			12 lp7#end = "-^-LA_9";
			13 return(blk5#time);
			14 }
			15 func TAL_fb(I_COUNT) {
			16 return TAL_fa(I_COUNT);
			17 }

Table 1 continued

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: Ln means a reference to line n of the original code.
	Note: The side constraints for $\mathbf{B}_1$ are not directly expressible in WCETC.
	
	B_2:
	`1 void cond (int a[], int b[]) {`
	`2  int i=0, j=0;`
	`3  WCET_SCOPE(s1) {`
	`4  while (i < 100) WCET_LOOP_BOUND(100) {`
	`5  if (a[i] < 10) {`
	`6  j++;`
	`7  WCET_MARKER(M1);`
	`8  }`
	`9  else {`
	`10  a[i]=10;`
	`11  WCET_MARKER(M2);`
	`12  }`
	`13  i++;`
	`14  if (b[i] < 10)`
	`15  j++;`
	`16  }`
	`17  WCET_RESTRICTION(M1 ≤ M2);`
	`18  } /* scope s1 */`
	`19 }`
		B_1:
		`1 subprogram "cond"`
		`2  loop`
		`3  repeats 100 times;`
		`4  end loop;`
		`5 end "cond";`
		
		Note: Due to the lack of anchors in the original program, a finer granularity is not possible. The side constraints for $\mathbf{B}_2$ 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` $L3$`;`
	`1 #define N 100 /* max. value of n */`
	`2 int compute_sum(int n) {`
	`3  int a=0, b=0, i=n, j, y;`
	`4  WCET_SCOPE(s1) {`
	`5  while (i>0) WCET_LOOPBOUND(N) {`
	`6  a=a+1;`
	`7  i=i-1;`
	`8  j=i;`
	`9  while (j>0) WCET_LOOPBOUND(N-1) {`
	`10  b=b+1;`
	`11  j=j-1;`
	`12  WCET_MARKER(M);`
	`13  }`
	`14  }`
	`15  WCET_RESTRICTION(M ≤ (N*(N-1)/2));`
	`16  } /* scope s1 */`
	`17  y=a+b;`
	`18  return y;`
	`19 }`
		`1 subprogram "compute_sum"`
		`2  loop that contains (loop)`
		`3  repeats N_MAX times;`
		`4  end loop;`
		`5  loop that is in (loop)`
		`6  repeats N_MAX-1 times;`
		`7  end loop;`
		`8 end "compute_sum";`
		
		Note: This assumes that N_MAX is a constant value.

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`;`
	`1 int fc (int m, int n) {`
	`2  return fa(m) + fb(n);`
	`3 }`
	`4 int fa (int i) {`
	`5  int j=0;`
	`6  /* specific loop bound for call`
	`7  context fb(fa()) is not supported */`
	`8  while (j<i) WCET_LOOP_BOUND(10) {`
	`9  j++;`
	`10  }`
	`11  return j;`
	`12 }`
	`13 int fb (int i) {`
	`14  return 8 + fa(i);`
	`15 }`
		`1 subprogram "fa"`
		`2  loop`
		`3  repeats ≤ 10 times`
		`4  end loop;`
		`5 end "fa";`
		
		Note: According to [HLS05], Bound-T performs context sensitive analysis when loop bounds depend on function parameters. It is not possible to annotate context-sensitive information directly in Bound-T.

```
1loop that
2 is in (loop that calls "Foo")
3 and contains (loop that not calls "Bar"
4               and calls "Fee")
5 and not contains
6   (loop that calls "Fee2")
7repeats 10 times end loop
```

Fig. 10 Example of a Bound-T annotation (from [24])

optimizations. Examples are call and loop statements; if-then-else statements are not. A detailed description can be found in [25]. Extensions for an improved mapping between source and object code are announced for a forthcoming revision of the Bound-T annotation language [27].

7.4 The Mälardalen flow-facts annotation language

The Mälardalen flow-facts annotation language has been developed by Engblom and Ermedahl [12,13]. Flow facts are linear flow constraints with additional context information.

Engblom and Ermedahl assume that flow analysis is performed prior to processor–behavior analysis. This means that the flow analysis does not have access to information about the execution time of code constituents. The flow analysis identifies a set of infeasible paths of the CFG. Among the remaining set of CFG paths, the WCET calculation determines those that exhibit the actual WCET. To this end execution time information on machine instructions is used.

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 recursive function or a loop. The entry edges to the scope may also point to nodes other than the header node. In this case the scope describes an unstructured loop. All scopes are associated with a loop count, even if that is just 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 fact is composed of three components:

i. the name of the scope, where the fact is defined,
ii. a context specifier, and
iii. a constraint expression.

The context specifier allows to specify the iterations of the scope in which the constraint expression is valid. Context specifiers are defined by their *type* and *iteration space*. By means of types it can be specified that a fact is considered for the sum over all iterations (`total`) or that it is considered for each single iteration separately (`foreach`). This means that the annotated iteration space can either be all iterations (`all`) of one or more loops, or some specified sub-range of iterations.

7.5 WCETC

The WCETC language has been developed by Kirner [30]. The WCET analysis tool using this annotation language is called *calcwcet167* [29]. WCETC is based on ANSI C and provides language level constructs to annotate timing information. Kirner et al. designed WCETC as a case study for a technique to correctly transform flow annotations by the compiler from source-code to machine-code level [31]. This technique works even in case of complex code optimizations performed by the compiler.

