Deriving the Worst-Case Execution Time Input Values

– Extended version –

Andreas Ermedahl[†‡], **Johan Fredriksson**[‡], **and Jan Gustafsson**[†]
School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
{andreas.ermedahl,johan.fredriksson,jan.gustafsson}@mdh.se

Peter Altenbernd[⋆]
Department of Computer Science
University of Applied Sciences, Darmstadt, Germany
p.altenbernd@fbi.h-da.de

***Abstract*—A Worst-Case Execution Time (WCET) analysis derives upper bounds for execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A major problem with today's WCET analysis approaches is that there is no feedback on the particular values of the input variables that cause the program's WCET. However, this is important information for the real-time system developer.**

We present a novel approach to overcome this problem. In particular, we present a method, based on a combination of input-sensitive static WCET analysis and systematic search over the value space of the input variables, to derive the input value combination that causes the WCET. We also present several different approaches to speed up the search. Our evaluations show that the WCET input values can be relatively quickly derived for many type of programs, even for program with large input value spaces. We also show that the WCET estimates derived using the WCET input values often are much tighter than the WCET estimates derived when all possible input value combinations are taken into account.

I. Introduction

A worst-case execution time (WCET) analysis derives an estimate to the worst possible execution time of a computer program. For the estimate to be applicable in safety-critical, hard real-time systems, where failures to meet deadlines can have catastrophic consequences, the estimate must be *safe*, i.e., greater than or equal to the actual WCET. To be useful, the estimate must also be *tight*, i.e., provide little or no overestimation compared to the actual WCET.

Reliable WCET estimates are a basis for most of the research performed within the real-time research community. They are used to create schedules and to perform schedulability analysis, to determine if performance goals are met for tasks, and to check that interrupts have sufficiently short reaction times.

[†] Supported by the the ALL-TIMES FP7 project, grant no. 2215068, and by the KK-foundation grant 2005/0271.
[‡] Supported by the Swedish Foundation for Strategic Research (SSF), via the strategic research centre PROGRESS.
[⋆] Supported by Deutscher Akademischer Austauschdienst (DAAD).

The WCET analysis problem has been well studied during the last decades. Today, several academic and commercial WCET analysis tools exist, see e.g., [24] for an overview. Many of them are used in industry on a regular basis. Except deriving WCET estimates, the tools often provide other valuable means for understanding the program's worst-case behaviour. This includes, e.g., visualisations of both the execution path and the hardware state that produced the estimate. However, deriving which *particular input value combination that caused the WCET* has, to the best of the authors' knowledge, hardly been addressed before, even though it is very valuable information.

For example, such information can be used for *identifying bottlenecks*, and hence to further optimizing the program. Further, it is valuable for whole-system *stress testing* i.e., identifying the overall system's worst-case behaviour, and for steering *measurement-based timing analysis* methods, to select input value combinations that forces the program to run for long execution times. Moreover, as outlined in Section II, it can be used to produce *tighter WCET estimates*, which allows for better system utilisation.

In general, the problem of determining the worst case input value combination of an arbitrary program is hard. This is because it is possible to create programs for which the WCET input is found only by testing all input value combinations.

However, large parts of software in, e.g., some embedded systems, does not exhibit this type of behavior [5]. If they contain a control flow that is not too complex (e.g., no recursion, unstructured code, deep loop nesting, or complex decision structures), and are run on quite simple hardware (e.g., no caches or speculation features), the WCET can be calculated with high precision by existing static WCET tools, without testing all input value combinations.

Based on these observations, we have developed several novel methods for deriving the WCET input values which are especially efficient for the simpler type of software described above. The concrete contributions of this article are:

- A novel search algorithm to find the WCET input values,

based on a combination of *static WCET analysis* and *systematic search over the value space of input variables*. The best running time is proportional to the 2-logarithm of the size of the input space. This is much faster that exhaustive search, with a running time proportional to the size of the input space.
- An *extreme-value heuristic* together with *other novel ideas* to to speed up the search, allowing the WCET input values in many cases to be derived more quickly than in the basic binary search approach.
- We show how to use our search methods to produce *tighter WCET estimates* compared to WCET estimates derived when all possible input value combinations are taken into account.
- We *evaluate our search methods* on a set of benchmarks and industrial codes, showing that the WCET input values often can be relatively quickly derived, even for program with very large input spaces.

The rest of the article is organised as follows: Section II introduces WCET analysis and presents related work. Section III presents our basic method for deriving WCET input values. Section IV presents approaches for quicker method termination and Section V outlines how to handle overestimations in the static WCET analysis. Section VI presents our analysis results and in Section VII we draw conclusions and outline future work.

