An Evaluation of WCET Analysis using Symbolic Loop Bounds

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr*

Vienna University of Technology

Abstract. In this paper we evaluate a symbolic loop bound generation technique recently proposed by the authors in [7]. The technique deploys pattern-based recurrence solving in conjunction with program flow refinement using SMT reasoning. The derived bounds are further used in the WCET analysis of programs with loops. This paper presents experimental evaluations of the method carried out with the r-TuBound software tool. We evaluate our method against various academic and industrial WCET benchmarks, and outline further challenges for symbolic loop bound computation.

1 Introduction

In this paper we present an evaluation of our symbolic loop bound computation method [7] for deriving tight upper bounds on the number of loop iterations. These bounds are further used in the WCET analysis of programs.

Given a program loop, the approach described in [7] derives tight upper bounds on the number of loop iterations as follows. First, if the loop contains multiple paths (i.e. multi-path loop) arising from conditionals, then the loop is over-approximated by a loop with only one path (i.e. simple loop). To this end, SMT reasoning in conjunction with control flow analysis is deployed. Next, the behavior of the simple loop is expressed by a system of recurrence equations over scalar loop variables and a new variable denoting the loop counter. These recurrence equations are then solved based on symbolic computation algorithms. Namely, a pattern-based recurrence solving algorithm over linear recurrences with constant coefficients is deployed. As a result, the values of scalar variables, i.e. the closed forms, at arbitrary loop iterations are derived as functions of the loop counter and initial values of the variables. If initial values of variables are missing, an over-approximation of non-deterministic assignments is deployed to derive sound initial values. Finally, closed forms of loop variables together with the loop condition are used to derive the smallest value of the loop counter such that the loop is terminated. To this end, SMT solving over arithmetical formulas is used. The inferred value of the loop counter yields an upper bound on the number of loop iterations of the simple loop, and consequently of the multi-path loop.

The approach was implemented in the r-TuBound software tool, and successfully tested on various examples.

* 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 was partly supported by Dassault Aviation and is partially funded by the FWF National Research Network RiSE (S11410-N23).

While the presentation in [7] focused on the methodology as such, we are now interested in evaluating the approach in practice on a number of benchmarks. This paper undertakes extensive investigation into understanding the practical power and limitations of the method and its implementation in r-TuBound. The key question we tried to address is the following. Is the method powerful enough to infer automatically loop bounds that can be further used for automating the WCET analysis of programs?

The main contribution of this paper is an experimental evaluation of the symbolic loop bound computation method, providing statistics and better understanding of the power of the method.

Related work. We only mention some of the many approaches our work can be compared to. For a detailed overview on related work we refer to [7].

In [13] the authors construct parameterized formulas to model the WCET of program functions or loops. To this end, closed form formulas that depend only on the number of loop iterations are generated. Unlike our approach, the inferred closed forms are evaluated at run-time to improve the performance of the system under analysis, by allowing optimized scheduling decisions and task selections.

In [1] the authors suggest algebraic techniques to construct a symbolic formula that characterizes the WCET of a function. For doing so, powerful computer algebra systems are used (CAS) to construct and simplify symbolic formulas. The described approach could be extended by symbolic summation in order to derive loop bounds and handle nested loops. The work presented in [1] relies on annotated programs and deploys a CAS, whereas we use a pattern-based recurrence solving approach to symbolic summation and deploy pre-computed closed form solutions.

The rest of this paper is structured as follows. Section 2 briefly overviews the symbolic loop bound computation method, which is then illustrated by concrete examples in Section 3. Section 4 describes the set of benchmarks used for experimental evaluations and details the obtained experimental results. We summarize the presented work in Section 5.

2 Symbolic Loop Bound Computation

In this section we briefly overview the main ingredients of the symbolic loop bound computation method of [7]. For more details and theoretic considerations we refer to [7].

Throughout this paper, a loop with only one path will be called a *simple loop*. Further, we will refer to a loop with multiple paths as a *multi-path loop*. In what follows, we will sometimes write *loop bound* or *loop iteration bound* to mean an an upper bound on the number of loop iterations.

In [7], special classes of simple and multi-path loops have been identified, as described below. For such loops a pattern-based recurrence solving algorithm together with control flow refinement using SMT reasoning is deployed to automatically derive tight loop iteration bounds.

```
for (fcode = (long) hsize;
    fcode < 65536L; fcode *= 2L)
  hshift++;
```

Fig. 1. Loop using using multiplication in the update expression.

```
while  (tries_left > 0) {
     ...
     tries_left--;
      if  (confirm_hit_result == 0)
         tries_left = 0;
}
```

Fig. 2. Loop with a conditional update to the loop counter.

