FROM TRUSTED ANNOTATIONS TO VERIFIED KNOWLEDGE [1]

Adrian Prantl, Jens Knoop, Raimund Kirner and Albrecht Kadlec
Technische Universität Wien, Austria
{adrian,knoop}@complang.tuwien.ac.at
{raimund,albrecht}@vmars.tuwien.ac.at
and
Markus Schordan
University of Applied Sciences Technikum Wien, Austria,
schordan@technikum-wien.at

Abstract
WCET analyzers commonly rely on user-provided annotations such as loop bounds, recursion depths, region- and program constants. This reliance on user-provided annotations has an important drawback. It introduces a Trusted Annotation Basis into WCET analysis without any guarantee that the user-provided annotations are safe, let alone sharp. Hence, safety and accuracy of a WCET analysis cannot be formally established. In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds. Fundamental to our approach is to apply model-checking in concert with other more inexpensive program analysis techniques, and the coordinated application of two algorithms for Binary Tightening *and* Binary Widening*, which control the application of the model-checker and hence the computational costs of the approach. Though in this paper we focus on the control of model-checking by Binary Tightening and Widening, this is embedded into a more general approach in which we apply an array of analysis methods of increasing power and computational complexity for proving or disproving relevant time bounds of a program. First practical experiences using the sample programs of the Mälardalen benchmark suite demonstrate the usefulness of the overall approach. In fact, for most of these benchmarks we were able to empty the trusted annotation base completely, and to tighten the computed WCET considerably.*

1. Motivation

The computation of loop bounds, recursion depths, region- and program constants is undecidable. It is thus commonly accepted that WCET analyzers rely to some extent on user-assistance for providing bounds and constants. Obviously, this is tedious, complex, and error-prone. State-of-the-art approaches to WCET analysis thus provide for a fully automatic preprocess for computing required bounds and constants using static program analysis. This unburdens the user since his assistance is reduced to bounds and constants, which cannot automatically be computed by the methods employed by the preprocess. Typically, these are classical data-flow analyses for constant progagation and folding, range analysis and the like, which are particularly cost-efficient but may fail to verify a bound

[1]This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract P18925-N13.

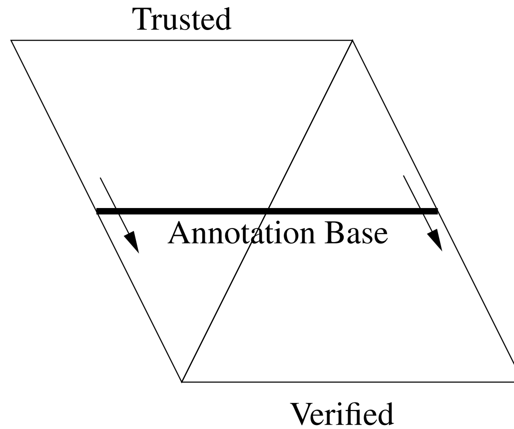

Figure 1. The annotation base: shrinking the trusted annotation base and establishing verified knowledge about the program

or the constancy of a variable or term. WCET analyzers then rely on user-assistance to provide the missing bounds which are required for completing the WCET analysis. This introduces a *Trusted Annotation Base* (*TAB*) into the process of WCET analysis. The correctness (safety) and optimality (tightness) of the WCET analysis depends then on the safety and tightness of the bounds of the TAB provided by the user.

In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds.

Figure 1 illustrates the general principle of our approach. At the beginning the entire annotation base that is added by the user where static analysis fails to establish the required information, is assumed and trusted to be correct, thus we call it *Trusted Annotation Base* (*TAB*). Using model checking we aim to verify as many of these user-provided facts as possible. In this process we shrink the trusted fraction of the annotation base and establish a growing verified annotation base. In Figure 1 the current state in this process is visualized as the horizontal bar. In our approach we are lowering this bar, representing the decreasing fraction of trust to an increasing fraction of verified knowledge, and thus transfer *trusted user-belief* into *verified knowledge*.

In this paper propose a systematic approach to making best use of it exploring thus the implications of trust. These are two-fold. First, TAB information allows for feeding-back user-provided timing information into the program analyses of the preprocess.

2. Shrinking the Trusted Annotation Base – Sharpening the Time Bounds

2.1. Shrinking the Trusted Annotation Base

The automatic computation of bounds by the preprocesses of up-to-date approaches to WCET analysis is a step towards keeping the trusted annotation base small. In our approach we go a step further to shrinking the trusted annotation base. In practice, we often succeed to empty it completely.