The design of WCETC concentrates on the research of flow-facts transformation and thus introduces only syntactic annotation constructs needed for this. The language WCETC extends the marker concept of Modula/R (cf. Sect. 6.4) to express arbitrary linear flow constraints (cf. Sect. 4). Such "low-level" annotations have been considered sufficient to demonstrate transformation of flow facts.

The language WCETC provides the following annotations for timing analysis:

- `maximum` *ConstEspr* `iterations`: loop bound information that has to be specified for each loop.
- `scope` *Ident* `{` *Stmts Restrictions* `}`: Similar to Modula/R, a scope construct allows to specify flow facts relative to a concrete control-flow location. However, the marker statements in WCETC just label control-flow locations: `marker` *Ident*. The flow facts themselves are given at the end of the scope as a list of linear flow constraints: `restriction` *ConstExpr Ident* ... <= ... `;` Each restriction keyword is followed by exactly one linear flow constraint that may use any markers that are defined within the scope.
- `wcet_buildinfo (`*String* `)`: this construct allows to specify information about the compilation process or other information that may be relevant to document the context of the timing analysis.
- `addcycles (`ConstExpr `)`: this construct allows to add additional execution time cycles to the block where it is embedded. This can be used to probe the impact of local code optimizations on the WCET bound, respectively, the worst-case path. Furthermore, this statement can be used to add the cost for a function call in case that the code of the callee function is not available.
- `wcet_blockbegin (`*Ident* `,` *ConstExpr* `)` and `wcet_blockend (`*Ident* `)`: A special feature of the WCET tool chain developed by Kirner et al. is that it is not restricted to taking source code as input programs. It also supports higher program abstractions like

```
#if (LANG_WCET) == WCETC /* WCETC */
#define WCET_MARKER(x) marker x
#else /* ANSI C */
#define WCET_MARKER(x)
#endif
```

Fig. 11 (De-)activation of wcetC keywords

Matlab/Simulink models [35]. The two constructs are used to enable the back-annotation of the WCET results from machine-code level not only to the source code but also further back to the model level. These constructs help to identify the block boundaries of each Matlab/Simulink block inside the source code.

The timing annotations in WCETC have intentionally been integrated into the programming language instead of using compiler pragmas for specifying them. This has the advantage that the flow facts can be annotated exactly at the location where they describe the program behavior while preserving compatibility with standard C Compilers that do not understand the timing annotations. As shown in Fig. 11, this is achieved by using a header file that defines conditional preprocessor macro definitions for the corresponding annotation keywords.

7.6 aiS, the annotation language of aiT

Like Bound-T, aiT is a commercial WCET analysis tool [1]. It is developed by AbsInt Angewandte Informatik GmbH, Germany, and available for different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555.

aiT features a value analysis to automatically calculate flow information. Additionally, it can be supplied with various kinds of specifications and annotations that help it to perform WCET analysis and to improve the precision of the results. Specifications and annotations are provided in the so-called aiS format. aiT accepts binary files as argument programs for WCET analysis. To make this more effective, aiT supports a special kind of object code annotations to reconstruct the CFG from the object code [15,21]. These annotations allow, for instance, the user to annotate the possible targets of a jump instruction in order to guide the object-code parser when reconstructing the CFG.

Annotations often refer to *program points*. Program points can be described not only by an address or a routine name but also by more sophisticated descriptions, e. g., the third computed call in a particular routine, or the loop beginning in a specific source code line. Such descriptions consist of more atomic elements like numbers, addresses, or names for routines or files.

Source code lines in annotations are translated to code addresses. Program points may refer to code lines, either explicitly via file 'Name' line Number or implicitly via the keyword 'here', which refers to the line where it occurs. The source code lines are translated into code addresses by using the line information in the executable. The line information, however, is not always accurate. The rule is that line n refers to the first instruction associated with a line number $\geq n$.

The following types of aiS annotations are relevant for source-code analysis and source-code annotations:

- `loop here min` *m* `max` *n*: The loop body containing the annotation is executed at least *m* and at most *n* times each time the loop is first entered.
- `recursion` "*function*" `min` *m* `max` *n*: The function of the given name executes at least *m* and at most *n* recursive calls each time it is called from another function.
- `condition here is always[true` *or* `false]`: The branch condition has been determined to always evaluate to true or false, meaning that only one of the branches can ever be taken.
- `snippet here is never executed`: The piece of code containing the annotation is infeasible; it can never be executed.
- `accesses default` "*function*" `to` *addresses*: Inside the function denoted by the annotation, any memory access that could not be resolved by aiT is guaranteed to access an address within the specified range.
- `instruction here calls` *functions*: The statement the annotation is associated with makes a call through a function pointer; the functions listed are the possible targets of this call.
- `flow each (here) <=` *n* (*"function"*): This annotation kind demonstrates the flexibility of aiT's flow constraint language; the annotated program point within the given function is executed at most *n* times whenever the function is called.

The following example illustrates the annotation of a loop. Note that 'here' need not exactly denote the loop start address. It suffices that it resolves to an address somewhere in the loop.

```
for (i=3; i*i ≤ n; i+=2) {
  if (divides(i,n))/* ai: loop here
      max 20; */
    return 0;
}
```

The aiS language also offers annotation variables (register variables) for annotating context-sensitive information. Assignments to these variables may be placed in any

annotation and checked at other arbitrary locations in a program for specific values. The values of annotation variables are taken into account by the abstract interpretation when analyzing the program. For example, if different values are assigned to two branches of an if-statement, the analysis will propagate a range containing both values in aiT's value analysis. Annotation variables can also be used for enumerating loop-contexts as well as calling-contexts and properly check for such a context enumeration at an other point in the program. Properties of calls can also be annotated, for example, a call to return `immediately` or `never`. That a routine never returns is important when handling calls of the operating system.

