# FFX: A Portable WCET Annotation Language

Armelle Bonenfant, Hugues Cassé, Marianne de Michiel
IRIT - Université de Toulouse
118 route de Narbonne, F-31062 TOULOUSE
[michiel,casse,bonenfant]@irit.fr

Jens Knoop*, Laura Kovács*†, Jakob Zwirchmayr*‡
E185.1 - Vienna University of Technology
Argentinierstraße 8, A-1040 VIENNA
[knoop,lkovacs,jakob]@complang.tuwien.ac.at

## ABSTRACT

In order to ensure safety of critical real-time systems it is crucial to verify their temporal properties. Such a property is the Worst-Case Execution Time (WCET), which is obtained by architecture-dependent timing analysis and architecture-independent flow fact analysis. In this article we present a WCET annotation language which is able to express such information originating from the user or the analysis. The open format, named FFX to stand for Flow Facts in XML, is portable, expandable and easy to write, understand and process.

We argue that FFX allows to reuse and exchange the annotation files among WCET tools. FFX therefore permits to tighten WCET results and decreases the effort to support new architectures. Additionally, FFX flow fact files allow fair comparisons of *both* flow facts and WCET results. FFX can be used for quality assurance when developing new analysis techniques, using it as a flow fact database to test against. We present a small case study exemplifying the above points. Our case study puts special focus on the aspect of comparability and information exchange among WCET tools. In our experiments with FFX, we use the WCET analysis tool chains Otawa/oRange and r-TuBound/CalcWCET167.

## Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time systems and embedded systems

*This research is partly supported by the FP7-ICT Project 288008 Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST), the FWF National Research Network RiSE (S11410-N23) and the WWTF PROSEED grant (ICT C-050).

†L. Kovács is supported by an FWF Hertha Firnberg Research grant (T425-N23).

‡J. Zwirchmayr is supported by the CeTAT project of TU Vienna.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*RTNS'12* , November 08 - 09 2012, Pont a Mousson, France
Copyright 2012 ACM 978-1-4503-1409-1/12/11 ...$15.00.

## 1. INTRODUCTION

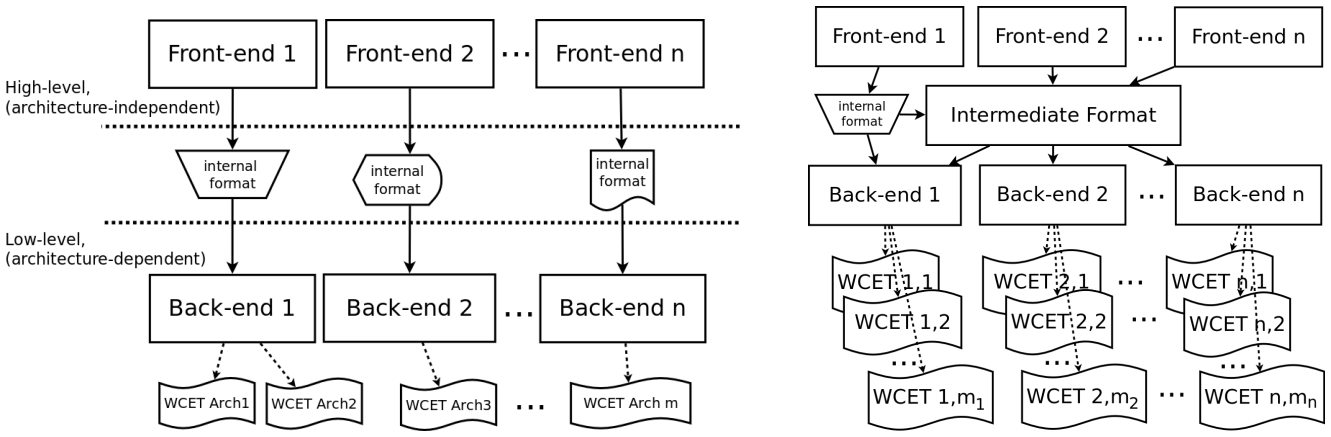
Critical hard real-time systems are composed of tasks which must imperatively finish before their deadline. A task scheduling analysis, requiring a priori WCET analysis of each task, is performed in order to guarantee safety of the system.

Static WCET analysis is performed using timing analysis tools which need flow fact information about the program under analysis. Such information may be given manually by the developer or inferred automatically by a flow fact analyzer. Well-known WCET tools, such as aiT [1], Bound-T [22] or SWEET [8], use so-called annotation languages in order to carry gathered information. In [12] the authors define and summarise ingredients of an annotation language in order to classify the information needed by the WCET analysis performed by various WCET tools. Following this line, the recent work of [13] encourages the WCET community to converge towards the use of a common annotation language for comparing various WCET tools.

In this article we address this suggestion and introduce a portable annotation language, called FFX (Flow Facts in XML). Fig. 1 illustrates the difference between the traditional workflow of WCET analyzers (Fig. 1(a)), relying on an internal format to exchange information between front-end and back-end, and the workflow of WCET analyzers using a portable annotation format, as proposed in this article (Fig. 1(b)). By a WCET analysis tool chain of a WCET analyzer we mean the full collection of tools used to analyze a program. Usually, a WCET analysis tool chain consists of architecture-independent high-level analyzers gathering flow facts about the program, e.g. oRange [3] or r-TuBound [15]. Such tools we name the WCET front-end. The parts of a WCET analysis tool chain that operate on the architecture-dependent lower level, we denote as WCET back-end, e.g. Otawa [3] and CalcWCET167 [11].

