Essential Ingredients for a WCET Annotation Language*

Technical Report 10/2008, *rev. March* 12*th*, 2008

Raimund Kirner, Albrecht Kadlec, Adrian Prantl, Markus Schordan, Jens Knoop
Vienna University of Technology, Austria

Abstract

Ambitions towards the definition of common interfaces and the development of open frameworks have been started within the last years to increase the efficiency of research on WCET analysis. Towards common interfaces the Annotation Language Challenge *for WCET analysis has been proposed.*

Within this paper we present a list of essential ingredients for a common WCET annotation language. The ingredients we have selected are a summary of features available in different WCET analysis tools, extended by several new concepts we identified. The annotation concepts are described conceptionally to ease instantiation at different representation levels.

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

1 Why a Common WCET Annotation Language?

The situation for WCET analysis is very heterogeneous. It is a well known fact, that manual annotations are needed to supplement non-perfect analyzes. Various tools exist in various stages of sophistication. However, as the *WCET Tool Challenge* [8] has shown, few tools share the same target, analysis method or annotation language.

While a multitude of targets is beneficial, and a diversity in tools and methods is favorable, a common annotation language is *required* for an accepted set of benchmarks in order to evaluate the various tools and methods. Still as a direct consequence of the first WCET tool challenge a set of accepted benchmarks is already being collected, without such annotation support.

To enable annotations within these benchmarks, the *WCET Annotation Language Challenge* [13] has formulated the need for a common annotation language. This language has the task of specifying the *problem-inherent information* in a tool- and methodology-*independent* way, supporting e.g.: static analysis equally well as measurement based methods, thus allowing the combination of their results. It also has the difficult task to enable annotations at the *source* level, which is the natural specification level, as well as allowing the annotation of binary or object code, if the source code is not available, like for e.g.: operating systems or libraries.

This common language may allow the tool developers to concentrate on their analysis methods, creating interchangeable building blocks within the timing analysis framework, as intended by ARTIST2 [11]. Using this common annotation format as a common interface, tools can evaluate the same set of sources for a fair comparison of performance and may exchange analysis results to synergetically supplement each other. Thus, the steps of manual annotation, automatic annotation and timing analysis can be repeated, iteratively refining the analysis results.

This all should foster common established practices and may, eventually, lead to standardization, leading to a broader dissemination of WCET analysis throughout research and industry.

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

2 Basic Concepts

2.1 Definitions

Flow Information We define *flow information* to be any information about the control or data flow of a program code. Typical examples of flow information are loop bounds or descriptions of (in)feasible paths.

Timing Information We define *timing information* to be any information that is introduced in order to describe the search space of the WCET analysis. Because control and data flow represents the basis for the WCET analysis, the *flow information* of a program is always part of the *timing information*. An example of *timing information* that is not also *flow information* would be the specification of access times of different memory areas.

Information versus Annotation We distinguish between the *timing information* and the *timing annotation* of program code. The timing information is the information per se and the timing annotation is the linkage of the timing information with the program code.

There are different techniques possible of how to annotate the program code with timing information. For example, one possibility to annotate the program is to write the timing information directly into the source code, either as native statements of the programming language or as a special comment. Another possibility of annotation is to write

Above arguments apply for control-flow annotations that are placed directly inside the program code. How about, if control-flow annotations are placed in separate files? A common syntax would make sense if a programmer has to annotate the program modules at different representation levels.

2.2 Invariants versus Overrules

Program annotations in WCET analysis are typically assumed to describe a superset of the possible program behavior, i.e., program invariants. We extend this annotation concept to information that doesn't have to be a superset of the program behavior. We call all timing information that describes a superset of the possible program behavior *timing invariants*. In contrast, we introduce *timing overrules* as arbitrary timing information the user wants to be used for WCET analysis. We add a flag to each timing annotation to specify whether it is about timing invariants or timing overrules.