Simple Loops with C-finite Updates. Consider the following simple loop pattern.

$$\texttt{for}\,(\texttt{i} = \texttt{a};\ \texttt{i} < \texttt{b};\ \texttt{i} = \texttt{c} * \texttt{i} + \texttt{d})\,; \tag{1}$$

where a, b, c, d, i, c are symbolic integer-valued constants such that a, b, c, d do not depend on i, and $c \neq 0$.

The variable i in (1) is called the loop iteration variable.

Note that updates over the iteration variable of (1) are given by linear arithmetic expressions over program variables. We call such updates C-finite updates.

For loops as in (1), a pattern-based recurrence solving algorithm over linear recurrences with constant coefficients is used to derive the value of i as a function of the initial values of loop variables and a new variable denoting the loop counter. That is, the closed form of i is derived. To this end, the analysis instantiates the closed form pattern of i with the appropriate symbolic constants of the loop (1). Further, SMT solving over arithmetic expression is deployed over the closed form of i and the loop condition, and a precise loop bound is derived expressing the smallest value of i at which the loop (1) terminates.

Let us note that in the case when the initial values of loop variables are missing, the approach of [7] soundly approximates the non-deterministic initial value assignments of loops (1), whenever the C-finite updates are expressed using left/right shift operations. We call a simple loop with C-finite updates involving only shift operations a *shift loop*.

Multi-path loops. Consider the multi-path loop patterns below.

$$\begin{array}{ll}
\texttt{for}\,(\texttt{i} = \texttt{a}; \texttt{i} < \texttt{b}; \texttt{i} = \texttt{c} * \texttt{i} + \texttt{d}) & \texttt{for}\,(\texttt{i} = \texttt{a}; \texttt{i} < \texttt{b}; \texttt{i} = \texttt{c} * \texttt{i} + \texttt{d}) \\
\quad \texttt{if}\,(\texttt{nondet}()) & \quad \texttt{if}\,(\texttt{B})\ \texttt{i} = \texttt{f}_1(\texttt{i}); \\
\quad\ \texttt{break}; & \quad \texttt{else}\ \ \texttt{i} = \texttt{f}_2(\texttt{i}); \\
\text{(a)} & \text{(b)}
\end{array} \tag{2}$$

where a, b, c, d, i, c are symbolic integer-valued constants such that a, b, c, d do not depend on i, $c \neq 0$, and f_1 and f_2 are monotonically increasing or decreasing linear functions.

The multi-path loop given in (2)(a) requires reasoning about abrupt termination, whereas (2)(b) involves conditional updates with linear monotonic functions f_1 and f_2. The method of [7] soundly translates the multi-path loops (2)(a) and (2)(b) into simple loops as in (1), by deploying SMT reasoning over arithmetic expressions. Next, loop bounds over the obtained simple loops are derived. These loops bounds are also tight loop bounds of the multi-path loops (2)(a), respectively (2)(b).

```
for  (i = 0; i < M; i++) {
   ...
  if (A[i] == 0) {
       failed = 1;
       break ;
  }
}
```

Fig. 3. Loop with abrupt termination.

```
while  (i > 0) {
   int  p = (i - 1) / 2;
   if  (nondet())  break ;
   i = p;
}
```

Fig. 4. Abruptly terminating loop with C-finite update.

```
while  (i < size) {
   int  j = 2 * i + 1;
   if  (nondet()) j++;
   if  (nondet())  break ;
   i = j;
}
```

Fig. 5. Abrupty terminating loop with C-finite and conditional updates.

Let us note that (2) does not cover arbitrary multi-path loops. For example, nested loops cannot yet be handled by our approach. We leave the study of nested loops and the generalization of (2) to future research.

3 Challenging Examples and Benchmarks

The following loop examples are taken from various benchmarks and illustrate the main ingredients of symbolic loop bound computation.