A key observation is that a user-provided bound – which the preprocessing analyses were unable to compute – can not be checked by them either. Hence, verifying the correctness of the corresponding user annotation in order to move it *a posteriori* from the trusted annotation base to the verified knowl-

edge base requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for the detection of copy constants, linear constants, simple constants, conditional constants, up to finite constants detecting different classes of constants at different costs [5]. This provides evidence for the variety of available choices for analyses using the example of constant propagation. While some of these algorithms might in fact well be able to verify a user annotation, none of these algorithms is especially prepared and well-suited for verifying a data-flow fact at a particular program location, a *data-flow query*. This is because these algorithms are exhaustive in nature. They are designed to analyze the whole program. They are not focused towards deciding a data-flow query, which is the domain of *demand-driven program analyses* [3, 4]. Like for the more expressive variants of constant propagation and folding, however, demand-driven variants of program analyses are often not available.

In our approach, we thus propose to use *model checking* for the *a posteriori* verification of user-provided annotations. Model checking is tailored for the verification of data-flow queries. Moreover, the development of software model checkers made tremendous progress in the past few years and are now available off-the-shelf. The Blast [1] and the CBMC [2] model checkers are two prominent examples. In our experiments reported in Section 4 we used the CBMC model checker.

The following example demonstrates the ease and elegance of using a model checker to verify a loop bound, which we assume could not be automatically bounded by the program analyses used. The program fragment in this example shows a loop together with a user-provided annotation of the loop and the formula presented to CBMC to verify or refute the user-provided annotation.

```
int binary_search(int x) {
  int fvalue, mid, up, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */
  {
    unsigned int __bound = 0;
    while(low <= up){
      mid = low + up >> 1;
      if (data[mid].key == x) { /* found  */
        up = low - 1;
        fvalue = data[mid].value;
      }
      else if (data[mid].key > x)
        up = mid - 1;
      else low = mid + 1;

      __bound += 1;
    }
    assert(__bound <= 7);
  }
  return fvalue;
}
```

In this example, the CBMC model checker comes up with the answer "yes," i.e., the loop bound provided by the user is safe; allowing thus for its movement from the trusted to the verified annotation base. If, however, the user were to provide $bound \leq 3$ as annotation, model checking would fail and produce a counter example as output. Though negative, this result is still most valuable. It allows for preventing usage of an unsafe trusted annotation base in a subsequent WCET analysis. Note that the counter example itself, which in many applications is the indispensable output of a failed run of a model checker, is not essential for our application. It might be useful, however, to present it to the user when asking for another candidate of a bound, which can then be subject to *a posteriori* verification in the same fashion until a safe bound is eventually found.

Next we introduce a more effective approach to come up with a safe and even tight bound, if so existing, which does not even rely on any user interaction. Fundamental for this are the two algorithms *Binary Tightening* and *Binary Widening* and their coordinated interaction. The point of this coordination is to make sure that model checking is applied with care as it is computationally expensive.

2.2. Sharpening the Time Bounds

2.2.1. Binary Tightening

Suppose a loop bound has been proven safe, either by model checking or a program analysis of the preprocess. Typically, this bound will not be tight. In particular, this will hold for user-provided bounds. In order to exclude channeling an unsafe bound into the trusted annotation base, the user will generously err on the side of caution when providing a bound. This suggests the following iterative approach to tighten the bound, which is an application of the classical algorithms for binary search, thus called *binary tightening* in our scenario.

Let b denote the value of the initial bound. Per definition b is a positive integer. Then: call procedure $binaryTightening$ with the interval $[0..b]$ as argument, where $binaryTightening([low..high])$ is defined as follows:

1. Set m to $\lfloor \frac{low+high}{2} \rfloor$.
2. ModelCheck(m is a safe bound):
3. no: if $m \neq high$: **return** $binaryTightening([m..high])$ else **return** $false$
4. yes: if $low \neq m$: **return** $binaryTightening([low..m])$ else **return** m

Obviously, $binaryTightening$ terminates. If the return value of $binaryTightening$ is false, a tighter bound than that of the initial value of $high$ could not be established. Otherwise, i.e., after termination with the return value m, m is the least safe bound. This means m is tight. If m is smaller than the initial value of $high$, we succeeded to sharpen the bound.

Binary widening described next allows for proceeding in this case with searching for a safe bound, if one exists, without any further user interaction.