7.7 FFX/F4, the annotation language of OTAWA

The *Open tool for adaptive WCET analysis* (OTAWA) is developed by Cassé et al. [11]. OTAWA is a framework for building research WCET analyzers, and is available under a free software license. It provides state-of-the-art WCET analysis using IPET and is specialized to work on binary programs.

Annotations in OTAWA can be specified in the *Flow Fact XML format* (FFX) which is an XML representation of flow information, and in the *Flow Facts File Format* (F4) which is a simpler textual format with a more compact syntax [3]. Annotations can be associated with a code location via an *address*, a pair of a *label* and *offset*, or a *source code line number*, with the caveat that compiler optimizations can make this mapping ambiguous.

The FFX language is hierarchically structured and provides several constructs to describe properties of functions, like `noreturn` for a function that will never return, and `nocall` for a function that should be ignored by the analysis. Functions are enclosed by the `<function>` tag. The annotations for sub-procedures can alternatively be nested into the `<call>` tag. The most important kind of control flow annotations are loop bounds. They are described by the loop tag, e. g.

```
<loop LOCATION maxcount="12"
    totalcount="20" />
```

In this example, *maxcount* denotes the maximum number of iterations in relation to the loop entry and *totalcount* the maximum number of iterations since the program start. Addresses in F4 can also describe specific call contexts, e. g.

```
loop ADDRESS ... in ADDRESS1 /
    ADDRESS2 / ... ;
```

The above example describes the loop located at `ADDRESS`, when its containing function is called by the function at `ADDRESS1`, that in turn was called from `ADDRESS2`, etc. If the sequence of addresses does not reach back to the task entry point, the annotation is valid for all contexts that share the specified prefix.

Since OTAWA is used to analyze binary programs, the annotation language F4 also contains a

```
checksum OBJFILE SUM;
```

element to ensure that the annotated binary has not been modified since the annotations were written.

7.8 Flow information in chronos 3.0

The Chronos WCET analysis tool has been developed by Li et al. [41]. It uses Integer Linear Programming (ILP) to statically compute a WCET bound for an argument program. Chronos is based on SimpleScalar, a cycle-accurate architectural simulator [Sim]. Chronos supports simplified processor models of SimpleScalar. The constraints these models have to obey involve omission of the data memory hierarchy, limiting the memory hierarchy to one level and several limits on jump prediction. The exact limitations can be found in the user manual [40]. Despite these limitations, microarchitectural features like *out-of-order* and *dynamic global branch prediction* can be modeled. By simulating the code using SimpleScalar, the WCET estimate of Chronos can be compared against the actual execution time.

Chronos does not have a specific annotation language on its own: the ILP problem is solved by *lp_solve* [43] or CPLEX [28]—the user may specify additional constraints for basic blocks (in strict ILOG/CPLEX format) to supply additional information, e.g., on infeasible paths or loop bounds. Version 3.0 of Chronos features also an automatic infeasible path analysis.

The graphical front-end also supports line numbers to ease the formulation of constraints. The example below shows the constraint "block c0.2 is executed 10-times as often as block c0.3", where c0 addresses function 0, with the basic-block number following after the dot.

```
c0.2 - 10 c0.3 = 0
```

Alternatively, this can be expressed using line numbers:

```
line25 - 10 line42 = 0
```

7.9 The annotation language of TuBound

TuBound is a portable tool for WCET analysis of C++ programs developed by Prantl et al. [22,53,56]. TuBound aims at

increasing the productivity of the programmer by allowing for high-level annotations to be placed in the source code and by reducing the need for user-assistance by providing multiple static program analyses at the source code level.

Annotations in TuBound serve two purposes: first, to provide a programmer-friendly user interface to support the timing analysis with domain-specific knowledge; second, to represent a textual intermediate form for results automatically generated by the static program analysis component. Since the result of the static analysis is attached to the program source code, the programmer can inspect it and then decide where to manually refine the annotations, thus keeping user-assistance at a minimum.

Annotation Syntax and Source Code Integration. TuBound uses the `#pragma`-directive to embed annotations into C++ sources. With this mechanism, it is possible to place annotations at each sequence point of the program. Currently, annotations comprise the following four kinds:

1. *Loop-bounds* Loop bound annotations can be located anywhere inside the scope of the loop body. They always refer to the loop construct that directly dominates the lexical scope containing the annotation. The annotation consists of a numerical expression that denotes an upper bound for the number of times a loop is executed in relation to the number of times the basic block that dominates the loop entry is executed. By convention, a special loop bound of -1 is used to indicate a diverging (main-)loop.
2. *Markers* The concept of a marker is closely related to labels to identify addressable units in the argument program. A marker creates a symbolic name for a basic block that can be used in constraint specifications. A marker is always associated with a scope. As default, this is the global scope.
3. *Constraints* This is the most generic and powerful annotation concept currently supported by TuBound. Constraints allow to express arbitrary relations between the execution counts of basic blocks, referred to via markers.
4. (Marker-)*scopes* Marker-scopes are syntactic sugar allowing to reduce the amount of typing when doing manual annotation. Consider two basic blocks b_1, b_2 where b_1 dominates b_2: An occurrence of a local marker $m_{2loc} \in b_2$ with a scope of b_2 is equivalent to the expression $m_2 * m_1$, where $m_2 \in b_2$ and $m_1 \in b_1$ are global markers.