To give a precise criterion of whether a timing information is an *invariant* or an *overrule*, we define SB_F to be the set containing all feasible system behavior and $SB(I)$ to be the potential system behavior allowed by a timing information I and the syntactical control-flow structure of the program code. For each timing information I_{inv} that is an *invariant* it holds that

$$SB_F \subseteq SB(I_{inv})$$

For each timing information I_{ovr} that is an *overrule* it holds that

$$SB_F \not\subseteq SB(I_{ovr})$$

Timing overrules have a higher priority than any information given implicitly by the possible program behavior. In case an analysis tool can check that given overrules and invariants are in conflict, a warning should be given. However, depending on the complexity of the annotations and the analysis method, it might be possible to identify the conflicting annotations. If the conflict between overrules and invariants cannot be resolved by giving priority to overrules, an error should be reported.

2.3 Layers

The WCET of a program cannot be determined precisely without knowing information about the execution platform of the program. The execution platform of a program includes, for example, the development tools, any operating system, the hardware, and the application environment. Naturally, the execution platform is sliced into layers to benefit from the independence of different parts of the execution platform. For example, the operating system is an optional layer that may be placed on top of the hardware layer, and again, the layer of the development tool chain may be on top of the operating system.

These platform layers allow the reuse all *timing annotations* that address only properties of platform layers above any platform layer to be changed. For example, if we change the processor type but still use exactly the same code binary, any timing information describing the behavior of the compiler can still be reused.

Interestingly, the classification of whether a timing information is an invariant or an overrule can depend on the considered level of platform layers. For example, when looking at the operation environment we might see that a system has only four sensors, thus the loop for polling the sensors has a loop bound of 4, which is an invariant at the operation layer. But if we only look at the source-code layer then it is not known, how the

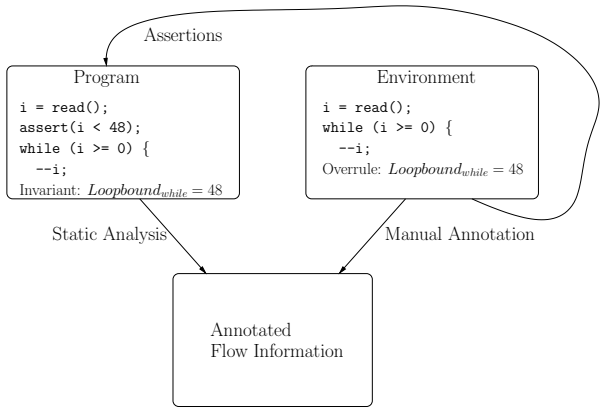


Figure 1. Lifting environmental information to the program layer

program will be used. Thus the same loop bound of 4 will be an overrule at the source-code layer.

2.4 Testing of Invariants

As we have seen in Section 2.3, it is possible for timing information to originate from the execution environment. Manual annotation of assumptions about the environment is potentially hazardous and may yield incorrect WCET estimates. It is possible, however, to "lift" environmental information to the program information layer, e.g., by inserting range checks and similar assertions wherever appropriate. The connection between these two approaches is illustrated in Figure 1. These kinds of assertions can easily be generated by an automatic tool and could be valuable for diagnose and testing of annotations. An example of using runtime checks with special support by the compiler is ModulaR [17].

By lifting annotations from the platform layers to the program layer, the resulting program becomes a specialized instance of the original program. These specializations may also improve the code performance if it allows the compiler to perform additional code optimizations.

3 Ingredients of the Basic WCET Annotation Language

In the following we describe essential ingredients for a WCET annotation language. The different timing informations are described at a conceptional level without focusing on the concrete syntax of an annotation language. The instantiation of a concrete syntax is left as a separate step. We focus on timing information that is somehow connected with the program code. Timing information that is not connected to the program code, like the description of a cache implementation, is left out to be better directly specified to the WCET analysis tool.

The code examples we use to motivate the usefulness of the different timing informations are given in ANSI C.

T1 Annotation Categorization

We define attributes for timing information to categorize and group them. These categorization attributes help to organize, check, and maintain timing annotations.

T1.1 Specification of annotation class

The annotation class is an optional attribute of timing information in general. As described in Section 2.2, besides the *invariants* we introduce so-called *overrules* as an additional class of timing information. A timing information should therefore contain a flag that indicates whether it is an *invariant* or an *overrule*.