The workflow presented in Fig. 1(b) emphasizes the contributions of this article. FFX portability allows one to use and reuse results from one tool within another tool. As back-ends might support multiple architectures, such an intermediate format allows one to extend the usage of various WCET tools to different platforms and architectures. The intermediate format in our case is FFX. A presentation of the FFX format is already reported in [4]. In this article we extend [4] by the following aspects.

(i) We propose an open, portable and expandable annotation format, called FFX, as an intermediate format for WCET analysis. Such an annotation format is needed for a fair comparison of various WCET results. For instance, during the WCET tool challenge [23] almost all participants




**(a)** **(b)**

**Figure 1: (a) shows the traditional workflow of WCET analyzers using different internal formats. (b) shows the abstract setup of the FFX experiment for an arbitrary number of WCET analysis tool chains. It depicts tool chain 1 translating its internal format to the intermediate format, tool chain 2 supporting it natively and tool chain n dropping its internal format in favor of the intermediate format.**

had trouble annotating the benchmarks with the supplied flow constraints: the restrictions, often formulated in natural language, can be hard to understand correctly and even harder to annotate in a given flow fact language. We believe that with such an annotation language at hand, it is easier to tackle the WCET tool challenge and possibly open the opportunity for other tools to participate.

(ii) The FFX format allows combining flow fact information from different high-level tools. WCET analysis is usually two-fold: the timing analysis part is architecture-dependent, whereas the flow fact analysis is not. For this reason one cannot compare results from tools that do not support a common architecture, e.g. ARM or PowerPC, in the WCET tool challenge. With the FFX format at hand, we believe tools have the opportunity to extend their flow fact analysis to other architectures that are not supported by their original tool chain.

(iii) FFX decreases the implementation effort when integrating different WCET analysis tool chains. One could argue that integration of two WCET analysis tool chains by supporting the native format of the other tool chain has the same implementation effort. Nevertheless, when using the intermediate format FFX, the effort decreases with a higher number of tool chains involved. Consider Fig. 1. Adding another tool, that is Tool n+1, to Fig. 1(a) requires to implement $n$ translations to the native formats of the other tools. Using Tool n+1 in Fig. 1(b), with FFX as intermediate format, reduces implementation effort to *exactly* 2 translations. Moreover, an intermediate format comes with the advantage that a change in the internal format of a specific tool does not require all the other tools to update their translation. Because of this decoupling, it is only necessary to update the translation from FFX to the modified internal format for one tool [6].

(iv) Introducing a common annotation language is important in order to tighten and compare WCET results. FFX allows to perform the timing analysis by using an annotation file from arbitrary source, for example provided by users or inferred by a flow fact analyzer. To improve the WCET accuracy, a tool could use annotations provided by another flow fact analyzer. It is possible to merge the results of several flow fact analyzers in order to obtain the most accurate information available. This approach is already used partially in the Otawa tool in order to collect and merge analysis information from several sub-analyzers.

(v) FFX can be used for quality assurance to test and validate new analysis techniques and tools against. This is achieved by using FFX as a flow fact storage or knowledge base about benchmarks. Therefore, we believe that having the same annotation language, i.e. FFX, will also help the real-time systems developers. They will be able to adapt their choice of WCET tool to their targeted architecture or type of programs without learning a new format or tool.

(vi) We introduce an FFX support for the r-TuBound tool and perform experiments with FFX using r-TuBound and oRange. To this end, we instantiate Fig. 1(b) with the r-TuBound tool [15] and the oRange tool [16] as high-level tools, and use FFX as intermediate format. Otawa [3] and CalcWCET167 [11] serve as WCET back-ends for the tool-chains. We thus get two sets of comparable FFX flow facts and two WCET estimates for each of the architectures supported by the two tool chains. Additionally, it is possible to combine the flow facts gathered by the tools to get a more accurate WCET estimate. We report on our experiments and conclude both practical and theoretical benefits of using a common format that allows to exchange flow facts and back-ends for the WCET analysis of systems.

Let us note that the approach pursued with the FFX format is especially suited for WCET analysis tools that perform the high-level WCET analysis on source level. This is so because WCET tools usually define ways to safely map source constructs to locations in the binary representation (for example, by using debugger information, by preventing certain compile time optimizations, or by safely co-

transforming the annotations depending on the optimizations applied). Naturally, flow facts extracted on binary level can also be stored in FFX format, nevertheless, caution is required when comparing these with flow facts constructed from source level, as the mapping of source level constructs to the binary representation can be tricky.

This article is structured as follows. After focusing on related work in Section 2, we present in Section 3 the ingredients of the FFX annotation language, followed by a short description of the Otawa/oRange tool in Section 4.1. Otawa/oRange use FFX as their native annotation language. In Section 4.2 we give an overview about r-TuBound/CalcWCET167 and describe initial FFX support. We then investigate benefits of using FFX on our case study examples and present our results in Section 5. In Section 6, we conclude and discuss future extensions of the approach.

## 2. RELATED WORK

The majority of existing WCET tools come with their own annotation language to carry flow facts. For example, the SWEET tool [8] uses a flow information format called *"context-sensitive valid-at-entry-of flow facts"*, the aiT tool uses the *"aiS"* format [1], and the Bound-T tool uses the *"Bound-T Assertion Language"* [22].

SWEET offers *"value annotations"* to specify constraints on possible input values of a program, and allows to annotate flow facts in both local and global contexts. Additionally, flow information can be marked to hold only in certain function call-string contexts [8]. On the other hand, aiT understands both source level annotations as well as binary annotations, using the aiS format. It uses program points to identify source locations as addresses, routine and file names, and it combines so-called atoms to construct more complex flow facts. Bound-T uses source-code mark positions and carries additional assertions in a text file. Assertions are statements about the program that bound certain aspects of the behaviour, e.g. loop bounds or information about the stack usage. Bound-T assertions are then valid flow information at certain program points, identified by the markers.