Figure 12 shows an example which makes use of different kinds of annotation mechanisms supported by TuBound. The annotations express the constraint that the loop construct is executed up to 42 times upon entering the parent scope.

```
1 {
2 #pragma wcet_marker(m1)
3   do {
4 #  pragma wcet_marker(m2)
5 #  pragma wcet_marker(m3)
6 #  pragma wcet_scope(m3)
7
8 #  pragma wcet_constraint( m2 =< m1*42 )
9 #  pragma wcet_constraint( m3 =< 42 )
10 #  pragma wcet_loopbound( 42 )
11     ...
12   } while (!done);
13 }
```

Fig. 12 Annotations [8,9] and [10] carry the same information

8 Other annotation concepts

8.1 Symbolic annotations

Blieberger proposed an approach which combines aspects of a pure annotation language with those of a programming language extension [6]. The clue to 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. Like for for-loops, however, also for discrete loops the loop bounds can often automatically be computed by means of reasonably simple mathematical reasoning. The automatic computability is ensured by using annotations which describe the possible update range of the loop (induction) variable. Particularly well-suited for this purpose are methods for symbolic analysis. We thus use the term *symbolic annotation* for this approach here.

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

```
1 k:= ...;
2 discrete h := k in 1..N/2
3          new h := 2*h | 2*h+1 loop
4 <loop body>
5 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. The actual update of the loop variable `h` has

to be programmed explicitly by additional code. 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, if possible, or checked at run-time otherwise.

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 demanding, or impossible at all.

8.2 Assertion language

The *assertion language* was developed by Lisper [39]. It is a generic annotation language addressing annotations for quite different aspects of program behavior. It has not been specifically designed for WCET analysis but can be used for describing flow constraints for WCET analysis as well. The assertion language was derived from the existing assertion language used in Floyd/Hoare-style logic [16,26].

The assertion language allows to express assertions over program states. As usual, a program state is defined as a mapping of variables to values. In particular, `PC` denotes the value of the program counter. Assertions are basically constructed by first-order logic operators. For example, `(X+Y)[PC=L] < 17` says that the sum of the values of the variables `X` and `Y` is less than 17 at program point `L`. The form `e@L` is a synonym for `e[PC=L]`. In general, the construct `e[p]` denotes that the scope of expression `e` is bound to the program states where `p` holds.

Besides program variables, the expressions can also refer to execution counters, for example, `#L` is the execution count of program point `L`. Execution counters are initialized to zero at program start and are incremented by one whenever the corresponding program point is executed. For example, `#L <= 30` says that the statement at program point `L` is executed at most 30 times.

The assertion language supports also the specification of real-time constraints: `T[p]` denotes the elapsed time when reaching any of the states where the property `p` holds. For example, $\forall$`i.T[PL=L` $\wedge$ `I=i]` $\leq$ `T[PL=L'` $\wedge$ `I=i]` says that the elapsed time reaching program point `L` is always less than or equal to the elapsed time reaching program point `L'` whenever the value of variable `I` is the same.

The assertion language has been proposed as a case study for a generic annotation language. The language is Turing complete, i.e., it can be used to describe the behavior of arbitrary complex programs. It does not yet include mechanisms to modularize a program description. So far there is no WCET calculation method that can take expressions of the assertion language into account. For WCET analysis it could make sense to define a subset of the assertion language that is less expressive but allows for efficient analysis and WCET computation.

9 Annotation examples

In order to highlight the different capabilities and annotation mechanisms we created a set of benchmarks together with appropriate flow information which allow to discriminate them according to the comparison criteria for WCET annotation languages introduced and discussed in Sect. 5. We then formulated the specified flow information in multiple annotation languages (cf. Table 2). To this end we picked four languages, which each stand as a representative for a class of similar annotation languages:

Language	*Represents*
TAL	Hierarchy-oriented (Timing schema) + Others
PL/IDL	Path-oriented (EPET)
WCETC	Constraint-oriented (IPET)
Bound-T	Industrial language (IPET)

In detail: TAL is selected because it represents a hybrid of hierarchy-oriented annotations and other concepts. TAL provides the programmer only with a hierarchical skeleton of the program and, in consequence, a lot of freedom and responsibility. PL and IDL are selected because they represent the singular explicit path-oriented annotation languages. WCETC is selected because it prototypically represents the many constraint-oriented annotation languages. Bound-T, though also constraint-oriented, is selected, because it represents a current commercially marketed annotation language.

In Table 2 the languages are compared by using the same benchmark where each selected language is one representative of its respective group as explained in the beginning of this section. Four such comparisons are shown in Table 2 ranging from execution order, to loop iterations, and context-sensitive information. Each of those four benchmarks B_1 to B_4 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 CFGs in the third column. The annotated examples for each annotation language are presented in the subsequent columns.

10 Comparison

The grouping of WCET annotation languages according to the WCET calculation method utilized by the WCET analysis tools using them yields a first classification scheme of WCET annotation languages as we have seen in the previous sections (cf. Fig. 1).

In this section we complement this scheme by a second one. This second scheme is given by the set of orthogonal classification criteria introduced and discussed in Sects. 3 and 5. These criteria refer to features and properties which are inherent to the annotation languages themselves and are pivotal for their suitability and usefulness in theory and practice.