Invariants (default): If not specified explicitly, a timing information is by default assumed to be an *invariant*. An *invariant* just makes more explicit what is already given by the semantics of the system.
For example, given the following code, a specified upper loop bound of 10 is an *invariant*:

```
1       for (i = 0; (i < 10); ++i) {
2         a[i] = b[i];
3       }
```

Overrules: As *overrules* are used to exclude feasible system behavior, it is important to explicitly mark such timing information as *overrule*. WCET analysis based on *overrules* may underestimate the real WCET. On the other side, classifying invariants mistakenly as *overrules* may result in unexpected warnings by the WCET analysis tool. *Overrules* may be used to experiment with the timing behavior of the system.
Another use of *overrules* is the ability to analyze the WCET for a selected *application mode*. *Application modes* describe subsets of the program behavior and are typically used to analyze only selected execution patterns of the concrete application. As an example for an *execution mode*, given a communication protocol stack that

manages connections between communication partners, an execution mode of interest might be solely the communication without the overhead of adding or removing communication partners.
Referring to the code example given for the *invariant* above, an example of an overrule would be the specification of a loop bound of 3, since the program semantics implies that the loop can only iterate 10 times.

The criterion of whether a timing information is an *overrule* is not only that it restricts the semantics of the program code. This is because, as shown in Figure 2.b, the system can be annotated at different layers (layers are described by timing information **T1.2**). For example, if a timing information describes properties of the execution platform, we have to look at the concrete execution platform to decide whether the timing information is an *overrule* or not.

T1.2 Specification of annotation layer

The annotation layer is an optional attribute of timing information in general. As described in Section 2.3, the WCET of a program depends on its execution platform. The execution platform is typically divided into several layers, allowing the customization of the system at each layer.

As shown in Figure 2 we propose to support the specification of at least the following three layers:

Program Layer (default): If not specified explicitly, a timing information is by default assumed to belong to the program layer, i.e., the timing information is by default assumed to be platform-independent. Here it is important to note that in programming languages like C or C++ the functional behavior is not fully platform-independent, i.e., some timing information about the control flow may already belong to the *platform layer*.

Platform Layer: The platform of a program includes everything necessary to execute the program. If a finer granularity is needed, the platform may be divided into different layers, like, for example, the build and run environment, the operating system, any middleware, and also the hardware (as shown in Figure 2.a).
For example, the cache geometry and the cache miss penalty may be specified at the hardware layer. As another example, knowing the attached flash memory device, one may specify the time needed by busy-waiting for the completion of a write access.
Figure 2 also shows the difference between platform (interface) and layers. In Figure 2.a we see the different annotation layers, including the platform layers, each of them clearly separated from the others. In contrast, a platform is an interface that subsumes all the layers below it. Thus, as shown in Figure 2.b, the influenced system behavior of each interface contains all layers below it.

Application Layer: The application layer describes the usage of the computer system, i.e., how the environment of the system is configured and how this environment behaves.
For example, timing information may describe at the application layer that the computer system is connected to three sensors, implying that a loop in the software to poll these sensors will iterate exactly three times.

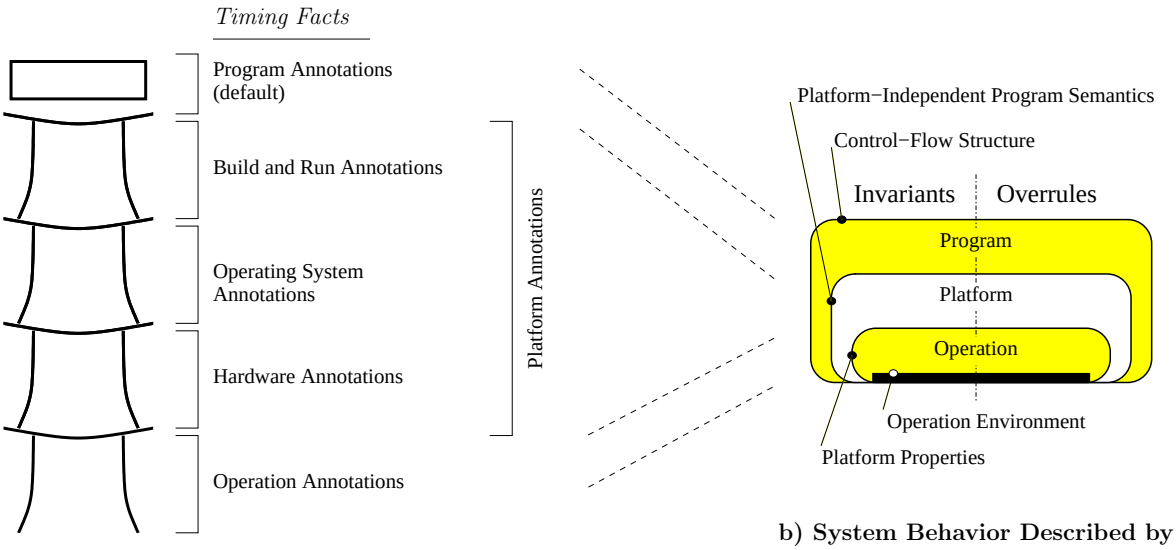
In case that a timing information describes properties of different annotation layers, the annotation layer of a timing information is the layer equal to the lowest layer among each of its properties.

