r-TuBound: Loop Bounds for WCET Analysis (tool paper)

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr*

TU Vienna

Abstract. We describe the structure and the usage of a new software tool, called r-TuBound, for deriving symbolic loop iteration bounds in the worst-case execution time (WCET) analysis of programs. r-TuBound implements algorithms for pattern-based recurrence solving and program flow refinement, and it was successfully tested on a wide range of examples. The purpose of this article is to illustrate what r-TuBound can do and how it can be used to derive the WCET of programs.

1 Introduction

One of the most challenging tasks in the worst-case execution time (WCET) analysis of programs with loops comes with the task of providing precise bounds, called loop bounds, over the number of loop iterations.

In this article we describe the r-TuBound tool for deriving automatically loop bounds in the WCET analysis of programs. Several software packages for this purpose have already been developed in the past and can be classified within two categories. One line of research uses powerful symbolic computation algorithms to derive loop bounds (see e.g. [1]), but makes very little, if any, progress in integrating these loop bounds in the program analysis environment of WCET. Another line of research makes use of abstract interpretation based static analysis techniques to provide good WCET estimates; however, often loop bounds are assumed to be a priori given, in part, by the user (see e.g. [10, 4, 7]).

The philosophy of our tool is somewhat in the middle of these two research trends. Rather than a package integrating powerful symbolic computation algorithms, r-TuBound uses pattern-based recurrence solving (Section 2.4) for a restricted, yet in practice quite general class of programs. Loop bounds are inferred to be satisfiable instances of a system of arithmetic constraints over the loop iteration variable. r-TuBound can thus derive non-trivial loop bounds, but not only that. The inferred loop bounds are further used in the WCET analysis of programs. To make the loop bound computation techniques scale for the WCET analysis, r-TuBound translates loops with nested conditionals into loops without conditionals (Section 2.3) using SMT reasoning in conjunction with program flow refinement. When evaluated on a large class of benchmarks, our experiments indicate the applicability of r-TuBound in the WCET analysis of programs (Section 3).

* This research is supported by the CeTAT project of TU Vienna. The second author is supported by an FWF Hertha Firnberg Research grant (T425-N23). This research is partly supported by the FWF National Research Network RiSE (S11410-N23) and the WWTF PROSEED grant (ICT C-050).

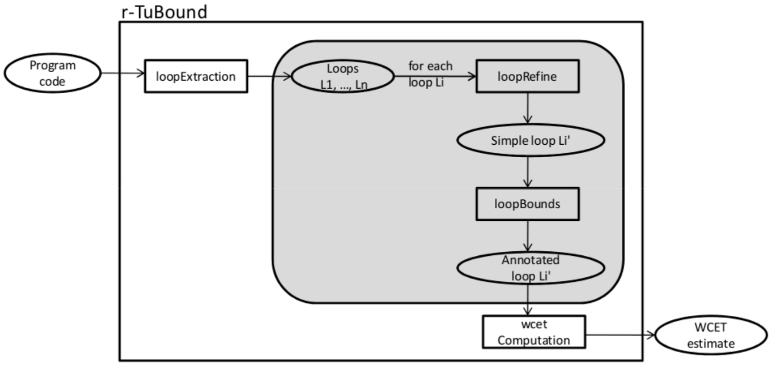

Fig. 1. The r-TuBound tool.

The goal of this article is to describe *what* r-TuBound can do, explain how to *use it*, and give *implementation details* about the structure of r-TuBound. We describe only briefly *how* r-TuBound obtains its results and refer to [6] for more details.

Implementation and Availability. r-TuBound is implemented in C++ and the Termite library [9] for Prolog. It is available at www.complang.tuwien.ac.at/jakob/tubound/. All results presented in this article were obtained on a machine with a 2.53GHZ Intel Core i5 CPU and 4GB of RAM. To improve readability, we employed minor simplifications over the input and output format of the examples discussed in the article.

2 r-TuBound: Tool Description and Usage

2.1 Workflow

The overall workflow of r-TuBound is given in Figure 1.