Basic annotations, e.g. loop bounds, are supported by all aforementioned annotation languages. Nevertheless, the formats of deployed annotations are quite different. In what follows, we argue that our intermediate flow fact format FFX allows to specify important flow information in most WCET analyzers. We therefore believe, that even if some flow facts from different tools cannot yet be formulated in FFX, our work already supports many properties of state-of-the-art annotation languages. Thus, FFX can be used to make fair comparisons in order to gain comparability between the tools. Further, it allows to exchange and interchange information and back-ends between tools, resulting in better WCET estimates when using the tools in collaboration.

The approach of [17] argues that source based annotations are portable, easy to use and flexible to integrate in existing tool chains. FFX follows this line of argument by focusing especially on the flexible integration within existing tool chains.

The strength of FFX has already been shown to some extent in [3], as it is the internal format of the oRange/Otawa tool. oRange/Otawa benefits from the fact that all components of the tool chain use a common format to carry analysis information. Additionally, in the Merasa[1] project the FFX format was successfully used in order to compare results of Otawa [3] with RapiTime[2]. FFX thus already proved to be a suitable annotation language as the internal format of the Otawa/Orange WCET tool chain [3]. Adding FFX support for r-TuBound/CalcWCET167, as presented in this article, shows that FFX is also suitable for intermediate flow fact representation.

The FFX format described in this article extends [3] and supports most of the ingredients proposed in [12]. Unlike [12], where only the theoretical benefits of a common annotation language are discussed, in this article we propose FFX as a common annotation language. We address both theoretical and practical details of FFX as a common annotation language and present initial experiments with FFX. As the format is expandable, we believe that all the required ingredients of an annotation language can be supported without breaking compatibility with tools not supporting new information.

## 3. THE FFX ANNOTATION LANGUAGE

The FFX format has been introduced for and with the Otawa WCET analysis tool chain [5]. FFX is an XML-based file format that is used to represent flow facts. Such information is either used to help to achieve the computation, or to exhibit WCET in particular situations. The main concepts and ideas that drive the design and the development of FFX include – *freedom*: tools using FFX do not need to support all features. In the worst-case, unsupported features induce a loss of precision but never invalidate information; *expandability*: the use of XML allows to easily insert new elements or new attributes without breaking the compatibility with other tools; *soundness*: all provided information must be ruled by a precision order, ensuring that, at least, information in most generic cases must not be more precise than in particular cases.

One of the goals of FFX is to have an annotation language that can be extended by constructs that yield improvements in the WCET computation, e.g. flow facts (loop bounds, exclusive paths, indirect function calls, control flow constraints, etc.), platform configuration (I/O, caches, etc.), target processor, tasks or entry points, configurations of analyzers, data domain information and more.

Another motivation for FFX is to have the capability to merge the result files of different tools, allowing the tools to interchange information, e.g. between the cache analyzer and the flow fact analyzer. The choice to carry all that information in a single file is made in order to take into account the enormous number of paths and context sensitive information in a program. Carrying all this information directly in the source would clutter the source under analysis. Obviously this comes with a drawback, storing the information external to the source file could lead to divergence between code and information. This however, is a matter of bookkeeping, as one can, e.g. use version strings or source hashes to avoid divergence. FFX offers means to store that information, for example, in constructs comparable to the already mentioned platform configuration part. Additionally it eases communication between tools that apply the analysis results on different representations, be it on source

[1] `http://www.merasa.org`

[2] `http://www.rapitasystems.com/`

or binary level.

The XML nature of the format allows to collect information provided by different tools[3]. XML is standardized, easy to read and write for the developer as XML is a textual format and it offers great tool and library support. Its tree structure is particularly interesting when representing control flow graphs, as it can implicitly represent a call graph.

## 3.1 FFX Elements

In this subsection we present the most important elements of FFX, fully described in [5]. The notation uses EBNF rule format. We focus here on flow fact elements[4].

At the current state of development, FFX permits to address interesting program parts and to express information to resolve complex control structures, e.g. indirect branches due to switch statements that are compiled to function tables or function pointer calls. An FFX file may include only the code parts for which flow facts have been derived.

### 3.1.1 Location Attributes

Location attributes (Fig. 2) allow to identify code either in the binary or in the source representation of the program. This is done using different sets of attributes depending on the constructs used.

```
LOCATION-ATTRS ::=
| IDENTIFICATION
| ADDRESS-LOCATION
| LABEL-LOCATION
| SOURCE-LOCATION
```

**Figure 2: FFX Location Attributes**

IDENTIFICATION: Instead of identifying concrete locations in the program, it allows to make references to parts in the code represented by the other location attributes. A typical usage is to annotate the execution count of a piece of code inside a control constraint.

ADDRESS-LOCATION: Location by an address is the simplest scheme. A target location is represented as an address stored in an attribute. The drawback is that the location may be invalidated each time the application is compiled.

LABEL-LOCATION: The label location provides more flexibility. It encodes locations either by a label, or by a label and an offset. This scheme does not support recompilation but relinking of libraries, as the code is not modified. It only requires the translation of the label address from one position to another.

SOURCE-LOCATION: Source locations support recompilation and relinking but not source modifications. It defines the location as a source file name together with a line number in the file. To be applicable, it requires either a source representation or a binary representation embedding debugging information. It is preferable to have only one statement per line, otherwise ambiguity can emerge between locations in certain circumstances. It is necessary to obtain an XML tree corresponding to the call graph, which is guarantied if there is only one statement per line.