Fig. 1 lists a loop with a C-finite update. It can be characterized by the following linear recurrence relation:

$$fcode(n+1) = 2 * fcode(n), \quad \text{where } n \geq 0 \text{ denotes the loop iteration counter.}$$

Hence, the value of $fcode$ at an arbitrary iteration n can be derived by instantiating the closed form solution of (1). As a result, the value of $fcode$ is expressed as a function of n. Note that the loop condition $fcode < 65536L$ holds at any loop iteration, and therefore $fcode(n) < 65536L$ is a valid formula. A precise loop bound for Fig. 1 is then derived by computing the smallest value of n such that the loop terminates. In other words, the (smallest) value of n is inferred such that the formula $(fcode(n) < 65536L) \ \wedge \ (fcode(n+1) \geq 65536L)$ is satisfiable. This satisfiability problem is encoded as an SMT query over arithmetical formulas.

Consider the loop from Fig. 2. This loop fits the loop pattern given in (2)(b). Deriving a tight loop bound for Fig. 2 requires reasoning about the the conditional update to $tries\_left$. To this end, the loop iterates over $tries\_left$ either by monotonically decreasing the value of $tries\_left$, or by setting the value of $tries\_left$ to 0. As in both cases the value of $tries\_left$ decreases[1], the conditional statement of Fig. 2 is omitted and Fig. 2 is approximated by a simple loop of the form (1). Further, a precise loop bound for the simple loop is computed, yielding thus a tight loop bound for Fig. 2.

Fig. 3 lists an abruptly terminating loop. The loop fits the loop pattern given in (2)(a). The difficulty in deriving a tight loop bound for Fig. 3 comes from the presence of the conditional statement yielding an abrupt termination of the loop. To this end, the control flow of Fig. 3 is refined in [7] by abstracting away the $break$ statements, and Fig. 3 is approximated by a simple loop with C-finite updates as in (1). A precise loop bound of the simple bound is next derived, from which a tight loop bound for Fig. 3 is obtained.

[1] note that the loop condition assumes that $tries\_left$ is a positive non-zero symbolic scalar

Benchmark Suite	#Loops	TuBound	r-TuBound	Types
Mälardalen	207	163	165	AT, CF
Debie	75	58	59	SH, CU
Scimark	34	24	26	AT, CF
Dassault	77	39	46	AT, CF-CU-AT (2), CU(4)
Total	393	284	296	AT (3), SH, CU (5), CF (2), CF-CU-AT (2)

Table 1. Experimental results and comparisons with r-TuBound and TuBound.

The examples given so far described loops fitting exactly one of the loop patterns (1), (2)(a), and (2)(b). Our experiments show however that these loop patterns are used in their combination. To this end, consider Fig. 4 and Fig. 5.

The loop from Fig. 4 requires reasoning about abrupt termination and C-finite updates. A tight loop bound for Fig. 4 is inferred by first applying flow refinement to transform Fig. 4 into a simple loop with C-finite update, and then derive a precise loop bound for the obtained simple loop.

The program from Fig. 5 implements an abruptly terminating loop with C-finite and conditional linear updates. After flow refinement and establishing the monotonicity property of the conditional update, Fig. 5 is rewritten into a simple loop with only C-finite updates. A precise loop bound of the simple loop is next computed by applying the pattern-based recurrence solving approach of [7], and a loop bound of Fig. 5 is finally inferred.

4 Experimental Evaluation

The approach of [7] is implemented in the r-TuBound software tool. r-TuBound extends the TuBound framework [10, 6] by an SMT-based control flow refinement approach, a pattern-based recurrence solving algorithm, and a loop preprocessing step applying simple loop rewriting techniques. In this section we report on experimental results obtained by applying r-TuBound on a number of challenging examples. These examples include (i) WCET benchmarks, such as the Mälardalen and Debie benchmark suite [4], (ii) scientific benchmarks, such as the SciMark2 benchmark suite, and (iii) industrial benchmarks using a number of examples sent by Dassault Aviation.

In our experiments, we aimed at the automated derivation of loop bounds. To this end, we did not investigate the low-level WCET calculus on the underlying hardware architecture, but only applied a high-level analysis of program loops on the software level.

Let us note that in our experiments we did not use r-TuBound in conjunction with the software model checking extension of [9]. The results reported below were obtained by only deploying the symbolic loop bound computation framework of [7] implemented in r-TuBound.

4.1 Benchmarks and Evaluations

WCET Benchmarks. The Mälardalen Real Time and the Debie benchmark suites are both well known in the WCET community and were used in the WCET tool challenges since 2006 [4].