Together, the two schemes span a two-dimensional space, in which we classify each of the languages. The results of this classification are summarized in Table 1. They allow us to identify and compare the relative strengths and limitations of the languages from different angles.

10.1 Language design

Expressiveness The expressiveness of annotation languages is closely tied to the calculation method applied by the tools using them. As an example, in Table 1 the linear-flow-constraint-based calculation methods show more features than the hierarchical computation methods. This can be attributed to the more generic nature of data representation and the calculation method. We consider this a consequence of the fact that annotation languages were often designed ad hoc by tool developers in need of a user interface.

For our comparison, we focus on the six kinds of flow information discussed in Sect. 3 that users typically want to annotate. These kinds span a spectrum ranging from obvious (such as loop bounds) to fairly advanced ones (such as execution order).

Loop bounds Loop bounds are indispensable for successful WCET analysis. It is thus not surprising that all annotation languages in Table 1 support annotations for the specification of loop-bounds.

Triangle loops Constraint specifications with inequalities allow a more precise description of the iteration space of triangle-loops than plain loop bounds. Inequalities are supported not only by the linear flow constraints based IPET tools, such as Bound-T, but also by other approaches. This includes timing schema and symbolic annotations. In case of Bound-T, triangle-loops can only be annotated, if they contain an anchor allowing to identify them, such as a call statement.

Call contexts TAL, for example, supports call context annotations in its calculation schema through a functional abstraction: It is possible to define multiple (timing-)functions to annotate one function of the program and choose a different one for each call context. Bound-T, on the other hand, does not expose context-sensitive information to the annotation language. Nonetheless, it is aware of call context information during the automatic computation of loop-bounds.

Loop contexts As described in Sect. 7.4, the first and subsequent loop iterations behave often very differently, e. g., because of cache effects. The precision of analyzing such loops would be improved, if annotation languages would allow for annotating this accordingly. Heptane and the Mälardalen Language directly support loop contexts.

Application contexts SPARK Ada provides a unique feature called *modes* to describe application contexts.

Execution Order Only PL and IDL, which focus entirely on modeling execution paths, allow to explicitly describe the execution order. Neither constraint-based nor hierarchical methods support this.

Annotation Placement and Abstraction Level For the programmer it is usually easier and more convenient, if annotations can directly be placed in the source code instead of a separate file. However, source code annotations may affect the readability of the program, especially in the case of library functions, which usually have many different call sites. Another important argument against direct source code annotations is that in a production setting any modification of the source code may require a new audit.

The decision of where to place annotations is also closely tied to the question of the abstraction level provided by the tool. Since object code annotations are impractical, all tools of Table 1 that operate on the object code level (TAL, Bound-T, aiS) choose to place the annotations in a separate file.

In general, however, the following three choices of annotation placement are possible:

Abstraction Level	*Annotation placement*	
Source Code	1. Inside source code	2. External file
Object Code	(*not practicable*)	3. External file

Choosing the low-level representation, i. e., the object-code abstraction, gains independence from the compiler, but complicates the development phase where the source code is frequently changing. Choosing the high-level representation, i. e., the source-code abstraction, the interaction with the compiler becomes a delicate issue as optimizations that change the control flow may invalidate the annotations.

Programming Language In principle, a WCET annotation language becomes independent of a particular programming language, if it focuses entirely on the object code. This advantage, however, is hardly exploited. For practical reasons therefore many of the surveyed annotation languages focus on subsets of the C language that is typically used for implementing applications.

10.2 Intuitiveness

Typically, there is a trade-off between the expressiveness and the complexity of annotations. This trade-off affects how intuitively they can be used. Getting annotations describing complex constraints right, can be costly and error-prone.

TAL, for example, leaves many aspects of the WCET calculation to the user. It is possible to specify almost arbitrary formulæ within a TAL-script. With this freedom it is theoretically possible to achieve the highest precision, but it also demands an unrealistically high effort, since the user has to implement details that later approaches would integrate into the WCET calculation tool.

SPARK Ada, on the other hand, only supports loop-bounds that are annotated directly into the source code. This is much less demanding (albeit not of particularly high expressiveness).

The annotation languages of aiT and Bound-T, finally, strive for a balance. They support tailored language constructs for different kinds of flow information in order to limit the demand on the user.

10.3 Tool availability

WCET Tool Most of the annotation languages considered in this article are developed at academic institutions. aiS and Bound-T are notable exceptions; they are currently being marketed as commercial products. According to Praxis High Integrity Systems, a future release of a SPARK Ada-based source code annotation language is in progress.[3]

Industrial Application of the Tools All tools that are listed in Table 1, including the academic prototypes TAL, SPARK Ada, and WCETC/*calcwcet167* (see column 'WCET Tools'), have been applied in numerous industrial application scenarios, or at least for analyzing specific industrial applications. The annotation languages of the academic tools Heptane, WCETC, F4/FFX, Chronos, TuBound as well as the Mälardalen Language are still under active development.