The specification of the annotation layer is also important to decide the annotation class of a timing information. For example, a concrete control-flow information may be an *invariant* at the application layer, but an *overrule* at the program layer. Note that *application-modes* belong to annotation class attributes (as a form of *overrule*) and not to the annotation layer (see timing information **T1.1**).

T1.3 Specification of Annotation Group

Timing informations that are *invariants* at the program layer are relatively easy to maintain. They can be checked directly against the source code and they only have to be changed if the program code changes. They remain valid if the execution platform changes.

For timing information that refers to annotation layers different to the program or timing information that represents overrules, more care has to be taken to ensure their intended use.



a) Layered System View

b) System Behavior Described by Telescoping Timing Facts

Figure 2. Layered Timing Information

The grouping mechanism allows to give each timing information membership to multiple groups. A group simply is a symbolic name together with a description field. There is no special semantics behind the groups: their intended meaning has to be described in its description field. With the group mechanism one can specify which timing informations will be used together for WCET analysis. Hierarchical definitions of groups is supported by specification of an optional list of nested groups.

There are several reasons why one might use different sets of timing information. For example, one might want to use and annotate different scenarios at the application layer, or different tool chains at the platform layer, etc. Also *overrules* might be organized in groups to ensure their selective and intended use.

T2 Program-specific High-level Annotations

We define high-level annotations as informations that directly describe the control flow of a program. The term "high-level" hereby refers to the program layer being at the highest position in the annotation layer stack.

T2.1 Loop bounds

Loop bounds comprise the minimal information that is necessary to estimate the WCET of a simple program. For this reason, they were the first type of annotation to be introduced in the short history of WCET annotation languages [13].

Although loop bounds can always be expressed through linear flow constraints, there are practical reasons to allow loop bounds to be specified in a specialized and more compact notation. To maintain a tight execution count estimate after certain loop optimizations, it is desirable to specify lower loop bounds as well.

```
1 // This loop will be executed n times
2 // when the enclosing scope is entered
3 int i;
4 for (i = 0; i < n; ++i) {
5   // Basic block bb
6 }
Annot.: if {i,n} ∈ transp_bb then loopbound_bb = n
```

T2.2 Recursion bounds

As soon as the monotonicity of a recursion variable is established, the recursion is bounded and a maximum recursion depth can be established from the start and end values. Stack space requirements are then bounded using the recursion depth. If such conditions cannot be established by analysis, user annotations can supply the required data. In analogy to the earlier work on loop-bounds [2], Blieberger and Lieger establish the conditions necessary for establishing upper bounds for stack space and time requirements of directly recursive functions [4]. They also generalize the approach to indirect recursive functions [3]. Recursion depth annotations are also used by Ferdinand et al. [6].

```
1 // The recursion depth of fac()
```

```
2 // is depending on n
3 unsigned fac(unsigned n) {
4   if (n == 0) return 1;
5   else return n*fac(n-1);
6 }
```
Annot.: $recursionbound_{fac} = n$

T2.3 Linear flow constraints

Linear flow constraints are the basis for state-of-the-art WCET calculation methods. In the course of the calculation, all other annotations will eventually get translated into linear flow constraints. While flow constraints have a very high expressiveness, they are not necessarily as easy to write down as e. g. loop bounds, which is one of the reasons to allow multiple ways to annotate the same flow information.