Inputs to r-TuBound are arbitrary C/C++ programs. In the first part of r-TuBound, the input program is parsed and analysed. As a result all loops and unstructured goto statements of the input code are extracted. To this end, various static analysis techniques from [10] are applied, such as parsing by the EDG C/C++ frontend, building the abstract syntax trees and the control-flow graph of the program, interval analysis over program variables, and points-to analysis. Further, the extracted loops and goto statements are rewritten, whenever possible, into the format given in equation (M). Doing so requires, among others, the following steps: rewriting while-loops into equivalent for-loops, rewriting if-statements into if-else statements, translating multi-path loops with abrupt termination into loops without abrupt termination, and approximating non-deterministic variable assignments. The aforementioned steps for parsing, analysing and preprocessing C/C++ programs are summarized in the `loopExtraction` part of Figure 1. If a program loop cannot be converted into equation (M) by the `loopExtraction` part of r-TuBound, r-TuBound does not compute a loop bound and hence the WCET computation step of r-TuBound will fail.

Next, the loops extracted in the format given in equation (M) are analysed and translated into equation (S), required by the loop bound computation engine of r-TuBound

```
void test() {
 int i, a[16];
 i = 0;
 while (i < 100) {
  if (a[i] > 0) i = i * 2 + 2;
  else i = i * 2 + 1;
  i = i + 1;  }}
```

Fig. 2. Input C program `test.c`

```
1 void test() {
2  int i, a[16];
3  for (i = 0; i < 100; i = i + 1)
4   if (a[i] > 0) i = i * 2 + 2;
5   else i = i * 2 + 1;
6 }
```

Fig. 3. C program `test.c` satisfying the format requirements of equation (M)

(see Section 2.3). This step is performed in the `loopRefine` part of Figure 1. As a result of `loopRefine`, the multi-path loops of equation (M) are translated into the simple loops presented in equation (S).

For deriving loop bounds in the `loopBounds` step of Figure 1, each loop is analysed separately and bounds are inferred using recurrence solving. The computed loop bounds are added as annotations to the loops and are further used to calculate the WCET of the input program, as illustrated in the `wcetComputation` engine of Figure 1. Outputs of r-TuBound are thus the WCET of the input programs. For simplicity, in the current version of r-TuBound, the execution time of a processor cycle is assumed to be of 20 nanoseconds per cycle.

Let us note that in the `wcetComputation` and `loopExtraction` steps of Figure 1, r-TuBound makes use of the static analysis framework of [10]. The distinctive features of r-TuBound, and the main contributions of this article, come with the *automatic inference of loop bounds* (steps `loopRefine` and `loopBounds` of Figure 1). To this end, r-TuBound implements pattern-based recurrence solving and program flow refinement techniques, and integrates these techniques in the WCET analysis of programs.

To invoke r-TuBound, one uses the following command.

Command 2.1 : `rTuBound program.c`

Input: C/C++ program
Output: WCET of `program.c`
Assumption: Loops of `program.c` are or can be transformed in equation (M)

EXAMPLE 2.1 Consider the `test.c` program given in Figure 2. The WCET returned by r-TuBound is listed below.

```
Input:  rTuBound test.c
Output: 26
```

The WCET of `test.c` is thus inferred to be of 26 time units. For doing so, the while-loop of Figure 2 is first translated into equation (M), as given in Figure 3. Next, in the WCET computation of Figure 3, we assume for simplicity that all program expressions take one time unit to execute. Therefore, the execution of one iteration of the loop between lines 3-5 of Figure 3 takes 4 time units: 1 unit each to check the boolean conditions of the loop and of the if-statement, 1 unit to execute the assignment statement $i = i + 1$ from the loop header, and 1 unit to execute the assignment from one branch of the if-statement, depending on the boolean test $a[i] > 0$. Further, r-TuBound

```
for (i = 0; i < 100; i = i + 1) {
  if (a[i] > 0) i = i * 2 + 2;
  else i = i * 2 + 1;
}
```

Fig. 4. Multi-path loop from `test.c` (`mpath1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
}
```

Fig. 5. Over-approximation (`simple1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
  #wcet_loopbound(6)
}
```

Fig. 6. Annotated with loop bound (`annot1.c`)

computes 6 to be the loop bound, from which it infers that the execution of all loop iterations takes altogether 24 time units. As the initialisation of the loop counter i, as well as the last test of the loop condition takes one unit each, the WCET for `test.c` is hence computed to be 26 time units.