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
adpcm	18	17	17	-	qurt	1	1	1	-
bs	1	0	0	-	select	4	0	0	-
bsort100	3	3	3	-	statemate	1	0	0	-
cnt	4	4	4	-	sqrt	1	1	1	-
cover	3	3	3	-	fft1	11	6	6	-
crc	3	3	3	-	lms	10	6	6	-
duff	2	1	1	-	whet	11	11	11	-
edn	12	12	12	-	ludcmp	11	11	11	-
expint	3	3	3	-	compress	7	2	3	CF
fibcall	1	1	1	-	fir	2	1	1	-
janne_c	2	0	0	-	minver	17	16	16	-
nsichneu	1	0	0	-	qsort_exam	6	0	1	AT
insertsort	2	0	0	-	fdct	2	2	2	-
jfdctint	3	3	3	-	lcdnum	1	1	1	-
matmult	5	5	5	-	ndes	12	12	12	-
ns	4	4	4	-	**sum**	207	163	165	2

Table 3. Evaluation of r-TuBound on the Mälardalen benchmarks.

From the Mälardalen suite we investigated 31 files, containing 207 loops. r-TuBound analyzed all simple loops with constant increments/decrements (i.e. a special case of (1) with $c = 1$) that were analyzed also by TuBound. However, r-TuBound inferred loop bound also for two additional loops on which TuBound failed since reasoning about C-finite updates (with $c \neq 1$ in (1)) and abrupt termination was required. In contrast to [7] we evaluated r-TuBound on more programs from the Mälardalen suite. The additional examples contained almost only simple loops, for example the `minver.c` file containing 16 simple loops, and the `ndes.c` program containing 12 simple loops (see Table 3 from Section 4.2).

The Debie benchmark suite contains 75 loops. r-TuBound successfully analyzed 59 loops. From these 59 loops, 58 simple loops were bound also by TuBound. Let us however note, that in the presence of initial value information, r-TuBound finds tighter bounds for simple loops with C-finite updates involving shift operations. The one loop that TuBound was not able to analyze is a loop with monotonic conditional update as in (2)(b).

When compared to TuBound, one may say that r-TuBound brings an insignificant increase in performance when applied on the WCET benchmarks. However, let us note that these benchmarks have been used in the WCET community already since 2006. Therefore, it is expected that WCET tools, including TuBound, implement various heuristics and perform very well on these benchmarks. The benefit of r-TuBound can be better evidenced on new examples, including industrial ones, with relatively complex arithmetic and control flow. We report on these results below.

Scientific Benchmarks. The SciMark2 benchmark suite contains 34 loops. r-TuBound derived loop bounds for 26 loops, whereas TuBound could analyze 24 loops. The 2

BM	#L	TB	r-TB	G
class	2	2	2	SH
debie	1	0	0	-
harness	45	34	34	-
health	11	9	10	CU
hw_if	3	2	2	-
measure	6	4	4	-
tc_hand	3	3	3	-
telem	4	4	4	-
sum	75	58	59	1

Table 5. Evaluation of r-TuBound on the Debie benchmarks.

loops which could only be handled by r-TuBound required reasoning about abrupt-termination and C-finite updates.

Industrial Benchmarks. We evaluated r-TuBound on 77 loops sent by Dassault Aviation. r-TuBound inferred loop bounds for 46 loops, whereas TuBound analyzed only 39 loops. When compared to TuBound, the success of r-TuBound lies in its power to handle abrupt termination, conditional updates, and C-finite behavior.

Analysis of Experiments. Altogether, we ran r-TuBound on 4 different benchmark suites, on a total of 393 loops and derived loop bounds for 296 loops. Out of these 296 loops, 286 loops were simple and involved only C-finite reasoning, and 10 loops were multi-path loops which required the treatment of abrupt termination and conditional updates. TuBound could handle 284 simple loops only.

We note that, 75% of the 393 loops were successfully analyzed by r-TuBound, whereas TuBound succeeded on 72% of the 393 loops.

When compared to TuBound, the overall quality of loop bound analysis within r-TuBound has increased by 3% (72% to 75%). However, TuBound already performed well on Mälardalen and Debie, and therefore the increase given by r-TuBound is only of 1% (78% to 79% in Mälardalen and 77% to 78% in Debie). For the SciMark2 and the examples from Dassault, the increase in performance given by r-TuBound is of 6% (70% to 76%) and 9% (50% to 59%), respectively. The practical importance of r-TuBound can thus be better evidenced on examples with more complex arithmetic and/or control-flow.