Linear flow constraints are used to express a relationship between certain reference points in the CFG of a program. From the perspective of the source language this necessitates the introduction of auxiliary annotations like *markers* (to obtain a reference point) and *scopes* (to restrict the lexical validity of a constraint). The constraints themselves are usually called *restrictions*.

```
1 // This is an example of how to express Linear
2 // Flow Constraints with scopes and markers
3 // Scope({m1})
4 for (i = 0; i < n; ++i) {
5   for (j = i; j ≥ 0; --j) {
6     // Marker m1
7   }
8 }
```
Annot.: $restriction_1 \;=\; m_1 \leq n * (n-1)/2$

T2.4 Variable value restrictions

This is not a direct control-flow restriction; it has to be transformed into an explicit control-flow restriction by a program analysis tool.

```
1 if (i < 72) {
2   // In this block i is confined to be < 72
3   ...
```
Annot.: $i \;\in\;]-\infty, 72[$

T2.5 Summaries of External Functions

Often, software libraries are distributed as binaries and without any source code. In these cases, the library manufacturer could provide summaries of the library functions that contain the missing information that is necessary to analyze programs that contain calls to the library. A summary of a function may contain side effects (list of modified items) or value ranges of the returned values. A summary function would become superfluous when the source code is available.

```
1 // This function is pure and returns ±1
2 int signum(int x);
```
Annot.: $modifies_{signum} = \emptyset$
$returns_{signum} = \{-1, 1\}$

T3 Addressable Units

Addressable units in an annotation language are those that can be associated with timing information. The more language constructs and levels of abstraction can be addressed, the more fine grained the timing information can be specified. In this section we list all language construct that we consider relevant for being annotated with timing information.

T3.1 Control-Flow Addressable Units

typically express relationships between nodes, edges and paths of the control flow graph (CFG). If the paths between functions are included in the graph as well then we call this graph an interprocedural control flow graph (ICFG). Although the ICFG is implicitly defined by the program structure, it is never visible and will be generated ad hoc in the compiler. The annotation language therefore faces the problem to address entities inside a graph that has no standardized explicit representation.

We thus propose the following addressable units of the ICFG based on the program source code:

T3.1a Basic blocks as addressable units

Basic blocks are one-to-one equivalent to locations in the program code with single entry and single exit points. For timing analysis it is relevant that execution passes the entry points as often as the exit points.

T3.1b Edges as addressable units

Edges in the CFG, however, do not necessarily have a direct counterpart in the programming language's syntax because they are implicitly defined by the semantics of the respective language construct.

To circumvent this problem we introduce a set of reserved edge-names for each control flow construct of the source language. For example, considering some constructs of the C language, as shown in Figure 3, these are: $TrueEdge_{if}$, $FalseEdge_{if}$},

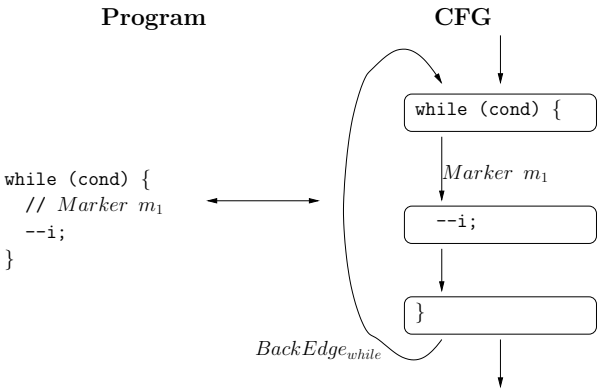


Figure 3. Addressable units in the CFG

$BackEdge_{while}$, etc. Such names allow a user to associate timing information with specific edges of the respective CFG for a given language construct.

T3.1c Subgraphs as Addressable Units

Subgraphs of the control flow graph, the call graph, or a combination of both, the interprocedural control flow graph (ICFG), [15, 1] can be addressed and thus annotated. For example, an annotation can be associated with an entire function, or with a statement containing several function calls, or some nested loops.

To handle control flow inside of expressions, such as function calls and short-circuit evaluation, it is necessary to normalize the program first. In this step short-circuit evaluation will be lowered into nested if-statements and function calls are extracted from the expressions. For addressing subexpressions a mapping between the normalized code and the original code must be established.