2.2 Loop Restrictions

The syntax of program loops that can be handled by the loop bound computation part of r-TuBound is given below.

```
for (i = a; i ◇ b; i = c * i + d) {
    i = f_0(i);
    if (g_1) i = f_1(i); else i = f_2(i);
    if (g_2) i = f_3(i); else i = f_4(i);                    (M)
    ... ;
    if (g_m) i = f_{2m-1}(i); else i = f_{2m}(i);
}
```

where $\diamond \in \{<, >\}$, $g_1, \ldots, g_m$ are boolean expressions, and a, b, c, d are symbolic integer-valued constants such that a, b, c, d do not depend on i and $c > 0$. The expressions f_s, with $s = 0, \ldots, 2m$, are non-constant linear integer arithmetic functions over i; that is, $f_s(i) = c_s * i + d_s$ where c_s, d_s are symbolic integer-valued constants that do not depend on i and $c_s > 0$. Moreover, either $i < c * i + d$ and $i \leq f_s(i)$ hold for $i \geq a$, or $i > c * i + d$ and $i \geq f_s(i)$ are valid for $i \leq a$.

Let us make the following observation over the restrictions of (M). As $c > 0$ and $c_s > 0$ in equation (M), the functions $i \mapsto c * i + d$ and $f_0(i), \ldots, f_{2m}(i)$ are all monotonically increasing. Therefore, when translating arbitrary loops into the format of equation (M) in the `loopExtract` part of r-TuBound, we proceed as follows. To ensure that $i < c * i + d$ and $i \leq f_s(i)$ hold it suffices to check whether $i < c * i + d$ and $i \leq f_s(i)$ are valid for $i = a$.

In what follows, we fix some terminology used in the rest of the article. In the sequel whenever we write *loop* (M) or (M) we refer to a multi-path loop as given in equation (M). We refer to the variable i in equation (M) as the *loop counter* or the *loop iteration variable*, whereas the assignment $i = c * i + d$ in the loop header is called the *update expression* of the loop. The constant a is called the *initial value* of i. We consider a loop a *simple loop* if there is only one execution path through the body, i.e. if $m = 0$ in equation (M). Otherwise, if $m > 0$, the loop (M) is said to be a *multi-path loop*. Finally, the assignments $i = f_s(i)$, with $s = 1, \ldots, 2m$, are called the *conditional updates* of the loop.

2.3 Program Flow Refinement

Given a multi-path loop (M), the `loopRefine` part of r-TuBound translates (M) into a simple loop, such that the loop bound of the simple loop is also a loop bound of the multi-path loop (M).

To this end, the multi-path behavior of (M) is safely over-approximated, as follows.

The boolean conditions $g_1, \ldots, g_m$ are first ignored, yielding thus a loop body with non-deterministic conditional statements.

Next, for each $s = 1, \ldots, m$, we are left with choosing $k_s \in \{2s-1, 2s\}$ such that

$$f_{k_s}(i) \leq f_{2s-1}(i) \ \text{ and } \ f_{k_s}(i) \leq f_{2s}(i) \ \text{ for every } i \rhd a, \tag{1}$$

where $\rhd \in \{\leq, \geq\}$ is defined as follows:
- $\rhd$ is $\geq$ if $i < c*i+d$ in (M);
- $\rhd$ is $\leq$ if $i > c*i+d$ in (M).

The conditional update $i = f_{k_s}(i)$ determined by (1) yields thus the minimal increase, respectively the minimal decrease, over i after an arbitrary execution of the if-statement with ignored test condition g_s. Therefore, by replacing each if-statement with the corresponding $i = f_{k_s}(i)$ at every iteration of (M), a safe loop bound for (M) can be derived.