II. WCET ANALYSIS AND RELATED WORK

A. Classification of WCET analysis.

A WCET analysis can be categorised as being based on *static analysis*, on *measurements*, or on a combination [24].

A *static WCET analysis* derives WCET estimates without actually running the program. Instead, it takes into account all input value combinations, together with the characteristics of the software and hardware, to derive a safe WCET estimate. The analysis is commonly subdivided into the three phases of *flow analysis*; where bounds on the number of times different instructions can be executed are derived, *low-level analysis*; where bounds on the time different instructions may take to execute are derived, and *calculation*; where a WCET estimate is derived based on the information obtained in the first two phases [7], [24].

Many static WCET analyses are *input-sensitive*, meaning that they are able to take constraints on input variable values into account when calculating a WCET estimate [24]. In general, such analyses should derive more precise WCET estimates than non-input-sensitive, since limitations of inputs may limit the possible paths that can be taken. This is the case, e.g., when the value of an input variable is used to decide the number of times a loop can iterate. In this work, we assume that an input-sensitive static WCET analysis is available, and we use it as a basis for our methods.

It is not always possible to statically deduce the exact timing behaviour of a program with a set of possible inputs, due to the complexity of the control flow and the used hardware. In these cases, static WCET analysis tools make conservative overapproximations in their subanalyses [24], something which may result in non-tight WCET estimates [9]. For example, the WCET may be based on a series on worst-case loop behaviors, each possible for some value of some input variable, but never possible simultaneously for any input value combination. Moreover, different overapproximations are often conservatively combined in the WCET estimate calculation. One example is when a cache analysis conservatively classifies a memory access as a always cache miss even though it may, for a input value combination that gives a large loop bound, result in a cache hit.

Thus, a static input-sensitive WCET analysis tool will sometimes, when run with a single worst-case input value combination, be able to produce a *tighter WCET estimate*, compared to when run with all possible input value combinations. This allows for better overall system utilization and makes it easier for the real-time system designer to produce a schedulable system.

A *measurement-based* WCET analysis executes the program on the hardware for some input value combinations, using some type of time measurement equipment, such as oscilloscopes, logical analysers, or hardware trace mechanisms to derive the timing of the program or parts of the program [2], [23], [25]. Since it is impossible for most programs to test all input value combinations, only a subset of the possible executions are run, hoping that the selected subset will include the WCET-causing input value combination. If not, this may lead to dangerous underestimations of the WCET. Consequently, methods for quickly identifying input value combinations which are candidates to produce the WCET, as the ones presented in this article, should *provide valuable guidance for measurement-based WCET analyses*.

B. Related work

To the author's knowledge the problem of deriving which input values actually cause the WCET has been addressed only by few researchers.

Wenzel et al. [23] use measurements, model checking and genetic algorithms to derive which input values that make a certain code part to execute. The worst timing measured for different code parts are then combined to an overall program WCET estimate. In contrast, our approach does not use measurements and is able to derive the overall program WCET input values.

Fredriksson et al. [10], [11] derive a parametrical WCET formula for a software component, based on an automatized partitioning of the component's input value space and multiple runs by an input-sensitive WCET analysis tool. Staschulat et al. [18] derives a similar context-dependant partitioning of the execution time behaviour of software modules from graphical models of the system. Our approach has some similarities with, and is reusing part of Fredriksson's work. However, since we derive a single WCET input value combination and not a parametrical WCET formula, the algorithms and implementations differs a lot.

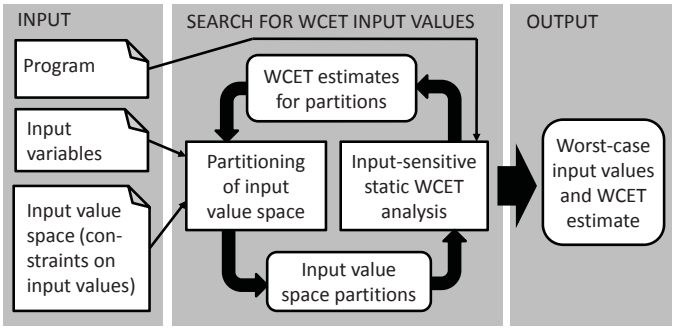

Fig. 1. WCET input value analysis

```
find_WCET_input_values(prog, vs) :=
  vsp <- vs;
  prio_queue <- empty;
  WHILE cardinality(vsp) > 1 DO
    vsp_set <- partition(vsp);
    FOREACH vsp' in vsp_set DO
      wcet <- calc_wcet(prog, vsp');
      prio_queue.insert(wcet, vsp');
    END FOREACH
    vsp <- prio_queue.extract_max();
  END WHILE
  RETURN vsp.pop_vc();
```

Fig. 2. WCET input search algorithm

III. Deriving the WCET input values

Our method consists of a search algorithm (see Fig. 1) with a combination of systematic search over partitions of the *value space of the input variables* and an *input-sensitive static WCET analysis*. The value space of the input variables is all concrete combinations of the input variable's values of the program.