2.2.2. Binary Widening

Binary widening is dual to binary tightening. Its functioning is inspired by the risk-aware gambler playing roulette betting money exclusively on a 50% chance like red or black. In principle, he can flatten any loss by doubling his bet in the next game. In reality, the maximum bet allowed by the casino or his limited monetary resources, whatever is lower, prevent this strategy to work out in reality. Nonetheless, this externally given limit yields the inspiration for the *Binary Widening* algorithm to come up with a safe bound, if one exists, and to terminate, if the size of the bound is too big to be reasonable, or does not exist at all.

1. if $b > limit$: **return** false

2. ModelCheck(b is a safe bound):

3. yes: **return** b

4. no: $binaryWidening(2 * b)$

Obviously, $binaryWidening$ terminates, if a safe bound exists.[1] Otherwise, it will diverge. Hence, there must be a rule to avoid looping of $binaryWidening$. A simple approach is to call $binaryWidening$ only a predefined maximum number of times. This corresponds to the limit set by a casino to a maximum bet. The ratio behind this approach is the following: if a bound exists, but exceeds a pre-defined threshold, it can be considered practically useless. In fact, this scenario might indicate a programming error and should thus be reported to the programmer for inspection. A more refined approach might set this threshold more sophisticatedly, by using application dependent information, e.g., such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time this loop shall be allowed for.

2.2.3. Coordinating Binary Tightening and Widening

Once a safe bound has been determined using binary widening, binary tightening can then be used to compute the uniquely determined safe tight bound. Because of the exponential resp. logarithmic behaviour in the selection of arguments for binary widening and tightening, model checking is called moderately often. This is the key for the practicality of our approach, which we implemented in our WCET analyzer TuBound, as described in Section 3. The results of practical experiments we conducted with the prototype implementation are most promising. They can be found in Section 4.

3. Implementation within TuBound

3.1. TuBound

TuBound [8] is a research WCET analyzer tool which is unique for uniformly combining static program analysis, optimizing compilation and WCET calculation. Static analysis and program optimization take place at a very high level of abstraction given by the abstract syntax trees of a subset of the C++ language. TuBound is built upon the SATIrE program analysis framework [11] and the TERMITE program transformation environment.[2] TuBound features an array of algorithms for loop analysis of differing accuracy and computation cost including sophisticated analysis methods for nested loops. A detailed account of these methods the reader is given in [7].

3.2. Implementation

The Binary Widening/Tightening algorithms are implemented using a TERMITE source-to-source transformer. The specific transformer implemented locates the first occurrence of a `wcet_trusted_loopbound(N)` annotation in the program source and then proceeds to rewrite the encompassing loop in the fashion sketched in Figure 2.[3] For simplicity we assume that the loop is free of extraneous

[1] In practice, the model checker might run out of memory before verifying a bound, if it is too large, or may take too much time for completing the check.

[2] http://www.complang.tuwien.ac.at/adrian/termite

[3] For enhanced readability, the extra arguments containing file location and other bookkeeping information have been replaced by "...".

```
assertions(..., Statement, AssertedStatement) :-
  Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
  get_annot(Stmts, wcet_trusted_loopbound(N), _),

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

  AssertedStatement =
    basic_block([CounterDecl,
                 while_stmt(Test, basic_block([Count|Stmts], ...), ...),
                 CounterAssert], ...).
```

Figure 2. Excerpt from the source-to-source transformer

```
...
int complex(int a, int b)
{
  while(a < 30) {
#pragma wcet_trusted_loopbound(16)
      while(b < a) {
#pragma wcet_trusted_loopbound(16)
        if (b > 5)
          b = b * 3;
        else
          b = b + 2;
        if (b >= 10 && b <= 12)
          a = a + 10;
        else
          a = a + 1;
      }
      a = a + 2;
      b = b - 10;
    }
  return 1;
}
...
```

→

```
...
int complex(int a, int b)
{
    while(a < 30) {
#pragma wcet_loopbound(16)
      {
        unsigned int __bound = 0U;
        while(b < a){
#pragma wcet_trusted_loopbound(16)
          ++__bound;

         if (b > 5)
           b = b * 3;
         else
           b = b + 2;
         if (b >= 10 && b <= 12)
           a = a + 10;
         else
           a = a + 1;
      }
      assert(__bound <= 16U);
    }
    a = a + 2;
    b = b - 10;
  }
  return 1;
}
...
```

Containing two trusted loop annotations — Outer loop annotation verified, inner being checked

Table 1. Implementation of binary widening

control flow introduced by `goto` statements, which can easily be checked beforehand. Surrounding the loop statement, a new compound statement is generated, which accommodates the declaration of a new unsigned counter variable which is initialized to zero upon entering the loop. Inside the loop, an increment statement of the counter is inserted at the very first location. After the loop, an assertion is generated which states that the count is of most of the value N, where this value is taken from the annotation.