### 3.1.2 Context Elements

A context element (Fig. 3) defines a condition on the information it contains. Information embedded in unsatisfied contexts is not considered, i.e. only information contained in valid contexts is used in further analysis.

```
CONTEXT ::=
  <context name="TEXT">
    TOP-LEVEL-ITEM*
  </context>
```

**Figure 3: FFX Context Element**

```
<context name="arm">
  <function name="f">
    <loop maxcount="10"/>
  </function>
</context>

<context name="arm">
  <context name="task_1">
    <function name="f">
      <loop maxcount="5"/>
    </function>
  </context>
</context>
```

**Figure 4: FFX Context Example**

Consider, for example, Fig. 4: if the only valid context is `arm`, the loop bound will be 10. If both context `arm` and `task_1` are valid, the bound 5 will be used for further analysis.

Note that context names may be classified and prefixed using the following constructions below: `hard:TEXT` for hardware contexts, `task:TEXT` for task contexts and `scen:TEXT` for scenario contexts.

A hardware context represents different behavior of an application, depending on the underlying hardware. For example, the number of possible iterations of a loop counting the number of one-bits in a word depends on the size of the word. It will be 32 on 32-bit machines and 64 on 64-bit machines.

Functions composing an application may be called from different tasks that make up the real-time system. Depending on the task calling the functions, some flow properties may have different values. One could just take the worst-case behavior to characterize the functions, independent of the context. Nevertheless, this would come at the price of a loss of precision and an overestimation of the WCET.

Finally, flow information may depend on a chosen scenario. For example, an execution configuration chosen by the user, or a system that exhibits state variables controlling the running mode of the application. These might bring the application to a “running state”, “failure state” or “critical state”. It may be interesting to examine the different properties of a task according to the running mode, as the scheduling decisions can also change accordingly. In addition, the properties defined in a scenario may also be used to force the behavior of the task, for example, by fixing the value of the state variables.

[3]XInclude `http://www.w3.org/TR/xinclude/`

[4]An example for the usage of cache analysis results is given in [2].

### 3.1.3 Control Flow Elements

A function element (Fig. 5) represents the static code location for the given function. It can contain statements and thus allows to identify and access dynamic locations inside the function.

```
FUNCTION ::=
  <function LOCATION-ATTRS INFORMATION-ATTRS>
    STATEMENT*
  </function>
```

**Figure 5: FFX Functions**

The LOCATION-ATTRS identify the static location of the function in the code. INFORMATION-ATTRS represent generic hooks where any flow information can be attached to (frequency, execution time, etc.).

Different statements are supported inside functions (Fig. 6) and represent the flow structure of the code. They can be composed to express dynamic locations that depend on a specific context.

```
STATEMENT ::= BLOCK | CALL | CONDITION | LOOP
```

**Figure 6: FFX Statements**

BLOCK: A block element identifies a piece of code, possibly composed of several execution paths, but with a single entry point only.

CALL: A call element identifies the call to a function. Its location represents the caller and it must contain a function element representing the callee. Multiple call elements that embed functions allow to represent call-chain locations.

CONDITION: A condition element represents a condition with several alternatives. In the C language, it applies to both `if` statements and `switch` statements.

LOOP (Fig. 7): A loop element matches a loop construct in the code. It may contain *iteration* elements in order to represent properties that are valid only during certain iterations of the loop.

```
LOOP ::=
  <loop LOCATION-ATTRS INFORMATION-ATTRS>
    STATEMENT*
  </loop>
| <loop LOCATION-ATTRS INFORMATION-ATTRS>
    <iteration number="INT">
      STATEMENT*
    </iteration>
  </loop>
```

**Figure 7: FFX Loops**

The iteration `number` $i$ can be positive, identifying the ith iteration, or negative, identifying the $i$th iteration counted from the last iteration. A Loop bound attribute is an INFORMATION-ATTRIBUTE but is limited to loop elements (Fig. 8).

The attribute `executed` is set to `false` if the element is never executed. `maxcount` and `totalcount`, attributes of `loop` elements, denote the maximum number of loop iterations for each loop entry and the maximum number of loop iterations in relation to the outer loop scope. Those

```
LOOP-ATTR ::=
| maxcount="INT|NOCOMP"?
| mincount="INT|NOCOMP"?
| totalcount="INT|NOCOMP"?
| exact="BOOL"?
| executed="BOOL"?
| expmaxcount="TEXT"?
| exptotalcount="TEXT"?
```

**Figure 8: FFX Loop Attributes**

attributes can either be integer values, `NOCOMP` (not computable) or parameterized expressions. The `loop` attribute `exact` is set to true if the `totalcount` attribute is exact, i.e., no overestimation. `expmaxcount`, respectively `exptotalcount`, is a formula computing the value of `maxcount`, respectively `totalcount`, if no concrete value can be inferred for `maxcount`, respectively `totalcount`. These attributes represent the iteration count as a formula, using a syntax similar to that of C arithmetic expressions.

The power of FFX comes with the fact that it is expandable. One could add custom tags in order to make a certain tool more efficient. The custom tags will simply be ignored by tools that do not support them. As an example, consider a WCET back-end that relies on the *Implicit Path Enumeration Technique* (IPET) that usually only handles linear constraints, whereas FFX could introduce arbitrary constraints. Nevertheless, the back-end can just ignore these non-linear constraints and find a solution for the remaining constraints. A similar situation is the `iteration` construct: it is unused, and hence not written, by both (static analyzers) oRange and r-TuBound, but emitted by the measurement based analyzer rapiTa [21], where it is much more valuable to inspect single iterations of loops.

# 4. FFX SUPPORT IN ORANGE/OTAWA AND R-TUBOUND

This section overviews features of the FFX annotation format and presents its usage in the flow fact analyzers oRange and r-TuBound and the WCET analysis tools Otawa and CalcWCET167, respectively.