The possible values of the input variables are restricted by the data type in the program. They can often be further restricted, e.g., by natural physical value constraints. For example, a variable *speed* declared as an 8-bit unsigned integer can hold integer values between 0 and 255. Assuming that *speed* holds the input of a vehicle speed sensor, and the vehicle can't go faster than 200 km/h, then *speed* can be further constrained by $speed \leq 200$. These additional restrictions are often declared by user annotations.

The algorithm systematically divides this input value space into smaller *input value space partitions*, each with a subset of the input value space. Consider a program with three integer input variables i, j, and k, with the input value constraints $0 \leq i \leq 1$, $0 \leq j \leq 0$, and $0 \leq k \leq 1$. The input value space vs is then $\{\langle i \leftarrow 0, j \leftarrow 0, k \leftarrow 0\rangle, \langle i \leftarrow 1, j \leftarrow 0, k \leftarrow 0\rangle, \langle i \leftarrow 0, j \leftarrow 0, k \leftarrow 1\rangle, \langle i \leftarrow 0, j \leftarrow 0, k \leftarrow 1\rangle\}$ The number of input value combinations is $|vs| = 4$, and the number of non-empty value space partitions are $2^{|vs|} - 1 = 15$.

A. Algorithm description

The basic search algorithm is given in Fig. 2. It works by iteratively calculating WCET estimates for different partitions of the program's input value space. In each iteration the partition with the largest WCET estimate is selected and divided into two or more smaller partitions, for which WCET calculations are made. The process continues until the selected partition holds only one input value combination, which is then returned. Under the algorithm prerequisites given below, this is a WCET input value combination[1]. The analysis can be stopped at any time to return the largest WCET currently selected. This is a safe value, and possibly tighter than the value derived by running the static WCET analysis with the original input value space.

[1]Note that there sometimes may be several input value combinations which results in the WCET.

The arguments to the algorithm are `prog`, the program under analysis, and `vs`, a representation of the input variable's value space. A value space partition, `vsp`, holds a subset of `vs`. The function `cardinality(vsp)` returns the number of input value combinations held by the argument `vsp`. The function `partition(vsp)` creates two or more disjunct partitions from the argument `vsp`.

The `prio_queue` is a priority queue where inserted partitions are indexed upon their corresponding calculated WCET estimates. The function `extract_max()` removes and returns the partition with the largest WCET estimate in the queue.

B. Algorithm prerequisites

The algorithm puts some demands on the WCET analysis used. Let $calc\_wcet(vsp)$ be the WCET estimate calculated by a WCET analysis for a partition vsp. Further, let $meas\_time(vc)$ be the measured execution time for an input value combination vc. For the algorithm to work correctly the following assumptions should hold:

(1) The WCET calculation must never underestimate the WCET, i.e.,

$$max_{vc \in vsp}\ meas\_time(vc) \leq calc\_wcet(vsp)$$

must be true for any vsp.

(2) A WCET calculation run with a single-valued input value partition should produce a WCET estimate equal to the time for running the program with these inputs. I.e., if

$$vc \in vsp \text{ and } |vsp| = 1$$

then the following holds:

$$calc\_wcet(vsp) = meas\_time(vc)$$

(3) For any two input value space partitions vsp_1 and vsp_2 such that $vsp_1 \subseteq vsp_2$ then

$$calc\_wcet(vsp_1) \leq calc\_wcet(vsp_2)$$

should hold.

We claim that assumptions (1) and (3) are sound and valid for most type of today's input-sensitive WCET analysis tools. However, assumption (2) might not always be true. In Section V we outline how to modify our algorithm to overcome this.

```
partition(vsp) :=
  vsp_set <- empty;
  v <- select_var_with_many_values(vsp);
  l..u <- get_assigned_range(vsp, v);
  m <- l + ((u - l) / 2;
  vsp' <- new_vsp(vsp, v, l..m);
  vsp'' <- new_vsp(vsp, v, m+1..u);
  vsp_set.insert(vsp');
  vsp_set.insert(vsp'');
  RETURN vsp_set;
```

Fig. 3. Binary search using ranges

C. Binary search using ranges

A compact and efficient way of representing a set of input value combinations is to assign a $l..u$ range to each input variable v. Then $v \leftarrow l..u$ means that all input values that can be assigned to v are $\geq l$ and $\leq u$. Assuming that v is an integer variable then the number of values held by the $l..u$ range represention is $u - l + 1$. Moreover, assuming that we have n input variables each assigned a range, then the total number of concrete input value combinations is the product of all the range sizes.