T3.2 Loop Contexts as Addressable Units

For all kinds of loops, it may be of interest to annotate specific iterations separately, or to exclude specific iterations, i.e. annotate all but these specific annotations. The most prominent example is that the first (few) iteration(s) may be very different from the following ones due to cache effects.

```
1 for (int i = 0; i < n; ++i) {
2   // Due to the warming-up of the cache,
3   // the first iteration will show a
4   // different behavior than the
5   // subsequent iterations
6   for (int j = 0; j < d; ++j) {
7     a[i][j] *= v[j];
```

T3.3 Call Contexts as Addressable Units

As different call sites are bound to present different preconditions for a function e.g.: input values, separate annotation of these different call contexts must be possible.

```
1 // If f() is called by g(),
2 // the loop will iterate 50 times
3 void g() {
4   f(50);
5 }
6
7 int f(int i) {
8   while (--i ≥ 0) {
9     ...
10  }
```

Annot.: $if\ callee_f = g\ then\ loopbound_{while} = 50$

T3.4 Values of Input Variables as Addressable Context

If a function behaves significantly different depending on the values of input parameters, it can be useful to provide different sets of annotations for each case. This kind of annotation was first introduced with SPARK Ada [16] under the name "modes".

```
1 // We want to use a different set of
2 // annotations depending on the value of x
3 int f(int x) {
4   ...
```

Annot.: $if\ i{=}0\ then\ Annotations_{early\_exit}$
$if\ i{\geq}1\ then\ Annotations_{normal}$

T3.5 Explicit Enumeration of (In)feasible Paths

In path-based approaches [5, 9, 16, 18], explicit knowledge of the feasibility of paths could be incorporated into the analysis process.

```
1 // init() is never called through worker()
2 void init();
3
4 void worker() {
5   while (cond) {
6     process();
7   }
8 }
9
10 void process() {
11 if (!initialized)
12   init();
13 ...
14 }
```

Annot.: $path_{worker\$process\$init} = 0$

T3.6 The Goto Statement

The goto statement is the most general way to introduce arbitrary new edges into the control flow graph. Per definition, an (unconditional)

goto statement is always the last statement of a basic block. Thus, it is not necessary to introduce any special annotations to specifically address a goto statement in the CFG; the containing basic block can be used equivalently. If the target address of a goto is not statically known, it makes sense to annotate possible jump targets as described in paragraph **T4.3**.

The break, continue and return statements are specialized instances of the goto statement.

T4 Control Flow Information

The CFG is a valuable abstraction level, that can be refined in various ways to improve the precision of the analysis. This is to aid the automatic CFG generation within the tools by additional information that is not available within the program itself.

T4.1 Specification of Unreachable Code

This is a high-level annotation, which has been used by Heckmann [10]. Unreachable code could be also specified by linear flow constraints, but having a specific mechanism for this makes the intention of the user more explicit.

T4.2 Specification of Predicate Evaluation

Closely related to the above case, this was also introduced by Heckmann [10]. This kind of annotation describes for a condition/decision whether it will always evaluate to True or False.

T4.3 Control-Flow Reconstruction

Introduced by Ferdinand [7], and further elaborated by Kirner and Puschner [14], the CFG Reconstruction Annotations are used as guidelines for the analysis tool to construct the control flow graph (CFG) of a program. Without these annotations it may not be possible to construct the CFG from the binary or object code of a program.

On one side, annotations are used for the construction of syntactical hierarchies within the CFG, i.e. to identify certain control-flow structures like loops or function calls. For example, a compiler might emit ordinary branch instructions instead of specific instructions for function calls or returns. In such cases it might be required to annotate a branch instruction whether it is a call or return instruction. A work-around, that sometimes helps avoiding code annotations is to match code patterns generated by a specific version of a compiler. However, such a "hack" cannot cover all situations and may also have the risk of incorrect classifications, for example, if a different version of the compiler is used.

On the other side, annotations may be needed for the construction of the CFG itself. This may be the case for branch instructions where the address of the branch target is calculated dynamically. Of course, static program analysis may identify a precise set of potential branch targets for those cases where the branch target is calculated locally. In contrast, if the static program analysis completely fails to bind the branch target, it has to be assumed that the branch potentially precedes each instruction in the code, which obviously is too pessimistic to be able to obtain a useful WCET bound. In such a case, code annotations are required that describe the possible set of branch targets.