Annotation Producer/Annotation Consumer The last two columns of Table 1 report whether the WCET analysis tool resp. language approach (Real-time Euclid, WCETC, Symbolic Annotations) consumes given annotations, or if it also actively produces annotations (possibly by a separate analysis tool). Roughly, a third of the tools also actively produces annotations.

10.4 Summing up

The findings summarized in Table 1 show that each of the WCET annotation languages considered takes care of a specific set of programming language constructs, for which it provides support to annotate them. Obviously, each of the constructs supported by some annotation language is useful to support; but each of the languages fails to support all of them. There is no annotation language which uniformly outperforms the others. Instead, each of them has its own individual strengths and limitations. This would become even more obvious, if we were to take further criteria into account, e. g., the possibility and ease of reconstructing the CFG on the object-code level such that it precisely reflects its counterpart on the source-code level [36], or if we were to consider application domains of annotation languages beyond plain WCET analysis, e. g., such as aimed at by Lisper with the *assertion language*.

Moreover, there are many reasonable and practically justified demands on WCET annotation languages, which are not yet met by any of the languages. One such demand is that a WCET annotation language should support annotations of both the source code and object code of a program (Bound-T and aiT already have limited support for annotating both source and assembler code). Annotating the source code is usually less costly and error-prone. Often, however, the source code is not available (think of library code for example), or source code annotations might be invalidated by complex and intransparent optimizations applied by a compiler. In such cases it is inevitable to annotate the object code. Another demand is that WCET annotation languages should allow for the annotation of characteristics which do not directly refer to the application program but to the processor architecture, the operating system, the run-time environment, the programming language and compiler, the compiler optimizations, etc., which are most relevant for WCET analysis, too.

We believe that uniform support for all this by a unifying WCET annotation language would be most beneficial. Such a language should also cover the features supported by current WCET annotation languages in order to qualify it as a widely usable exchange format for WCET analysis. By enabling the mutual supplement of the various tools, this would yield a strong impetus for advancing the field of WCET analysis as a whole.

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

11 Conclusions and perspectives

There is a significant body of work on WCET annotation languages. The precision, generality, and ease of use of WCET analysis tools depend much on the kind and the expressiveness of the annotation language used to feed the tool with program-specific timing-relevant information. The choice of the annotation language is a 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 versus ease of use and analyzability, make the choice of a "good" language a challenge of its own. It is thus by no means surprising that annotation languages attracted so much attention of researchers working on WCET analysis and that so many different kinds of annotation languages have been proposed and used by WCET analysis tools.

The findings of our comparison, which are summarized in Table 1, show that none of the annotation languages is uniformly superior to the others. All of them leave room for improvement in one way or the other. The *WCET Annotation Language Challenge* has been proposed to strengthen research efforts towards a universal WCET annotation language for a wide range of analysis tools [32,65].

We plan to contribute towards mastering this new challenge. As a first step, we have proposed a list of ingredients which we consider essential for a universal WCET annotation language [33]. Currently, we are working on an extended and refined version of this proposal, which shall be the basis for a concrete language proposal.

Acknowledgments We gratefully acknowledge the helpful comments of the anonymous referees and the feedback of the participants of the WCET'07 workshop. Especially, we would like to thank Niklas Holsti and Henrik Theiling for their many and very detailed comments, which helped to clarify and improve the presentation of this article.

References

1. AbsInt. aiT. Web page. http://www.absint.com/aiT. Accessed online in February (2010)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley, New York (2007). ISBN 0-321-48681-1
3. Ballabriga, C., Cassé, H., Nemer, F., Rochange, C., Sainrat, P.: OTAWA. Online Program Documentation. University of Toulouse, France. http://www.otawa.fr/ (2008)
4. Bernat, G., Colin, A., Petters, S.M.: WCET analysis of probabilistic hard real-time systems. In: Proceedings of the 23rd Real-Time Systems Symposium, December, pp. 279–288, Austin, TX, USA (2002)
5. Bodík, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. SIGSOFT Softw. Eng. Notes **22**(6), 361–377 (1997)
6. Blieberger, J.: Discrete loops and worst case performance. Comp. Lang. **20**(3), 193–212 (1994)
7. Chapman, R., Burns, A., Wellings, A.: Integrated program proof and worst-case timing analysis of SPARK Ada. In: Proceedings of the ACM Workshop on Language, Compiler and Tool Support for Real-time Systems, pp. K1–K11, June (1994)
8. Chapman, R., Burns, A., Wellings, A.: Combining static worst-case timing analysis and program proof. Real-Time Syst. **11**(2), 145–171 (1996)
9. Chen, M.: A Timing Analysis Language—(TAL). Department of Computer Science, University of Texas, Austin, TX, USA (1987). Programmer's Manual
10. Colin, A., Puaut, I.: A modular and retargetable framework for tree-based WCET analysis. In: Proceedings of the 13th Euromicro Conference on Real-Time Systems, pp. 37–44, Delft, Netherland, June (2001). Technical University of Delft
11. Cassé, H., Sainrat, P.: OTAWA, a framework for experimenting WCET computations. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse, 25/01/06-27/01/06, page (electronic medium), http://www.see.asso.fr, January, See p. 8 (2006)