Based on this range representation we can construct a type of binary search over the input value space as outlined in function `partition` in Fig. 3. From the argument `vsp`, which represent two or more concrete input value combinations, we first select a variable `v` with a multi-valued range, i.e., $u - l \geq 1$. `v`'s range is then extracted and used to create two new disjunct value space partition, each with a size about half of the argument `vsp`. The function `new_vsp(vsp,v,r)` creates a new value space partition from the argument `vsp` by replacing the range assigned to `v` with the new range `r`. The created partitions are inserted in the `vsp_set` set, which is returned.

D. An illustrative example

Fig. 4 illustrates how the algorithm works. The program has three input variables `i`, `j`, and `k` and has been given the initial value space of $\langle \mathtt{i} \leftarrow 0..15, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ which corresponds to $16 * 1 * 2 = 32$ concrete input value combinations. Variable `i` is selected first for range division. This produces two new partitions for which WCET calculations are made, from which the $\langle \mathtt{i} \leftarrow 0..7, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ partition gives the largest WCET estimate and the analysis therefore continues with this partition. This time `i`'s range is subdivided into $0..3$ and $4..7$, both producing partitions for which WCET calculations are made.

The division of `i`'s range continues until the value of `i` which produces the largest WCET estimate when $\mathtt{j} \leftarrow 0$ and $\mathtt{k} \leftarrow 0..1$ has been found. Since `j` only can hold a single value, the next variable selected is `k`. The division of `k`'s range produces two partitions, for which $\langle \mathtt{i} \leftarrow 5, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0 \rangle$ gives the largest WCET. There are no other partitions in the priority queue with a larger WCET estimate, so the iteration stops and this value combination is returned.

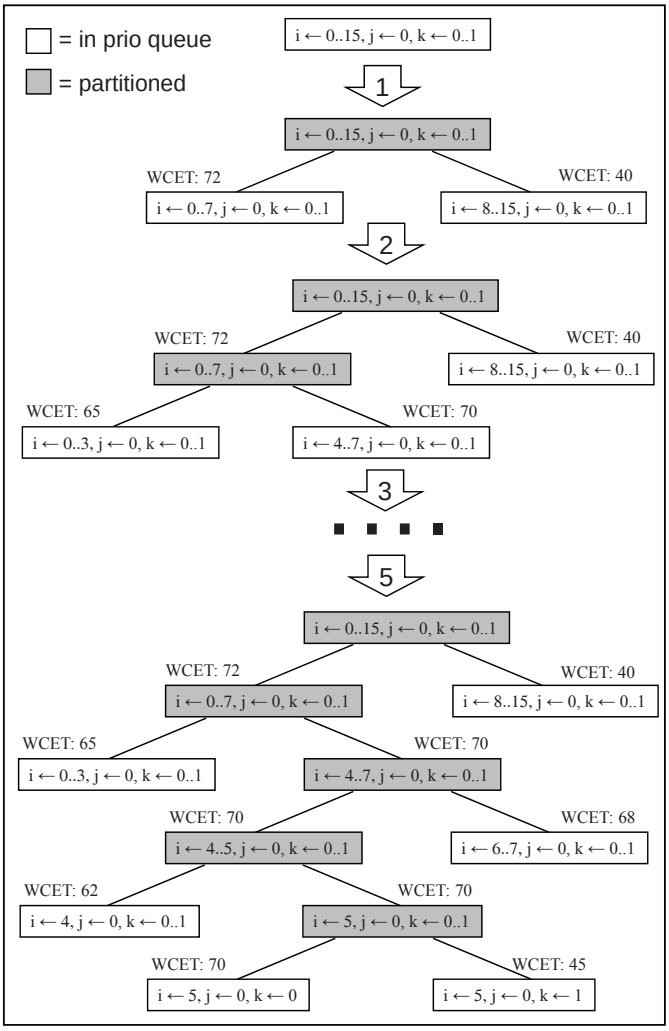

Fig. 4. Example of basic algorithm run

It should be noted that the range representation does not allow constraints between the values of two or more input variables to be expressed. Allowing such constraints makes the division of the input value space into smaller partitions somewhat more complicated. However, it is fully possible to extend our WCET input search methods to handle such constraints.