## 4.1 FFX in oRange/Otawa

oRange [16] is the flow fact analyzer for Otawa. It performs a static analysis based on flow analysis and abstract interpretation of C programs. Currently oRange does not use FFX as input but produces FFX files as output. It generates flow fact information for functions, calls, conditions and loops starting from a supplied entry point.

Fig. 9 shows an excerpt of a typical FFX file generated by oRange for the analysis of the **bs.c** benchmark from the Mälardalen suite [7]. As mentioned, the structure of the FFX file represents the structure of the C program, as it is comparable to a call graph representation of the program.

Some elements defined in FFX are currently not supported by oRange, others that are supported by oRange are not supported by Otawa. Due to the composable nature of FFX, Otawa combines analysis results of different sub-tools prior to WCET computation.

## 4.2 FFX in CalcWcet167/r-TuBound

r-TuBound extends the WCET analysis tool TuBound [19, 18] by combining symbolic computation techniques with

```
<loop loopId="1" line="67" source="bs.c" exact="false" maxcount="5"
      totalcount="5"
      maxexpr="floor((log(14-0)-log(ceil(0+epsilon)))/log(1/0.5)+1)+1"
      totalexpr="floor((log(14-0)-log(ceil(0+epsilon)))/log(1/0.5)+1)+1">
  <conditional>
    <condition varcond="IF-1" line="85" source="bs.c"
               isexecuted="true" expcond="" expcondinit="(data[mid]).key==x"></condition>
      <case cond="1"  executed="true"></case>
      <case cond="0" executed="true">
        <conditional>
          <condition varcond="IF-2" line="79" source="bs.c" isexecuted="true"
                     expcond="" expcondinit="(data[mid]).key>x">
          </condition>
          <case cond="1"  executed="true"> </case>
          <case cond="0" executed="true"> </case>
        </conditional>
      </case>
        </conditional>
</loop>
```

**Figure 9: FFX flow facts output by oRange**

the timing analysis of programs. Inputs are arbitrary C/C++ programs. The r-TuBound tool chain consists of a high-level source analyzer, a WCET-aware compiler and a low-level WCET analyzer. The low-level WCET analyzer relies on flow facts inferred by the high-level source analyzer. The high-level source analyzer implements interval analysis, points-to analysis and loop bound computation using abstract interpretation, symbolic computation and a model checking extension. The flow facts derived during the high-level source analyses are further used as source code annotations in the low-level WCET analysis, in the form of `#pragma` declarations. The WCET analysis of the system is performed for the Infineon C167 microprocessor. It is realized by a WCET-aware compiler and the low-level WCET analyzer CalcWCET167 that applies the implicit path enumeration technique (IPET) using integer linear programming [20]. The WCET-aware compiler and the low-level WCET analyzer of r-TuBound are using the WCETC language [10]. WCETC is similar to the ANSI C, extending it by constructs for WCET path annotations. These annotations in WCETC allow one to specify loop bounds and to use markers to express restrictions in the runtime behavior of the program [9], e.g., relational constraints on the execution frequencies of program blocks. This is illustrated in Fig. 10. The loop bound computation step derives the loop bound of the program fragment in Fig. 10(a) to be 4. Fig. 10(b) shows the loop bound as a `pragma` annotation in the source. The annotated source code is then transformed to the WCETC program given in Fig. 10(c). This WCETC code is finally compiled and statically analysed using CalcWCET167.

As mentioned for Otawa/oRange, also r-TuBound/CalcWCET167 does not support all constructs offered by FFX. Nevertheless, this is not an obstacle, as unsupported constructs can be ignored by tools in further analysis. An example are loop attributes: FFX allows to annotate the `totalcount` of loop iterations, i.e., the maximal number of iterations of the loop during the whole execution of the program, as well as the maximum number of iterations each time the loop is entered. r-TuBound only supports the latter, but can just ignore the other attribute.
For further details on TuBound and r-TuBound we refer the reader to [19, 18, 14, 15].

## 4.3 Immediate Benefits

Implementing FFX support for r-TuBound allows us to use Otawa as WCET back-end for r-TuBound and to use CalcWCET167 as WCET back-end for oRange. This way, both tools are able to analyze code for a new architecture. Further, it is possible to compare flow facts in a common format and to compare WCET estimates on different platforms in a unified way.

In particular, for (r-)TuBound we overcome this way a major hurdle when participating in the WCET tool challenges. This is due to the WCET back-end used in the r-TuBound tool chain that does not support the ARM and the PowerPC platform (cf. [23]) used in the challenge. While updating the WCET back-end to support a new platform usually requires heavy engineering effort, the approach we followed here, i.e., introducing in r-TuBound the WCET annotation language FFX, turned out to be very light-weight.

In fact, extending r-TuBound with an FFX support required only modest effort, as r-TuBound outputs annotated source code after each analysis step. We extract from the annotated source code all necessary analysis results and store the results in FFX format to an FFX file. The tree-like nature of FFX supports this approach, allowing for the construction of the XML representation as yet another layer in the cascade of high-level analyses, executed before the low-level WCET analysis. No changes to either the high-level analyzer nor the low-level back-end are necessary. This advantage extends to other tool chains when using FFX as an intermediate format: developers of WCET tools only need to implement a translation from their internal format to FFX, without changing the internal format of their tools. Another benefit of exporting to FFX instead of directly translating to a specific tool format emerges when additional tools participate in such a tool-cooperation: Using FFX reduces the implementation effort to two transformers, one for the high level to translate flow facts to FFX and one for the low level to translate from FFX to the back-ends native format, no matter how many tools participate. Nevertheless, one gains, for each high-level tool, support for all platforms supported by any of the back-ends. Using the direct translation would require to write a transformer for each of the back-ends. Additionally, such a decoupling from the tools native formats results in robustness towards changes in native formats, as

```
for (i = 1;
     i < 100;
     i = i * 2 + 1)
  {...}
```

**(a)**

```
for (i = 1;
     i < 100;
     i = i * 2 + 1)
  {
#pragma wcet_marker location1
#pragma wcet_loopbound (4..4)
  ... }
```

**(b)**

```
for (i = 1;
     i < 100;
     i = i * 2 + 1)
  maximum 4 iterations
  {
    marker location1
    ... }
```

**(c)**

**Figure 10: C program analysed for WCET.**

only one FFX translator needs to be adapted.