Table 1 lists a summary of the experimental results obtained by using r-TuBound on the aforementioned four benchmark suites. Column 1 lists the benchmark suite, column 2 the number of loops contained, columns 3 and 4 list respectively the number of loops analyzed by TuBound and r-TuBound. Column 5 describes the type of loop and why they could only be analyzed by r-TuBound. To this end, we distinguish between simple loops with C-finite updates (CF), shift-loops with non-deterministic initializations (SH), multi-path loops with abrupt termination (AT), and multi-path loops with monotonic conditional updates (CU). Column 5 also lists, in parenthesis, how many of such loops were encountered. For example, among the loops sent by Dassault Aviation 4 multi-path loops with monotonic conditional updates, denoted as CU(4), could only be analyzed by

BM	#L	TB	r-TB	Gain	BM	#L	TB	r-TB	Gain
array	6	5	6	AT	scimark	0	0	0	-
fft	8	4	5	CF	sor	3	3	3	-
kernel	9	4	4	-	sparsecomprow	3	3	3	-
montecarlo	1	1	1	-	stopwatch	0	0	0	-
random	4	4	4	-	**sum**	34	24	26	2

Table 7. Evaluation of r-TuBound on the SciMark2 examples.

r-TuBound. Some loops require combinations of the proposed techniques, for example, multi-path loops with C-finite conditional updates and abrupt termination; such loops are listed in Table 1 as CF-CU-AT.

A detailed evaluation of r-TuBound and comparisons with TuBound is given in Section 4.2.

Limitations. We also investigated some of the examples on which r-TuBound failed to derive loop bounds. We list below some limitations of r-TuBound. Handling these limitations are left for future work.

One source of limitations is the presence of loops with more complex arithmetic updates than the ones captured by the C-finite updates of (1). Such a non-C-finite update is given by the update listed below:

$$\mathtt{d} = \mathtt{d} * -\mathtt{r} * \mathtt{r}/(2 * \mathtt{i}) * (2 * \mathtt{i} + 1); \quad \text{(where d is used in the loop condition).}$$

Reasoning about such updates would require solving recurrences with non-constant polynomial coefficients in the loop variables. Algorithmic methods are available for solving such recurrences – see [12, 5].

Another reason of r-TuBound's failure comes from the need of supporting numeric functions, for example, the absolute value function.

Other loops that r-TuBound cannot yet handle are simple loops that, in addition to the loop counter, also modify a variable in the loop condition. We list such an example below:

$$\mathbf{while}\,(\mathtt{i} < \mathtt{j})\ \{\mathtt{j} = \mathtt{j} - \mathtt{i};\ \ \mathtt{i} = \mathtt{i}/2;\}$$

In addition to the above mentioned limitations, our experiments showed that some loops require reasoning about the array content. Such an example is given by the following simple loop iterating over a character string:

$$\mathbf{while}\,(\mathtt{c}[\mathtt{i}]\ != \ 0)\ \{\mathtt{i}++;\}$$

To this end, we plan to extend r-TuBound with support for arrays by using SMT solvers [8, 3] or first-order theorem provers [11].

We did not yet investigate our approach for nested loops. We plan to study our work in conjunction with the approach of [2], where a large number of nested loops are handled using symbolic computation techniques over loop indexes.

4.2 Experimental Details and Results

In the following tables, column 1 ("BM") denotes the name of the benchmark file/program, and column 2 gives the number of loops in the corresponding file ("#L"). Column 3 lists how many of those loops were successfully analyzed by TuBound ("TB"), whereas column 4 lists how many loops were analyzed by the symbolic loop bound computation technique of r-TuBound("r-TB"). Column 5 ("G") shows which loop patterns (cf. the notation used in Table 1) could only be handled by r-TuBound[2].

Table 3 reports on the results obtained by running r-TuBound on 31 examples files from the Mälardalen benchmark set. Altogether, we ran r-TuBound on 207 loops from the Mälardalen suite. r-TuBound derived loop bound for 165 loops, whereas TuBound was able to analyze 163 loops. The two additional loops analyzed by r-TuBound required recurrence solving and control flow refinement.

Table 5 shows the performance of r-TuBound on examples from the Debie benchmark set. Altogether, we used 8 example files containing 75 loops. TuBound was able to compute loop bounds for 58 loops. In addition to these 58 loops, r-TuBound also inferred loop bound to one additional shift-loop.