A strategy for dividing the input space must be devised for each data type and representation. In the present implementation, we only have support for integers represented as intervals. However, we have the following ideas for other data types:

- Floating point input variables, which can represent vast amount of values, can be represented by intervals. The smallest input value space will be defined as an interval of suitable size.
- Pointer values can be represented as a combination of pointer sets and offset intervals (see [13]), and dividing the input space can be done using subsets and subintervals.

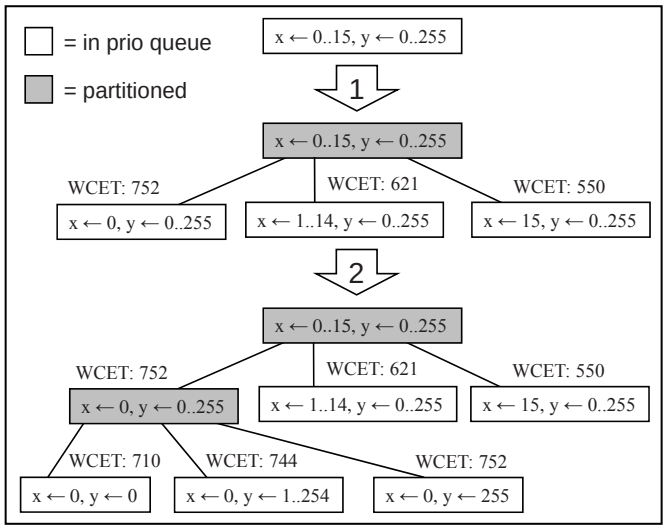

Fig. 5. Example of extreme value heuristic

E. Algorithm complexity and back-tracking

Our algorithm will, when using binary search and ranges as outlined in Fig. 3, have a best-case behaviour (in the number of WCET analysis runs) of $\Omega(2 * log_2|vs|) = \Omega(log_2|vs|)$, where vs is the input value space. This is because in each step the size of the currently selected partition is divided by two. Unfortunately, due to overapproximations made in the WCET analysis, this behaviour is not always the case. Sometimes both partitions originating from the selected partition gets a WCET estimate smaller than a WCET estimate found in the priority queue. The analysis then has to *back-track* and continue with the partition with the higher WCET. Assume that the $\langle \mathtt{i} \leftarrow 5, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0 \rangle$ partition in Fig. 4 gave a WCET estimate of 65 instead of 70. Then the analysis must continue with $\langle \mathtt{i} \leftarrow 6..7, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ with a WCET estimate of 68 instead of terminating, since $68 > 65$.

In the worst-case, this type of back-tracking causes a WCET calculation to be made for each concrete input value combination plus a WCET calculation for each node in the binary search tree. Thus, the algorithm has a worst-case behaviour (in the number of WCET analysis runs) of $O(2 * |vs|) = O(|vs|)$.

IV. Approaches for faster termination

A potential problem with the binary search algorithm is that many WCET calculations might be needed before the input WCET values can be determined, especially when $|vs|$ is large. This section outlines some approaches for faster algorithm termination.

A. Extreme-value heuristic

A general observation is that it is often either the smallest or the largest value in an input variable's value domain will be the value that gives the WCET. This is especially true if the outcome of loop conditions are dependent on this input variable.

Our *extreme-value search heuristic* originates from this observation. It modifies the algorithm in Fig. 3 as follows: whenever a variable v is selected for the *first time* its $l..u$ range will be divided into *three* new ranges (if possible): $l..l$, $l+1..u-1$, and $u..u$.

Fig. 5 illustrates how our extreme-value heuristic works. The program has two integer input variables `x` and `y`, with a given input value range of $0..15$ and $0..255$ respectively. Variable `x` is first selected for input range division. This produces three new partitions for which WCET estimates are calculated. The $\langle \mathtt{x} \leftarrow 0, \mathtt{y} \leftarrow 0..255 \rangle$ partition gets the largest WCET estimate and the analysis continues with a division of `y`'s range. This produces three new partitions for which WCET estimates are calculated. The $\langle \mathtt{x} \leftarrow 0, \mathtt{y} \leftarrow 255 \rangle$ partition gives a WCET estimate which is larger than all other partitions in the priority queue. Thus, the WCET is caused when `x` and `y` were assigned its minimum and maximum input values, respectively.