Flow facts annotated as `pragma`s are translated to the following list of FFX elements: `flowfacts`, `functions`, `loops`, `calls` and `conditionals`.

Further, the following FFX attributes are required for successful WCET analysis: the `line` attribute which is used to specify the source location of constructs. r-TuBound uses `#pragma wcet_marker` that need to be translated to line numbers. The most important flow information that r-TuBound infers are loop bounds. The `maxcount` attribute of the `loop` tags is used to encode those. The `totalcount` and the `exact` attribute are currently unsupported by r-TuBound. The elements and attributes are extracted from the annotated source that r-TuBound emits after each analysis step.

The following constructs are not used in the `pragma` translations; they could, however, be used to refine the flow information and thus tighten the WCET estimate: the `executed` attribute allows to constrain the execution of paths and/or calls. It could be used to encode `#pragma wcet_constraint` of a specific form. The `numcall` attribute could be extracted using r-TuBound's static profiler but is currently not considered (it encodes the total number of calls to a function). Some WCETC constructs cannot yet be translated to FFX, for example the `#pragma wcet_constraint` construct which limits the execution count of a program block.

The example in Fig. 11 presents a snippet of code from the Mälardalen [7] benchmark `bs.c`, annotated with flow information as used in r-TuBound, and its representation in FFX format. In r-TuBound `#pragma wcet_marker` is used to identify blocks and associate analysis information with them (e.g. the `#pragma wcet_loopbound`). FFX represents locations as FFX elements with line number attributes associated with them instead of markers. The loop bound is annotated as `maxcount` attribute of the `loop` element. Most of the additional information (`source`, `condition`, `extern`) can be extracted from the source. Expressing flow facts not only in WCETC but also in FFX offers numerous advantages: most important, we can compare WCET tools on multiple levels. Further, it allows to specify flow facts in an unambiguous way for different tools and to keep analysis information persistent. It offers the possibility of merging FFX files from different tools, i.e. acquiring a tighter WCET estimate by using the most exact information available, e.g., the tightest loop bounds.

```
    ...
// main calling binary_search

int binary_search(int x) {
#pragma wcet_marker(label30)
  ...
  while (low <= up) {
#pragma wcet_marker(label23)
    mid = ((low+up)>>1);
    if (data[mid].key == x) {
#pragma wcet_marker(label18)
      up = (low-1);
      fvalue = (data[mid].value);
    } else  {
      if ((data[mid].key) > x) {
#pragma wcet_marker(label21)
        up = (mid-1);
      } else {
#pragma wcet_marker(label22)
        low = (mid+1);
      }
    }
#pragma wcet_loopbound(8..8)
  }
  return fvalue;
}
...
<call name="binary_search" numcall="1" line="54"
      source="bs.c" executed="true"
      extern="false">
  <function name="binary_search" executed="true"
            extern="false">
    <loop loopId="0" line="67" source="bs.c"
          exact="false" maxcount="8">
      <conditional>
        <condition varcond="IF-1" line="70"
                   source="bs.c" isexecuted="true"
                   expcond="data[mid].key==x;" />
          <case cond="1" executed="true" />
          <case cond="0" executed="true">
            <conditional>
              <condition varcond="IF-2" line="79"
                         source="bs.c"
                         isexecuted="true"
                         expcond="data[mid].key>x;"/>
              <case cond="1" executed="true" />
              <case cond="0" executed="true" />
            </conditional>
          </case>
      </conditional>
    </loop>
  </function>
</call>
...
```

**Figure 11: Part of the Mälardalen benchmark s.c: on top, the original annotations as output after high-level analysis by r-TuBound, on bottom, the FFX translation.**

## 5. EXPERIMENTAL COMPARISON

As pointed out, our experimental case study focuses on the aspect of comparability, information exchange and extending WCET results to previously unsupported platforms by translating the most important flow facts to FFX and

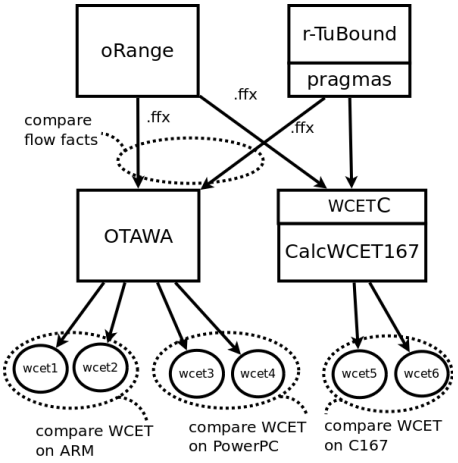



**Figure 12: Current use of FFX as intermediate format for r-TuBound/CalcWCET167 and Otawa/oRange. The experiments were performed for the ARM and C167 architecture.**

WCETC, respectively. Extensions to this work could focus on using a larger subset of FFX to also allow other tools to participate in information and WCET back-end interchange.

The translation from WCETC to FFX and vice versa allows to investigate differences in flow facts derived from the high-level (e.g. loop bounds) and to study their effect on the tightness of the WCET calculation for a specific platform.