The application of the transformer is controlled by a driver, which calls the transformer for every trusted annotation contained in the source code file. Depending on the result of the model checker and the coordinated application of the algorithms for binary widening and tightening, the value and the status of each annotation is updated. In the positive case, this means the status is changed from *trusted annotation* to *verified knowledge*, while simultaneously the value of the originally trusted bound is replaced by a sharper, now verified bound. In Table 1, which shows a snapshot of processing the *janne_complex* benchmark, this status and value change is highlighted by different colors.

Benchmark	Loops	TuBound basic	with Model Checking	Runtime
adpcm	18	83.3%	83.3%	timeout
bs	1	0%	100%	$0.03s$%
bsort100	3	100%	100%	–
cnt	4	100%	100%	–
compress	7	28.5%	28.5%	timeout
cover	3	100%	100%	–
crc	3	100%	100%	–
duff	2	50%	50%	$0s$
edn	12	100%	100%	–
expint	3	100%	100%	–
fdct	2	100%	100%	–
fft1	11	54.5%	81.8%	$0.43s$
fibcall	1	100%	100%	–
fir	2	50%	50%	timeout
insertsort	2	0%	0	timeout
janne_complex	2	0%	100%	$0.18s$
jfdctint	3	100%	100%	–
lcdnum	1	100%	100%	–
lms	10	60%	60%	timeout
ludcmp	11	100%	100%	–
matmult	5	100%	100%	–
minver	17	94.1%	100%	$0.06s$
nsichneu	1	0%	100%	$5.59s$
qsort-exam	6	0%	66.6%	$0.02s$
qurt	1	100%	100%	–
recursion	0	–	–	–
select	4	0%	0%	timeout
statemate	1	0%	100%	$0.06s$
sqrt	1	100%	100%	–
st	5	100%	100%	–
whet	10	100%	100%	–
Total Percentage		75.6%	84.2%	

Table 2. Results for the Mälardalen benchmarks

4. Experimental Results

We implemented our approach as an extension of the TuBound WCET analyzer and applied the extended version to the well-known Mälardalen WCET benchmark suite. As a baseline for comparison we used the 2008 version of TuBound, which took part in the 2008 WCET Tool Challenge, later on called the basic version of TuBound.

The experiments we conducted were guided by two questions: "Can the number of automatically bounded loops be increased significantly?" and "How expensive is the process?" The benchmarks were performed on a 3 GHz Intel Xeon, running 64-Bit Linux. The model checker used was CBMC 2.9, which we tried for loop bounds up to the size of $2^{13} = 8192$ and a timeout of 60 seconds.

Our findings are summarized in Table 2. The third column of this table shows the percentage of loops that can be analyzed by the basic version of TuBound; the fourth column shows the total percentage of loops the extended version of TuBound was able to bound. The last column shows the cumulated runtime of the model checker for the remaining loops.

Comparing column 3 and 4 reveals the superiority of the extended version of TuBound over its basic variant. The extended version succeeds to bound 63% of the loops the basic version failed to bound.

Considering column 5, it can be seen that the model checker terminates quickly on small problems, but that the runtime and space requirements can increase to practically infeasible amounts on problems that suffer from the state explosion problem, where success would require more than an uninformed

enumeration of the complete state space. Such a behaviour can be triggered, if the initialization values which are part of the majority of the Mälardalen benchmarks are manually invalidated by introducing a faux dependency on e.g. `argc`. This demonstrates that model checking should be used with care or fed with additional information guiding the model checker and simplifying the verification task.

The fully-fledged variant of our approach, which we highlight in the next section is tailored towards this.