Our extreme-value heuristic will have a best-case behaviour (in the number of WCET analysis runs) of $\Omega(3 * n) = \Omega(n)$ where n is the number of multi-valued input variables. The worst-case behaviour will, similar for our binary search algorithm, be $O(|vs|)$.

B. Proceeding depth first

When we have equal WCET estimates for two partitions, we should proceed with the one which has processed furthest down the tree. This is because we are closer to termination in this node, assuming that no reduction is found. This can be made by keeping track on the number of processing steps, i.e., the level in the tree, for each partition.

C. Postponing WCET calculations

Another optimization is to postpone a WCET calculation for a value space partition vsp_j if there is another partition vsp_k which originates from the same partition, say vsp_i, and $calc\_wcet(vsp_i)$ equals $calc\_wcet(vsp_k)$. If we find no reduction further down the search tree, we do not have to do a WCET calculation for vsp_j. This, however, requires an additional priority queue holding non-processed items. As an illustration, consider the processing of the $\langle \mathtt{i} \leftarrow 4..7, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ partition in Fig. 4. Since $\langle \mathtt{i} \leftarrow 4..5, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ also got 70 as WCET, we can postpone the WCET calculation of $\langle \mathtt{i} \leftarrow 6..7, \mathtt{j} \leftarrow 0, \mathtt{k} \leftarrow 0..1 \rangle$ until we get a smaller WCET.

D. Ordering the processing of variables

If some variable's input values have larger effect on the calculated WCET estimate than others, these variables should be processed first. The reason is that the number of nodes to process in the total search tree will be smaller in that case. For example, assume that we have two variables v_1 and v_2, and it is only the partitioning of the first that reduces the WCET estimate. If we partition on v_1 first, there will be no back-tracking from the partitioning of the v_2. If we take v_2 first, however, we will have to back-track to each node in the search

tree of v_1. This idea is similar to the First Fail Principle with Constraint (FFC) heuristic in constraint programming, where the variable which is most constrained is selected first [14]. Such ordering of variables could either be made statically, before the algorithm is run, or dynamically during run-time.

E. User interaction

Another option for improving the overall analysis time is to let the user provide an input value space partition in which he/she believes the worst-case input value combination is to be found. For example, in Fig. 4 the user might believe that the worst-case is when `k = 1` and `0 ≤ i ≤ 6`. The analysis can then be started with an initial set of partitions according to the user's assumptions, e.g., $\langle i \leftarrow 0..6, j \leftarrow 0, k \leftarrow 1 \rangle$, $\langle i \leftarrow 0..6, j \leftarrow 0, k \leftarrow 0 \rangle$, and $\langle i \leftarrow 7..16, j \leftarrow 0, k \leftarrow 0..1 \rangle$, where the first partition corresponds to the user provided assumption.

F. Using BCET calculations

If the used WCET tool also supports a safe *best-case execution time* (BCET) calculation (i.e., provides a lower bound of the actual BCET), the overall analysis time can sometimes be shortened. When $calc\_bcet(vsp) = calc\_wcet(vsp)$ holds for a selected value space partition vsp, all input value combinations in vsp have exactly the same timing. Thus, the analysis can then stop and any input value combination in vsp can be returned. The benefit of using this optimisation depends on the cost of doing BCET calculation and how likely it is that the BCET estimate becomes equal to the WCET estimate.

G. Using measurements

Another optimization is to initially make a few program runs on some measurement equipment for some input value combinations. The worst measured timing t is then a lower estimate of the WCET. Then, all partitions with a WCET estimate $\leq t$ can be pruned away from the priority queue, since they are guaranteed not to include the WCET input value combination. When $calc\_wcet(vsp) = t$ holds for a selected partition, the analysis can stop and the input value combination which generated the measured timing t can instead be returned.

V. Handling overestimations for single input value combinations

The methods outlined in Sections III and IV work under the assumption that, when run with a single value space partition ($|vsp| = 1$), the static WCET analysis should produce a value equal to measured value for the same input combination, i.e., if $vc \in vsp$ and $|vsp| = 1$ then

$$calc\_wcet(vsp) = meas\_time(vc)$$

should hold.

Unfortunately, this is not true for all type of static WCET analyses, mainly due to overapproximations made in the flow- and low-level analysis. Consequently, we can then only guarantee that for single partitions the calculated WCET estimate will be greater than or equal to the measured time, i.e., if $vc \in vsp$ and $|vsp| = 1$ then

$$calc\_wcet(vsp) \geq meas\_time(vc)$$

holds. This means that we might have two disjunct partitions vsp_1 and vsp_2, each holding a single input value combination, i.e., $vc_1 \in vsp_1$, $|vsp_1| = 1$, $vc_2 \in vsp_2$, $|vsp_2| = 1$, and that $calc\_wcet(vsp_1) \geq calc\_wcet(vsp_2)$ and $meas\_time(vc_1) < meas\_time(vc_2)$ both are simultaneously true. Thus, vc_1 might be wrongly reported as the WCET input value combination.