In our case study about FFX-based WCET analysis, we use and compare r-TuBound/CalcWCET167 and oRange/-Otawa. Fig. 12 illustrates the concrete setup of the experiments: the leftmost and the rightmost arrows represent the default workflow and flow of information for each tool chain. Arrows crossing from one tool chain to the other correspond to flow fact translation, either from FFX to WCETC or from `pragma`s to FFX. There exist multiple locations where the tools can exchange information and where the output of the tools can be compared. On flow facts level, depicted in the top left of the diagram, one can compare the FFX output of oRange with the FFX output translated from r-TuBound `pragma`s. Similarly, on WCETC/`pragma` level. On the low level, at the bottom of the diagram, one can compare the WCET estimates for architectures supported by Otawa and CalcWCET167 by supplying them with translated r-TuBound and oRange flow facts, respectively.

We will present a synthetic example to illustrate the precision gain from combining flow facts from different analyzers. Additionally, we perform experiments on three WCET benchmarks taken from the Mälardalen [7] benchmark family, **bs.c**, **cnt.c** and **minver.c**. All of them are small enough to manually inspect and compare the tool outputs. We choose benchmark **bs.c** because oRange performs better flow fact analysis than r-TuBound, **cnt.c** because oRange and r-TuBound infer the same flow facts and **minver.c** because r-TuBound performs better flow fact analysis on the benchmark than oRange. We summarize our results in Table 1. The columns X+Y denote the WCET obtained using X as the flow fact analyzer (r for r-TuBound and o for oRange) and the Y back-end (C167 for CalcWCET167 and ARM for ARM Otawa back-end). Therefore, only columns with same Y back-end (columns 2 and 3 or 4 and 5) are meaningful to compare.

```
struct DATA { int key; int value; };
struct DATA data[15] = {
  {1, 100}, {5,200}, {6, 300},
  {7, 700}, {8, 900}, {9, 250},
  {10, 400}, {11, 600}, {12, 800},
  {13, 1500}, {14, 1200}, {15, 110},
  {16, 140}, {17, 133}, {18, 10}
};

void main (void) {
  int i, nondet;
  int mid, up, low, x;

  for (i = 1; i < 100; i++) {
    if (nondet)
      i = i * 2 + 1;
    else
      i = i * 2 + 2;
    low = 0;
    up = 14;

    while (low <= up) {
      mid = (low + up) >> 1;
      if (data[mid].key == x)
        up = low - 1;
      else if (data[mid].key > x)
        up = mid - 1;
      else
        low = mid + 1;
    }
  }
}
```

**Figure 13: A synthetic example where r-TuBound helps oRange to infer a tighter bound on the total number of loop iterations. The example is constructed partly from the `bs` benchmark (inner loop) and examples presented in [14] (outer loop).**

(a) The WCET resulting from oRange flow facts on the C167 platform for function `b_s` is tighter than the WCET for r-TuBound flow facts (19140 vs. 29220). The WCET when using r-TuBound flow facts is tighter for function `minver` (910640 vs. unbound). (b) The WCET estimate on ARM is tighter when using oRange flow facts for the analysis of function `b_s` (775 vs. 11890), and tighter for function `minver` when using r-TuBound flow facts (98905 vs. unbound). Thus, in this case, one can merge the flow facts to achieve a better WCET than with the original tool chain (Table 2). The difference in WCET in this case is only due to differences in the loop bounds. We are currently investigating whether and how much the WCET result will diverge for larger benchmarks with additional differences in the FFX files (e.g. differences in flow information about infeasible paths).

The piece of code above (Fig. 13) shows the gain from combining flow facts extracted by different flow fact analyzers. The example is synthetic and its only purpose to illustrate the theoretical capabilities of merging flow facts: When analyzing the example, r-TuBound can make use of its loop refinement capabilities, thus refining the loop bound of the outer loop to 6. For the inner loop, r-TuBound infers an over-approximated loop bound of 8. Using these loop bounds for further analysis, a `totalcount` (the maximal number of executions of the inner loop when running

| BM | Fct | r+C167 | o+C167 | r+ARM | o+ARM | Notes |
|---|---|---|---|---|---|---|
| **bs** | main | 920 | 920 | 1220 | **815** | |
| | b_s | 29220 | **19140** | 1180 | **775** | bound 8 vs. 5 |
| **cnt** | Init | 216020 | 216020 | 13175 | 13175 | |
| | InitS | 920 | 920 | 45 | 45 | |
| | main | 1120 | 1120 | 31620 | 31620 | |
| | RandI | 1840 | 1840 | 35 | 35 | |
| | Sum | 39700 | 39700 | 18265 | 18265 | |
| | Test | 12360 | 12360 | 31530 | 31530 | |
| | ttime | 920 | 920 | 35 | 35 | |
| **minver** | main | 167760 | 167760 | **140920** | - | |
| | minver | **910640** | - | **98905** | - | bound for `while` |
| | mmul | 474880 | 474880 | 39145 | 39145 | |
| | mfabs | 7500 | 7500 | 250 | 250 | |

**Table 1: BM denotes the benchmark, Fct lists functions in the benchmark, the next 4 columns denote the WCET result on different platforms using the given flow fact analyzer and WCET back-end. The last column points out the difference in the flow facts.**

| BM | Fct | r/o+C167 | r/o+ARM | Improvement |
|---|---|---|---|---|
| **bs** | main | 920 | 815 | reduces to 88.58% of r-TuBounds original WCET on ARM |
| | b_s | 19140 | 775 | 65.5% of r-TuBound WCET on C167, 65.67% of r-TuBound WCET on ARM |
| **minver** | main | 167760 | 140920 | unbound (no result) for oRange/Otawa |
| | minver | 910640 | 98905 | unbound (no result) for oRange/Otawa |

**Table 2: WCET analysis using merged FFX files. Functions that did not change compared to the last table are omitted. Improvements denote the improvements in the WCET estimate of the tool configuration that performs better compared to the WCET of the original tool chain.**

the program) of 48. On the other hand, oRange would calculate a loop bound of 50 for the outer loop but find a tighter loop bound of 4 for the inner loop. Thus, the `totalcount` inferred is 200. In both cases, the `totalcount` is an over-approximation of the actual `totalcount`. Merging the flow facts allows to infer a safe and tighter `totalcount`: using r-TuBounds loop bound for the outer loop together with oRanges loop bound for the inner loop, results in a `totalcount` of 24!