12. Engblom, J., Ermedahl, A.: Modeling complex flows for worst-case execution time analysis. In: Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS), December, Orlando, FL, USA (2000)
13. Ermedahl, A., Engblom, J., Stappert, F.: A unified flow information language for WCET analysis. In: Proceedings of the 2nd International Workshop on Worst Case Execution Time Analysis. Technical University of Vienna, Austria, June (2002)
14. Ermedahl, A., Stappert, F., Engblom, J.: Clustered worst-case execution time calculation. IEEE Trans. Comp. **54**(9), 1104–1122 (2005)
15. Ferdinand, C., Heckmann, R., Theiling, H.: Convenient user annotations for a WCET tool. In: Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, pp. 17–20, Porto, Portugal, July (2003)
16. Floyd, R.: Assigning meaning to programs. In: Proceedings of the AMS Symposia in Applied Mathematics, pp. 19–32 (1967)
17. Gustafsson, J.: The WCET tool challenge 2006. In: Preliminary Proceedings of the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation, pp. 248–249, Paphos, Cyprus, November (2006)
18. Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D., Harmon, M.G.: Bounding pipeline and instruction cache performance. IEEE Trans. Comp. **48**(1) (1999)
19. Hecht, M.S.: Flow Analysis of Computer Programs. Elsevier, North-Holland (1977)
20. Heptane (Hades embedded processor timing analyzer). Tool web page: http://www.irisa.fr/aces/work/heptane-demo/heptane.html. Accessed online in February (2010)
21. Heckmann, R., Ferdinand, C.: Combining automatic analysis and user annotations for successful worst-case execution time prediction. In: Embedded World 2005 Conference, February, Nürnberg, Germany (2005)
22. Holsti, N., Gustafsson, J., Bernat, G., Ballabriga, C., Bonenfant, A., Bourgade, R., Cassé, H., Cordes, D., Kadlec, A., Kirner, R., Knoop, J., Lokuciejewski, P., Merriam, N., de Michiel, M., Prantl, A., Rieder, B., Rochange, C., Sainrat, P., Schordan, M.: WCET 2008—Report from the Tool Challenge 2008. In: Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis, July, pp. 149–171, Prague, Czech Republic (2008)
23. Harbour, M.G., Garcia, J.J.G., Gutierrez, J.C.P., Moyano, J.M.D.: MAST: Modeling and analysis suite for real time applications. In: Proceedings of the 13th Euromicro Conference on Real-Time

Systems, pp. 125 –134, Delft, The Netherlands. Euromicro (2001)
24. Holsti, N., Långbacka, T., Saarinen, S.: Worst-case execution time analysis for digital signal processors. In: European Signal Processing Conference 2000 (EUSIPCO 2000) (2000)
25. Holsti, N., Långbacka, T., Saarinen, S.: Bound-T timing analysis tool User Manual. Tidorum Ltd (2005)]
26. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
27. Holsti, N.: Bound-T assertion language: Planned extensions. Technical report. Tidorum Ltd, (2005)
28. IBM: IBM ILOG CPLEX—High-performance mathematical programming engine. Web page. http://www.ibm.com/software/integration/optimization/cplex/. Accessed in Feb. (2010)
29. Kirner, R.: User's Manual—WCET-Analysis Framework based on WCETC. Vienna University of Technology, July, Vienna, Austria, 0.0.3 edition, 2001. available at http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/
30. Kirner, R.: The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2002)
31. Kirner, R.: Compiler Support for Timing Analysis of Optimized Code: Precise Timing Analysis of Machine Code with Convenient Annotation of Source Code. VDM Verlag, Germany, July 2008. ISBN: 978-3-8364-6883-1
32. Kirner, R., Knoop, J., Prantl, A., Schordan, M., Wenzel, I.: WCET analysis: the annotation language challenge. In: Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, pp. 83–99, Pisa, Italy, July (2007)
33. Kirner, R., Kadlec, A., Puschner, P., Prantl, A., Schordan, M., Knoop, J.: Towards a common WCET annotation language: essential ingredients. In: Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis, pp. 53–65, Prague, Czech Republic, July (2008)
34. Kirner, R., Kadlec, A., Puschner, P.: Precise worst-case execution time analysis for processors with timing anomalies. In: Proceedings of the 21st Euromicro Conference on Real-Time Systems, pp. 119–128, Dublin, Ireland, July 2009. IEEE, New York
35. Kirner, R., Lang, R., Freiberger, G., Puschner, P.: Fully automatic worst-case execution time analysis for Matlab/Simulink models. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems, pp. 31–40, Vienna, Austria, June 2002. Vienna University of Technology. IEEE, New York
36. Kirner, R., Puschner, P.: Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In: Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis, Palma, Spain, July (2005)
37. Klingerman, E., Stoyenko Alexander, D.: Real-time euclid: a language for reliable real-time systems. IEEE Trans. Softw. Eng. **12**(9), 941–989 (1986)
38. Lokuciejewski, P., Falk, H., Marwedel, P., Theiling, H.: Wcet-driven, code-size critical procedure cloning. In: Falk, H. (ed.) SCOPES, ACM International Conference Proceeding Series, vol. 296, pp. 21–30 (2008)
39. Lisper, B.: Ideas for annotation language(s). Technical Report Oct. 25, Department of Computer Science and Engineering, University of Mälardalen (2005)
40. Li, X., Liang, Y., Mitra, T., Roychoudhury, A.: Chronos user manual. Web page. http://www.comp.nus.edu.sg/~rpembed/chronos/chronos_manual.pdf. Accessed online in February (2010)
41. Li, X., Liang, Y., Mitra, T., Roychoudury, A.: Chronos: A timing analyzer for embedded software. Science of Computer Programming, **69**(1–3):56–67, (2007). http://www.comp.nus.edu.sg/~rpembed/chronos
42. Li, Y.-T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Proceedings of the 32nd ACM/IEEE Design Automation Conference, June, pp. 456–461 (1995)
43. lpsolve. Tool web page: http://lpsolve.sourceforge.net/. Accessed online in February (2010)
44. Lundqvist, T., Stenström, P.: Timing analysis in dynamically scheduled microprocessors. In: Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS), December, pp. 12–21 (1999)
45. Mok, A.K., Amerasinghe, P., Chen, M., Tantisirivat, K.: Evaluating tight execution time bounds of programs by annotations. In: Proceedings of the 6th IEEE Workshop on Real-Time Operating Systems And Software, May, pp. 74–80, Pittsburgh, PA, USA (1989)
46. *Maxima Manual*, 5.18 edn. Available online at http://maxima.sourceforge.net/docs/manual/en/maxima.pdf
47. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997). ISBN 1-55860-320-4
48. MacNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Trans. Electron. Comp. **9**(39–47), (1960)
49. OMG. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE). Object Management Group, June (2008)
50. Park, C.Y.: Predicting Deterministic Execution Times of Real-Time Programs. Ph.D. Thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02
51. Park, C.Y.: Predicting program execution times by analyzing static and dynamic program paths. Real-Time Syst. **5**(1), 31–62 (1993)
52. Puschner, P., Koza, C.: Calculating the maximum execution time of real-time programs. J. Real-Time Syst. **1**, 159–176 (1989)
53. Prantl, A., Knoop, J., Kirner, R., Kadlec, A., Schordan, M.: From trusted annotations to verified knowledge. In Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis, Dublin, Ireland, June (2009)
54. Park, C.Y., Shaw, A.C.: Experiments with a program timing tool based on a source-level timing schema. Computer **24**(5), 48–57 (1991)
55. Puschner, P., Schedl, A.V.: Computing maximum task execution times—a graph-based approach. J. Real-Time Syst. **13**, 67–91 (1997)
56. Prantl, A., Schordan, M., Knoop, J.: TuBound—a conceptually new tool for worst-case execution time analysis. In: 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008), pp. 141–148, Prague, Czech Republic, 2008. ISBN: 978-3-85403-237-3
57. RAPITA Systems Ltd. Worst-case execution time analysis. White Paper (Automotive), Rev. 1.32, 21st Sep. (2006)
58. Stappert, F., Altenbernd, P.: Complete worst-case execution time analysis of straight-line hard real-time programs. J. Syst. Archit. **46**(4), 339–355 (2000)
59. Shaw, A.C.: Reasoning about time in higher level language software. IEEE Trans. Softw. Eng. **15**(7), 875–889 (1989)
60. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, chapter 7, pp. 189–233. Prentice Hall, Englewood Cliffs (1981)
61. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3), 594–614 (1981)
62. Tan, L., Echtle, K.: 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 pp.
63. Vrchoticky, A.: Modula/R—Language Definition. Technical Report 02/1992, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040, March. Vienna, Austria (1992)
64. Vrchoticky, A.: Compilation support for fine-grained execution time analysis. In: Proceedings of the ACM SIGPLAN Workshop

on Language, Compiler and Tool Support for Real-Time Systems, June. Orlando FL (1994)
65. WCET annotation language challenge. Web page: http://costa.tuwien.ac.at/languages.html. Accessed online in Feb. (2010)
66. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckman, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenstrom, P.: The worst-case execution time problem—overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. (TECS) **7**(3), (2008)
67. Wenzel, I., Kirner, R., Rieder, B., Puschner, P.: Measurement-based timing analysis. In: Proceedings of the 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Porto Sani, Greece (2008)
68. Wright, F.: Computing with Maple. Crc Mathematics Series. Chapman & Hall, London (2001)
69. Wenzel, I., Rieder, B., Kirner, R., Puschner, P.: Automatic timing model generation by CFG partitioning and model checking. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05), March, pp. 606–611. Munich, Germany. IEEE New York (2005)

Author Biographies

Raimund Kirner is Assistant Professor at the Institute of Computer Engineering of the Vienna University of Technology. The research focus of Kirner is on system reliability, especially worst-case execution time analysis of real-time programs, including compiler support and design methodologies to make systems predictable. Currently, Raimund Kirner is principal investigator of three research projects. R. Kirner chaired the PC of WDES 2006 and WCET 2008. He is a member of the IEEE Computer Society, the ACM, the IFIP WG 10.2 (Embedded Systems) and the Austrian Computer Society (OCG).

Jens Knoop is a full professor at the Faculty of Informatics at the Vienna University of Technology, where he leads the Programming Languages and Compilers Group. His research interests include programming languages and compilers, especially algorithms and tools for static analysis, verification, and optimization including resource analysis for safety-critical embedded real-time systems. He is the Programme Committee Co-Chair and Chair of PACT 2010 and CC 2011, a member of the IFIP Working Group 2.4 on Software Implementation Technology, and elected board member of the European Association for Programming Languages and Systems (EAPLS), the European Association of Software Science and Technology (EASST), and the Austrian Computer Society (OCG).

Adrian Prantl is PhD student at the Institute of Computer Languages at Vienna University of Technology. He is currently employed for the FWF-funded project "Compiler Support for Timing Analysis" (CoSTA) were he is working on high-level analysis and optimizing program transformations. He is the main author of the TuBound WCET analysis tool, which supports source-based flow annotations and annotation-aware source-to-source program optimizations. He is also a contributing author of SATIrE, an open-source framework for static program analysis of C++ programs.

Markus Schordan is Deputy Program Director for Multimedia and Software Engineering and also for Game Engineering and Simulation at the Institute of Computer Science at University of Applied Sciences Technikum Wien. His research interests include static analysis of object-oriented languages, worst-case execution time analysis, source-to-source transformation, high-level optimization, and parallelization. In 2009 he received an R&D 100 Award as co-author of the compiler infrastructure ROSE. Furthermore he initiated the development of the SATIrE framework, which is based on ROSE and integrates tools for static analysis. In 2008 he co-organized a Dagstuhl Seminar on Scalable Program Analysis and serves as PC member in conferences such as JMLC, SYNASC, and PACT 2009.

Albrecht Kadlec has spent seven years developing compilers for digital signal processors and embedded processors at the companies ATAIR and Mentor Graphics. At Mentor Graphics, he was lead engineer of compiler development. In 2007, he joined the CoSTA research team at the Institute of Computer Engineering of the Vienna University of Technology to obtain a PhD. His research centers around timing analysis and computer architectures, focusing on compiler countermeasures to timing anomalies.