This problematic situation can be handled by complementing our method with some type of measurement equipment. When a partition only holds a single input value combination, the program is run with this combination on the measurement equipment, instead of using the static WCET analysis. Thus, the

```
wcet <- calc_wcet(prog, vsp');
```

line in the Fig. 2 algorithm gets replaced with:

```
IF cardinality(vsp') == 1 THEN
  wcet <- meas_time(prog, vsp'.pop_vc());
ELSE
  wcet <- calc_wcet(prog, vsp');
```

In general, if the cost for doing static WCET analysis is much higher than the cost for doing measurements, it might be better to do exhaustive measurements for a value space partition when the number of included input value combinations is small enough.

It should however be noted that accurate timing measurement in itself is a challenging problem. On complex hardware architectures there are often many possible hardware states in which a program might start its execution, and many of these states might result in a different timing even though the input variable's values are fixed [16]. We do not consider setting up the intial worst-case hardware state for a measurement in this article. Instead, we refer to e.g., [4], [15], [24] for approaches to handle this problem.

VI. Experimental evaluations

A. The SWEET tool

The outlined methods have been implemented in our WCET analysis tool SWEET (SWEdish Execution time Tool) [7], [13]. SWEET performs flow analysis on an intermediate code representation (NIC) emitted by a research compiler [26]. The NIC code has a one-to-one mapping to the control-flow graph structure of the compiler generated object code. Flow analysis results derived on NIC can therefore be directly mapped to entities in the object code.

SWEET includes a flow analysis called *Abstract Execution* (AE) [13], which is a form of symbolic execution based on Abstract Interpretation (AI) [3], [20]. The AI is used to derive safe bounds on the possible values of variables at different program points. The AE is capable of deriving various type of flow information and can handle almost full ANSI-C, including pointers, bit operations, and aggregate data structures

Program	Description	LOC	Funcs	Loops	Conds	Ann	IAnn	|VS|
crc	Cyclic redundancy check computation.	128	3	3	9	4	4	16
edn	Finite impulse response filter calculations.	285	9	12	12	3	3	44
jcomplex	Nested loop program.	64	2	2	5	2	2	512
lcdnum	Read ten values, output half to LCD.	64	2	1	26	2	2	160
ns	Search in a multi-dimensional array.	535	2	5	5	1	1	2
nsichneu	Simulates an extended Petri net.	4253	1	1	625	6	6	54
task1	Industrial task developed by Volvo CE	55	1	0	4	6	3	36
task2	Industrial task developed by Volvo CE	56	1	0	4	11	6	$3.24 * 10^8$
task3	Industrial task developed by Volvo CE	58	3	0	4	12	7	$9.38 * 10^{10}$
task4	Industrial task developed by Volvo CE	72	1	0	13	20	13	$4.24 * 10^{17}$
task5	Industrial task developed by Volvo CE	86	2	0	9	11	6	$4.71 * 10^{10}$
task6	Industrial task developed by Volvo CE	43	5	1	10	12	7	$2.47 * 10^{13}$
task7	Industrial task developed by Volvo CE	123	4	1	32	24	15	$2.10 * 10^{19}$
task8	Industrial task developed by Volvo CE	119	5	0	17	12	4	$3.48 * 10^{11}$
task9	Industrial task developed by Volvo CE	49	5	0	9	4	4	$3.24 * 10^7$
task10	Industrial task developed by Volvo CE	195	3	0	40	24	9	$1.00 * 10^{12}$

TABLE I
INPUT-SENSITIVE BENCHMARKS USED

such as structs and arrays [13], [19]. AE has several options for trading precision and analysis time and can take constraints on input variable's values into account (thus being an input-sensitive flow analysis).

Constraints on input values are given in SWEET's annotation language. Numeric variables are constrained by intervals. Pointer constraints are sets of abstract addresses, each representing a range of NIC addresses. We can also give annotations on the content of (fields of) aggregate data structures, such as structs and arrays. The annotations can constrain variable values in specific program points. Normally, inputs are constrained at the program entry point.

The low-level analysis of SWEET [6] supports the NECV850E and ARM9 processors, each including a pipeline but no cache. The low-level analysis is not input-sensitive, i.e., the upper timing bounds derived for instructions, will not be affected by constraints on the program's input values. Three calculation methods are supported: a path-based method, a global IPET method, and a hybrid clustered method [7], [8].