# 6. CONCLUSIONS AND FUTURE WORK

Based on our experience with FFX we are confident that FFX is a suitable open format to store, exchange and collect flow fact information for later use in the WCET analysis of systems.

One major advantage of FFX is, illustrated in our case study, that source level analyzers supporting the FFX format can interchange WCET back-ends. At the same time it offers a way of comparing WCET tools and it allows to refine and possibly tighten WCET results by merging flow facts from different tools. In future work, we will introduce an order on flow facts and other FFX constructs that allows to determine in which manner FFX files should automatically be merged to gain better accuracy for WCET analysis. At the same time one could introduce consistency rules for relevant flow fact information. These rules can then be used to check validity and precision of gathered flow facts.

FFX allows to specify flow facts in an unambiguous way, therefore as future work, we propose to extend FFX in a way that makes it possible to encode problems from the WCET tool challenge, as this would allow for more exact problem specifications and tool comparisons. Additionally, we plan on investigating FFX for a larger experiment with additional WCET and flow fact analyzers involved.

# 7. REFERENCES


[1] AbsInt Angewandte Informatik GmbH. aiT. `http://www.absint.com`, 2007.

[2] Clément Ballabriga, Hugues Cassé, and Marianne de Michiel. A Generic Framework for Blackbox Components in WCET Computation (regular paper). In *Workshop on Worst-Case Execution Time Analysis, Dublin, 30/06/09*, volume 252, pages 118–129, http://www.ocg.at, octobre 2009. Austrian Computer society.

[3] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, Austria, October 2010. Springer.

[4] Hugues Cassé and Marianne De Michiel. FFX: Flow Facts in XML. Rapport de recherche IRIT/RR–2012-5–FR, IRIT, Université Paul Sabatier, Toulouse, April 2012.

[5] Hugues Cassé, Marianne de Michiel, and Armelle Bonenfant. FFX (Flow Fact in XML) format. Rapport de recherche RR–2012-5–EN, IRIT, Université Paul Sabatier, Toulouse, 2012.

[6] Melvin E. Conway. Proposal for an uncol. *Commun. ACM*, 1(10):5–8, October 1958.

[7] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of 10$^{th}$ Int'l Workshop on Worst-Case Exeuction Time Analysis (WCET 2010)*, pages 136 – 146, Brussels, Belgium, July 2010. Österreichische Computer Gesellschaft.

[8] Jan Gustafsson. SWEET. `http://www.mrtc.mdh.se/projects/wcet/sweet.html`, 2001.

[9] Raimund Kirner. Integration of Static Runtime Analysis and Program Compilation. Master's thesis, Vienna University of Technology, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, 2000.

[10] Raimund Kirner. User's Manual – WCET-Analysis Framework based on WCETC. Available at `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`, 2001.

[11] Raimund Kirner. The WCET Analysis Tool CalcWcet167. In *Proc. 5$^{th}$ Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation (to appear)*. Springer Verlag, October 2012.

[12] Raimund Kirner, Albrecht Kadlec, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a Common WCET Annotation Language: Essential Ingredients. In *Proc. 8$^{th}$ Intl. Workshop on Worst-Case Execution Time Analysis (WCET 2008*, 2008.

[13] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond Loop Bounds: Comparing Annotation Languages for Worst-Case Execution Time Analysis. *Software and System Modeling*, 10(3):411 – 437, 2011.

[14] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Proc. of PSI 11*, pages 116 – 126, Novosibirsk, Akademgorodok, Russia, 2011.

[15] Jens Knoop, Laura Kovaćs, and Jakob Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In *Proc. of 18$^{th}$ Logic for Programming Artificial Intelligence andReasoning (LPAR 18)*, volume 7180 of *LNCS*, pages 435 – 444, Mérida, Venezuela, 2012.

[16] Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proc. 14$^{th}$ IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, (RTCSA 2008)*, Taiwan, 2008.

[17] Adrian Prantl. *High-level Compiler Support for Timing-Analysis*. PhD thesis, Vienna University of Technology, Argentinierstraße 8/4/185-1, 1040 Vienna, Austria, 2010.

[18] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint Solving for High-Level WCET Analysis. In *Proc. of 18$^{th}$ Logic-based Methods in Programming Environments*, pages 77 – 88, 2008.

[19] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proc. 8$^{th}$ International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.

[20] Peter Puschner and Anton Schedl. A Tool for the Computation of Worst-Case Task Execution Times. In *Proc. of 5$^{th}$ Euromicro Workshop on Real-Time Systems*, pages 224 – 229, June 1993.

[21] Rapita Systems Ltd. RapiTime Explained – White Paper. `http://www.rapitasystems.com/downloads/rapitime_explained_white_paper`.

[22] Tidorum Ltd. Bound-T. `http://www.tidorum.fi/bound-t`, 2005.

[23] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Casse, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N.M. Islam, D. Kästner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr. WCET Tool Challenge 2011: Report. In *Proc. 11$^{th}$ International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*, 2011.