The following list summarizes examples of code annotations derived from aiT [7, 10]:

- `instruction <addr>`
  `calls <target-list>;`
- `instruction <addr>`
  `branches to <target-list>;`
- `instruction <addr>`
  `is a return;`
- `snippet <addr>`
  `is never executed;`
- `instruction <addr>`
  `is entered with <state>;`

Note that these annotations need not be linked to a specific instruction type, since an optimizing compiler may transform

```
1 call F
2 jump L
```

into:

```
1 push L ; prepare return to different address
2 jump F ; jump to function, return to target
```

This is also known as triangle call or triangle jumps. Now the jump instruction represents the logical call followed by the jump and must bear both annotations.

Relevant program features: function pointers and indirect conditional control-flow transfer.

```
1 // func may only point to reset() or iterate()
2 void process((void)(int*) func, int *data) {
3   (*func)(data);
4 }
```

Annot.: $target_{func}$=

```
{(void)reset(int*), (void)iterate(int*)}
```

T5 Hardware-specific Low-level Annotations

For a realistic modelling of the execution behavior of a program, an annotation language also needs mechanisms to describe the behavior of the underlying hardware. Many of these annotations are supported by industrial timing analyzers like aiT[10].

Since hardware-specific annotations are closely tied to a specific platform, they can easily be reused for multiple programs running on the same embedded platform. It thus can make sense to extract low-level information from the program code and gather it in a common location that can be referenced by the annotations of more than one program.

It is not always obvious where to draw the borderline between low-level annotations and information that is better managed by the analysis tool. Information like the timing of CPU instructions would fall into the latter category, for example. The following items are examples of informations that are reasonable to be expressed as annotations:

T5.1 Specification of the Clock Rate

Whenever an *absolute time bound* is given in time units (and not in clock cycles), it is necessary to specify clock rate to calculate the WCET in absolute time.

T5.2 Specification of the Memory and Memory Accesses

The temporal behavior of memory accesses depends on the characteristics of the memory. Embedded systems typically use different types of memory depending on the access frequency and pattern. It is thus necessary to specify the following characteristics:

- address range of read operations
- address range of write operations
- writeable memory area (e.g. RAM, Flash-ROM) and read-only memory area (ROM)
- data and code regions
- access time of specific memory regions (in cycles or ms)

If the memory is accessed through a cache, the analyzer also needs to model the behavior of the cache. However, as said above, the parameters of such a hardware model are beyond the scope of the annotation language and are better directly specified to the timing analyzer.

T5.3 Absolute Time Bounds

Using such a construct, one could specify the maximum and minimum execution time of a fraction of code. Such a feature can be found in WCETC [12], for example.

```
1 // Wait for a I/O device to be ready;
2 // the device always responds within 30 − 100µs
3 char poll() {
4   volatile char io_port;
5   while (io_port ≠ 0)
6     /* wait */ ;
7 }
Annot.: executiontime_poll ∈ [30µs, 100µs]
```

The above features are put in perspective in Figure 4. For example, loop and recursion bounds are an alternative way to specify linear flow constraints, which are the underlying generalized high-level representation. Still, the use of more specialized annotations has priority over generic ones as it allows for meta-information like grouping. For example, the developer should use *loop bounds* instead of the use of linear flow constraints to describe the upper bound of loop iterations. The idea is to ensure that the WCET annotation language can be used almost independently of the calculation method of the WCET analysis tool by using the highest possible abstraction.

Low-level architectural specifications like memory maps and access times are a much more general way of specification, than annotating each load or store with the respective execution times. CFG reconstruction annotations, on the other hand, are a prerequisite for low-level binary code to re-gain the abstraction level of the CFG that can be used for high level annotations.

4 Conclusion

The lack of common interfaces or open analysis frameworks is an impediment for the research in WCET analysis. Ambitions have been started within the ARTIST2 network of excellence to define such a common WCET analysis platform. As part of this, *The Annotation Language Challenge* for WCET analysis has been proposed [13]. This paper is aimed to be a first step towards a common WCET annotation language. It describes essential ingredients such an annotation language should include. The timing information is described conceptionally, to allow instantiation for different representation levels and tools.