However, a $k_s \in \{2s-1, 2s\}$ might not always be computed from (1), as (1) needs to hold for *every* $i \geq a$ (respectively, for *every* $i \leq a$). That is, the existence of $k_s \in \{2s-1, 2s\}$ such that (1) is valid depends crucially on the initial value of a. To overcome this limitation, we proceed as follows. Whenever $k_s \in \{2s-1, 2s\}$ cannot be computed from (1), we take $f_{k_s}(i) = i$. Based on the restrictions of equation (M), we clearly have $f_{k_s}(i) = i \leq f_{2s-1}(i)$ and $f_{k_s}(i) = i \leq f_{2s}(i)$ for every $i \geq a$ (respectively, $f_{k_s}(i) = i \geq f_{2s-1}(i)$ and $f_{k_s}(i) = i \geq f_{2s}(i)$ for every $i \leq a$). That is, $i = f_{k_s}(i)$ yields a smaller increase (respectively, decrease) over i than any branch of the if-statement with ignored test condition g_s. Therefore, if k_s cannot be computed from (1), we define $f_{k_s}(i) = i$ and replace the if-statement with the ignored test condition g_s by $i = f_{k_s}(i)$. The loop bound for (M) is thus safely over-approximated.

Based on the above observations, equation (M) is translated into the simple loop (T), given below.

$$\begin{array}{l} \mathbf{for}\ (\mathtt{i} = \mathtt{a};\ \mathtt{i} \diamond \mathtt{b};\ \mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d})\ \{ \\ \qquad \mathtt{i} = \mathtt{f}_0(\mathtt{i}); \mathtt{i} = \mathtt{f}_{\mathtt{k}_1}(\mathtt{i}); \ldots; \mathtt{i} = \mathtt{f}_{\mathtt{k}_m}(\mathtt{i})\} \end{array} \tag{T}$$

Let us write $\phi(i) = c*i+d$, and let $\circ$ denote the standard operation of function composition. Using this notation, (T) is further rewritten into the simple loop:

$$\mathbf{for}\ (\mathtt{i} = \mathtt{a};\ \mathtt{i} \diamond \mathtt{b};\ \mathtt{i} = (\mathtt{f}_{\mathtt{k}_m} \circ \cdots \circ \mathtt{f}_{\mathtt{k}_1} \circ \mathtt{f}_0 \circ \phi)(\mathtt{i}))\ \{\} \tag{S}$$

In the sequel whenever we write *loop* (S) or (S) we refer to a simple loop as given in equation (S).

Note that as linear functions are closed under composition, $(f_{k_m} \circ \cdots \circ f_{k_1} \circ f_0 \circ \phi)(i)$ yields a non-constant linear integer arithmetic function over i in (S).

The behavior of `loopRefine` is summarised below.

```
for (i = 0; i < 100; i = i + 1) {
  if (a[i] > 0) i = i * 3 + 2;
  else i = i * 2 + 10;
}
```

Fig. 7. A multi-path loop (mpath2.c)

```
for (i = 0; i < 100;
    i = i + 1) {
}
```

Fig. 8. Over-approximation (simple2.c)

```
for (i = 0; i < 100;
    i = i + 1) {
  #wcet_loopbound(100)
}
```

Fig. 9. Annotated loop (annot2.c)

Command 2.2 : `loopRefine loop.c`

Input: Loop
Output: a simple loop as in (S)
Assumption: `loop.c` as in (M)

EXAMPLE 2.2 For translating the multi-path loop given in Figure 4, `loopRefine` infers that the conditional update corresponding to the else-branch of the conditional statement of `mpath1.c` yields the minimal update over i. The multi-path loop of Figure 4 is thus rewritten into the simple loop given in Figure 5, as listed below.

```
Input:  loopRefine mathp1.c
Output: simple1.c
```

EXAMPLE 2.3 Consider now the multi-path loop given in Figure 7. Note that $3*i+2 \leq 2*i+10$ does not hold for every $i \geq 0$. Similarly, $2*i+10 \leq 3*i+2$ does not hold for every $i \geq 0$. Hence, no conditional update can be chosen by `loopRefine` as the update yielding the minimal increase over i. Therefore, the conditional statement of Figure 7 is over-approximated by the update $i = i$, and Figure 7 is rewritten into Figure 8, as shown below.

```
Input:  loopRefine mpath2.c
Output: simple2.c
```

The task of choosing k_s in (1) such that $f_{k_s}(i)$ yields the minimal increase over i, is encoded in r-TuBound as a set of SMT queries. For doing so, we interfaced `loopRefine` with the Boolector SMT solver [2]. To this end, for each variable in the program, a bit vector variable is introduced by `loopRefine`. An array is used to model the values of the variables involved. This representation allows `loopRefine` to capture the loop behavior in a symbolic manner, by using symbolic values to model the updates to the loop counter.