5. Extensions: The Fully Fledged Approach

The shrinking of the trusted annotation base and sharpening of time bounds, as described in Section 2, is based on model checking. Based on our experience, we believe that the model checking approach can be very valuable in the real world when

(i) it is combined with advanced program slicing techniques to reduce the state space and (ii) the results of static analyses (like TuBound's variable-interval analysis) are used to narrow the value ranges of variables, thus regaining a feasible problem size. This leads to the following extension of our approach to improve efficiency:

1. By using a pool of analysis techniques with different computational complexity: As shown in Figure 3, model checking is considered as one of the most complex analysis methods. On the other side, techniques like *constant propagation* or *interval analysis* are relatively fast. Thus we are interested in exploiting the fast techniques wherever beneficial and using the relatively complex techniques rarely.
2. By using a smart activation mechanism for the different analysis techniques: As shown on the right of Figure 3 we are interested in the interaction of the different analysis techniques. We do not aim to use the pool of analysis techniques in waves of different complexity, i.e., first applying the fast techniques and then gradually shifting towards the more complex techniques. Instead we aim for a smart interaction of the different analysis techniques. For example, whenever a technique with relatively high computational complexity has been applied, we can again apply techniques of relatively simple complexity to compute the closure of flow information based on previously obtained results.

We also think that profiling techniques are useful to guide the heuristics to be used within our static analysis techniques. For example, execution samples obtained by profiling can be used to elicit propositions to be verified by model checking.

The fully fledged approach envisioned in this section provides the promising potential as a research platform for complementing program analysis techniques.

6. Conclusions

Model checking has been used before in the context of WCET analyzers. Prominent examples are the ForTAS [6], MoDECS [12], and ATDGEN projects [9, 10], which are concerned with measurement-based WCET analysis. In these two projects, model checking is used to generate test data for the execution of specific program paths. Intuitively, in these application the model checker is presented

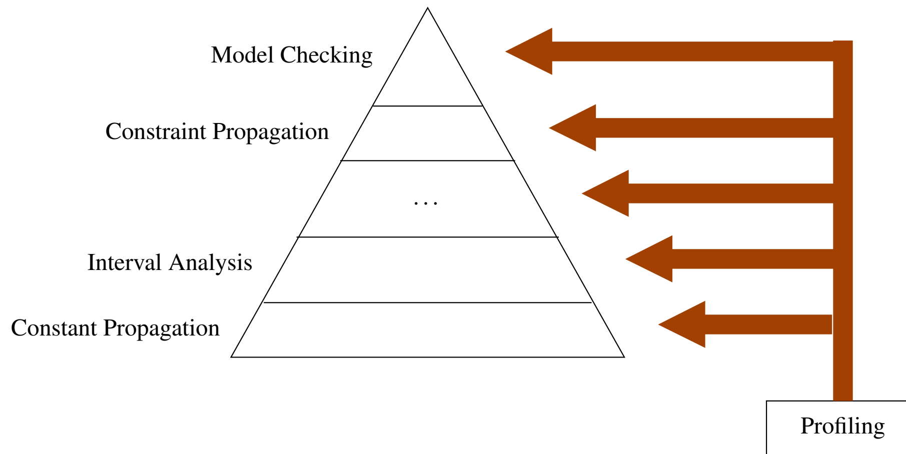

Figure 3. Pool of complementary analysis techniques with different complexity

with formulae stating that a specific program path is infeasible. If this can be refuted by the model checker, the counter example generated provide the test data ensuring the execution of the particular path. Otherwise, the path is known to be infeasible. Hence, the search for test data is in vain. In these applications the counter example generated in the course of failing model checker runs is the truly desired output, whereas a successful run is of less interest stopping just the search for test data for the path under consideration. This is in contrast and opposite to our application of shrinking the trusted annotation base. In our application, the counter example of a failed model checker run is of little interest. We are interested in successful runs of the model checker allowing us to changing a *trusted annotation* into *verified knowledge*. This opens a new application domain in the field of WCET analysis for model checking. Our preliminary practical results demonstrate the practicality and power of this approach.

References

[1] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007.

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[3] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992 – 1030, 1997.

[4] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*), pages 104 – 115, 1995.

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

[6] Vienna University of Technology and TU Darmstadt. The ForTAS project. web page (`http://www.fortastic.net`). accessed in Apr. 2009.

[7] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, Udine, Italy, December 12 2008.

[8] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis*, Prague, Czech Republic, July 2008.

[9] Bernhard Rieder. *Measurement-Based Timing Analysis of Applications written in ANSI-C*. PhD thesis, Technische Universität Wien, Vienna, Austria, May 2009.

[10] Bernhard Rieder, Ingomar Wenzel, and Peter Puschner. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement-based wcet analysis. In *Proc. 6th Workshop on Intelligent Solutions in Embedded Systems*, Regensburg, Germany, July 2008.

[11] Markus Schordan. Source-To-Source Analysis with SATIrE – an Example Revisited. In *Proceedings of Dagstuhl Seminar 08161: Scalable Program Analysis*. Germany, Dagstuhl, April 2008.

[12] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008.