As a first step we analyzed literature to collect existing timing annotation constructs and described them in a conceptional way. As a second step, we identified potential need for further mechanisms and developed some new ingredients for annotation languages.

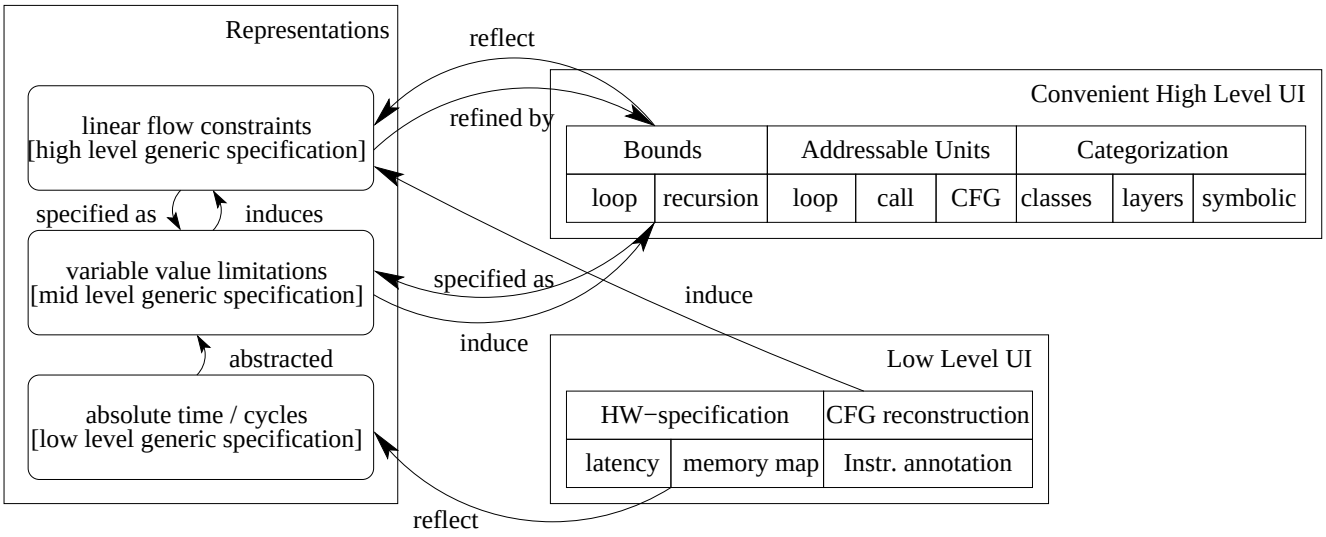


Figure 4. Relation between types of annotations and the various representations.

Among the new contributions are the separation between *invariant* and *overrule*, the introduction of *annotation layers*, grouping mechanisms, and a discussion of addressable units for annotation within the program.

The proposed list of ingredients for a WCET annotation language is by no means complete. Feedback from practitioners and researchers is encouraged to refine this list.

We would like to thank Peter Puschner for his valuable comments on earlier versions of this paper.

References

[1] M. Alt, F. Martin, and R. Wilhelm. Generating analyzers with PAG. Technical Report A10/95, Universität des Saarlandes, Germany, Dec. 1995.

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

[3] J. Blieberger. Real-time properties of indirect recursive procedures. *Inf. Comput.*, 171(2):156–182, 2001.

[4] J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 11(2):115–144, 1996.

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

[6] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. of the 1st International Workshop on Embedded Software (EMSOFT 2001)*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

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

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

[9] C. A. Healy and D. B. Whally. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions of Software Engineering*, pages 763–781, Aug. 2002.

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

[11] IST-004527. The ARTIST2 Network of Excellence on Embedded Systems Design. `http://www.artist-embedded.org/`, September 1st 2004 - August 31st 2008. funded by the European Commission within FP7.

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

[13] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET analysis: The annotation language challenge. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.

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

[15] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

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

[17] A. Vrchoticky. Modula/R - Language Definition. Technical report, Technische Universität Wien, Department of Realtime Systems, Vienna, Austria, Mar. 1992.

[18] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.