2.4 Pattern-based Recurrence Solving

Loop bounds of simple loops (S) are derived by the `loopBounds` part of r-TuBound. For doing so, `loopBounds` implements a pattern-based recurrence solving algorithm, as follows.

An additional variable, denoted by n, is introduced to speak about the value $i(n)$ of the variable i at the nth iteration of the loop. Using this notation, the update expression of (S) can be modeled by a linear recurrence equation with constant coefficient.

Such recurrences can always be solved [3]. Hence, $i(n)$ is expressed as a function, i.e. the closed form, over n and the initial value of i. However, loopBounds does not implement the general algorithm for solving linear recurrences of arbitrary orders, but it makes use of the restrictions imposed over (S). Since i is modified in (S) by a non-constant linear expression over i, the resulting recurrence equation of $i(n)$ is a (homogeneous) linear recurrence of order 2. Using the generic closed form pattern of such recurrences, loopBounds derives the closed form of $i(n)$ by instantiating the symbolic constants in the generic closed form with expressions over the initial value of i.

The loop bound of (S) is then inferred by computing the smallest n such that the loop condition holds at the nth loop iteration but it is violated at the $n + 1$th iteration (see [6] for more details). That is, the loop bound is obtained as a satisfying assignment over n such that the formula below holds:

$$n \geq 0 \ \wedge \ i(n) < b \ \wedge \ i(n+1) \geq b, \tag{2}$$

where the constant b is as given in (S).

The usage of the loop bound computation part of r-TuBound is listed below.

Command 2.3 : `loopBounds simple.c`

Input: Simple loop as in (S)
Output: Loop annotated with its loop bound

EXAMPLE 2.4 For the simple1.c loop given in Figure 5 we obtain:

```
Input:  loopBounds simple1.c
Output: annot1.c
```

where annot1.c is listed in Figure 6. The annotation #wcet_loopbound(6) specifies that loopBounds computed 6 as the loop bound of simple1.c.

Similarly, the annotated loop derived by loopBounds for Figure 8 is given in Figure 9.

The pattern-based recurrence solving algorithm and the satisfiability checking of (2) are implemented in loopBounds on top of Prolog. loopBounds operates on the TERM representation offered by the Termite library [9]. Let us note that the closed form representation of $i(n)$ in equation (2) involves, in general, exponential sequences in n. Therefore, to compute the value of n such that (2) holds, loopBounds makes use of the logarithm, floor and ceiling built-in functions of Prolog.

3 r-TuBound: Experimental Results

The overall flow of r-TuBound is given in Figure 1. The program analysis framework loopExtraction, the loop refinement step loopRefine and the WCET computation part wcetComputation of r-TuBound are written in C++. The loop bound computation engine loopBounds of r-TuBound is implemented on top of the Termite library of Prolog. The loopExtraction and wcetComputation components of r-TuBound are based on the work presented in [10]. Our contribution in r-TuBound comes with extending [10] with an automatic inference of symbolic loop bounds. r-TuBound offers thus software support for the timing analysis of programs by **r**ecurrence solving and SMT based flow **r**efinement in conjunction with WCET techniques (TuBound).

The `loopRefine` part of r-TuBound comprises about 1000 lines of C++ code, whereas the `loopBounds` engine of r-TuBound contains about 350 lines of Prolog code. The `loopRefine` and `loopBounds` parts of r-TuBound are glued together using a 50 lines shellscript code.

Experimental evaluation and comparison. We evaluated r-TuBound on a number of benchmarks coming from the WCET community, as well as on some industrial examples coming from Dassault Aviation. The results are summarized in Table 1. The first column of Table 1 contains the name of the analysed benchmark suite. The second and third columns give respectively the lines of code and the total number of loops in the benchmark suite. The fourth column presents the number of loops for which r-TuBound inferred loop bounds. For a detailed evaluation of r-TuBound we refer to [5].