Table 7 gives the performance of r-TuBound on 34 loops from the SciMark benchmark suit. In addition to the loops that could also be analyzed by TuBound, r-TuBound also inferred loop bounds on two loops with C-finite updates and abrupt termination.

Table 9 reports on the experimental results obtained by running r-TuBound on loops sent by Dassault Aviation. We evaluated r-TuBound on 51 benchmark files containing altogether 77 loops. Out of the 77 loops, r-TuBound derived loop bounds for 46 loops, whereas TuBound analyzed 39 loops. The 7 loops that could only be analyzed by r-TuBound included 4 loops with C-finite and conditional updates, 2 abruptly terminating loops with C-finite and conditional updates, and 1 abruptly terminating loop with C-finite updates.

5 Conclusion

We evaluate the symbolic loop bound computation method implemented in the r-TuBound tool. The method combines pattern-based recurrence solving with control flow refinement using SMT reasoning, and over-approximates non-deterministic initializations. Our experimental results give practical evidence of the applicability of r-TuBound for loop bound computation and WCET analysis.

Further work includes extending r-TuBound with more powerful symbolic computation algorithms, such as solving recurrences with polynomial coefficients. We also plan to combine r-TuBound with first-order theorem proving algorithms and derive loop bounds of programs whose behavior requires reasoning about array, lists, and pointers.

References

1. G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *In 25th IFAC Workshop on Real-Time Programming*, 2000.

[2] Subtracting column 5 from column 4 yields the number of simple loops with constant increments/decrements (i.e. $c = 1$ in (1))

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
all_zeros	3	1	1	-	minimum_sort	2	2	2	-
array_ptr	3	3	3	-	min_sort	2	2	2	-
asm_memset2	2	1	1	-	muller	2	2	2	-
behavior	1	1	1	-	nb_occ	1	1	1	-
b_s_o	1	0	0	-	negate	1	1	1	-
break	3	1	1	-	ovt_vs_poly	1	0	0	-
bresenham	1	1	1	-	permut_search2	1	0	0	-
bsearch	1	0	0	-	permut_search	1	0	0	-
bts0041-bis	3	3	3	-	r_strcpy	1	0	0	-
bts0041	3	3	3	-	string_constant	1	0	0	-
continue	3	0	3	CU,CU,CU	struct_hack	3	0	0	-
copy	1	0	0	-	sum1	1	1	1	-
count_bits	1	0	0	-	sum2	2	2	2	-
dillon4	1	1	1	-	test5_floats	1	0	0	-
division	1	0	0	-	trace.c	2	2	2	-
dowhile	1	1	1	-	vamos	2	0	0	-
fs253	1	0	0	-	vieira1	2	2	2	-
fs256	1	1	1	-	vieira2	1	0	0	-
fs350	1	1	1	-	weber1	1	1	1	-
ghost_label	1	1	1	-	weber3	1	0	0	-
heap	2	0	2	AT, CF-AT-CU	weber4	1	0	0	-
heapsort	3	1	2	CF-AT-CU	weber5	1	0	0	-
inv_perm_minimal	2	1	1	-	weber6	1	0	0	-
loop_eq	1	0	0	-	weber8	1	0	0	-
loop_inv	1	0	1	CU	weber9	1	1	1	-
malloc	1	1	1	-					
					sum	77	39	46	7

Table 9. Evaluation of r-TuBound on examples sent by Dassault Aviation.

2. R. Blanc, T. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Proc. of LPAR-16*, pages 103–118, 2010.
3. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of TACAS*, pages 174–177, 2009.
4. N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat, and M. Schordan. WCET 2008 - Report from the Tool Challenge 2008 - 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. In *Proc. of WCET*, 2008.
5. M. Kauers. SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *J. of Symbolic Computation*, 41(9):1039–1057, 2006.
6. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond Loop Bounds: Comparing Annotation Languages for Worst-Case Execution Time Analysis. *J. of Software and System Modeling*, 2010. Online edition.

7. J. Knoop, L. Kovacs, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Proc. of PSI*, 2011. To appear.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340, 2008.
9. A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. From Trusted Annotations to Verified Knowledge. In *Proc. of WCET*, pages 39–49, 2009.
10. A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint Solving for High-Level WCET Analysis. In *Proc. of WLPE*, pages 77–89, 2008.
11. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
12. C. Schneider. Symbolic Summation with Single-Nested Sum Extensions. In *Proc. of ISSAC*, pages 282–289, 2004.
13. E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In *Proc. of LCTES*, pages 88–93, 2001.