The Debie-1d and Mälardalen examples come from the WCET community and were used in the WCET tool challenges [11]. These examples are fine tuned for the WCET analysis of programs. Loop bounds need to be either inferred or assumed to be a priori given as program annotations. r-TuBound inferred loop bounds for 180 loops out of the 227 loops coming from the WCET community; some of the 180 loops could not yet be treated by other WCET tools, such as [10, 8]. The remaining 47 loops could not be handled by r-TuBound as various restrictions of equation (M) were violated. Namely, the loops had a nested-loop structure, loop updates contained operations over arrays and pointers, and non-linear and/or floating point arithmetic was used in the loop body,

We also run r-TuBound on 77 loops coming from Dassault Aviation. These examples have not yet been optimised for the WCET analysis of programs. When compared to [10], r-TuBounds infers non-trivial loop bounds for 46 loops. Out of these 46 loops, the approach of [10] can only handle 39 loops. The 7 loops that can only be treated by r-TuBound involved nested loops and multi-path reasoning with non-trivial linear arithmetic updates over the loop counter. r-TuBound failed on 31 loops coming from Dassault Aviation, as these loops required the analysis of nested loops with floating point arithmetic.

The current version of r-TuBound has successfully participated in the WCET 2011 tool challenge [11]. When compared to other WCET tools, such as Sweet [4] and OTAWA+oRange [7], we observed that the annotation language of r-TuBound has very little support for specifying variable input ranges or program execution frequencies. Moreover, r-TuBound was the only WCET tool whose results were obtained on the C16x microcontroller; the other WCET tools target the ARM7 or the Freescale MPC555x microcontrollers. Extending the annotation language and microcontroller support of r-TuBound is left for further work.

Experiments and runtime. We also analysed the runtime of the flow refinement and recurrence solving parts of r-TuBound. The pattern-based recurrence solving approach of `loopBounds` essentially takes no time: for every loop we tried, a loop bound is inferred in less than 0.5 seconds. The runtime performance of `loopRefine` is also relatively good; the flow refinement (i.e. parsing the code, executing the required SMT

Benchmark Suite	# LoC	# Loops	r-TuBound
Mälardalen	~ 7500	152	121
Debie-1d	~ 6100	75	59
Dassault	~ 1000	77	46
Total	~ 14700	304	226

Table 1. Experimental results with r-TuBound.

queries and writing back the simplified loop) of multi-paths loops with 1000 lines of code takes on average 5 - 20 seconds.

Our experiments thus suggest that r-TuBound is quite fast in practical application. We believe that improving the SMT based reasoning engine of `loopRefine`, for example by applying program slicing before monotonicity analysis, would yield overall better execution times for r-TuBound. We leave this task for further investigation.

4 Conclusion

r-TuBound offers software support for generating loop bounds in the WCET analysis of programs. The distinctive features of r-TuBound come with a pattern-based recurrence solving algorithm and over-approximating loop bounds of multi-path loops. For doing so, multi-path loops are translated into simple loops by using SMT encodings and deriving minimal updates over the loop counter. We presented the workflow of r-TuBound, illustrated how r-TuBound can be used on some example problems, and gave an overview on experimental results.

References

1. R. Blanc, T. Henzinger, T. Hottelier, and L. Kovacs. ABC: Algebraic Bound Computation for Loops. In *Proc. of LPAR-16*, pages 103–118, 2010.
2. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of TACAS*, pages 174–177, 2009.
3. G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.
4. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. of RTSS*, pages 57–66, 2006.
5. J. Knoop, L. Kovacs, and J. Zwirchmayr. An Evaluation of WCET Analysis using Symbolic Loop Bounds. In *Proc. of WCET*, 2011.
6. J. Knoop, L. Kovacs, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Proc. of PSI*, pages 116 – 126, 2011.
7. Marianne De Michiel and Armelle Bonenfant and Hugues Cassé and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *RTCSA*, pages 161–166, 2008.
8. Martin Schoeberl and Wolfgang Puffitsch and Rasmus Ulslev Pedersen and Benedikt Huber. Worst-case Execution Time Analysis for a Java Processor. *Software: Practice and Experience*, 40/6:507–542, 2010.
9. A. Prantl. The Termite Library. http://www.complang.tuwien.ac.at/adrian/termite/Manual/.
10. A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for WCET Analysis. In *Proc. of WCET*, pages 141–148, 2008.
11. R. von Hanxleden et al. The WCET Tool Challenge 2011: Report. In *Proc. of WCET*, 2011. under journal submission.