B. Evaluation issues

We have implemented the binary- and the extreme-value search, both using ranges for representing variable's values (see Section III and Section IV). In all our evaluations we used the depth-first and postponing optimizations. However, we have not used any user provided annotations to make an initial partition of the input value space, nor have we used BCET calculations or measurements optimizations to reduce the search time (see Section IV).

We have not complemented our evaluations with any measurement equipment, as described in Section V. However, similar to a time-accurate simulator, SWEET's low-level analysis include the possibility get the timing of instruction traces. Thus, for each worst-case input value combination derived by our search methods, we also derived a corresponding instruction trace. The timing for the trace, derived using SWEET's instruction trace simulation facility, was then compared to the calculated WCET estimate.

For all our experiments we used a less precise version of AE[2], optimising the WCET calculations for speed. This means that we may have encountered more back-tracking, compared to when using a more precise AE. We used the ARM9 processor model in the low-level analysis, and the clustered calculation method.

We have used programs from the Mälardalen WCET Benchmark suite [22], together with some task codes [1], [17] provided, together with associated input value annotations, by our industrial partner [21], to evaluate our methods[3].

Tab. I gives some details of the benchmark used. **LOC** gives lines of C code. **Ann** gives number of input annotations, while **IAnn** gives number of integer input annotations. The integer annotations where provided using ranges, thus suitable for partitioning. The annotations for the industrial tasks also included some pointer annotations. These annotations held, however, only one pointer value each, and did therefore not need any partitioning. Many of the task code annotations referred to 8-, 16- or 32-bit fields of aggregate data structures. No floating point annotations where provided. **|VS|** gives the size of the input value space, derived as the product of all the value sizes of the integer input variables annotations.

All evaluations where run in Cygwin and Windows XP on a Dell Precision M4300, with an Intel Core2 T7500 CPU, running at 2.2GHz and with 3.5GB of RAM. Since a lot of tool communication were needed, we used Bash shell programming and Cygwin built-in shell commands to implement our algorithms.

[2]We used full merging and generated only upper loop bounds and global upper bounds on the number of times different nodes in the program can be executed. For details on AE's analysis options, please see: [12], [13].

[3]Compared to [1] only tasks 1–10 were included in our experiments. Tasks 11–13 were excluded due to incomplete input annotations files.

Program		Binary						Extreme								
	OrgW	WC	BT	MnT	MxT	WT	AT	WC	BT	MnT	MxT	WT	AT	FinW	RedW	SimW
crc	83275	5	0	10	12	56	105	5	0	10	11	54	91	83275	0%	83275
edn	9241	6	0	20	27	134	217	4	0	20	21	82	121	9241	0%	9241
jcomplex	4557	383	279	1	8	1404	3496	391	288	2	8	1447	3620	561	87.7%	561
lcdnum	858	17	9	2	5	55	163	5	2	2	4	13	40	858	0%	858
ns	1603	2	0	4	31	35	53	2	0	4	28	32	49	1603	0%	1603
nsichneu	16242	12	7	91	101	1153	1238	12	7	93	103	1177	1296	16242	0%	16242
task1	82	7	1	12	17	98	163	5	1	12	17	69	116	82	0%	82
task2	60	30	2	1	7	93	365	11	5	1	5	33	129	60	0%	60
task3	156	39	4	36	53	1462	1938	23	15	36	52	868	1124	156	0%	156
task4	576	77	20	15	25	1673	2342	55	43	15	23	1169	1511	514	10.8%	514
task5	378	-	-	-	-	-	-	-	-	-	-	-	-	376	0.5%	-
task6	237	57	15	3	6	241	803	40	32	3	6	173	446	226	4.6%	226
task7	756	418	359	6	15	3361	6983	742	726	6	12	5894	11476	639	15.5%	639
task8	256	265	226	5	10	1592	3366	274	269	5	16	1688	3473	229	10.5%	229
task9	367	32	7	3	8	152	471	18	13	3	6	80	207	295	19.6%	295
task10	446	252	213	19	46	7381	9065	243	234	22	38	7143	8964	403	9.6%	403

TABLE II
ANALYSIS RESULTS FOR INPUT-SENSITIVE BENCHMARKS

C. Obtained results

Tab. II shows analysis results. **Binary** gives results for the basic input derivation method, while **Extreme** gives results for our extreme value search method. **OrgW** gives the original WCET estimate (in clockcycles) derived by SWEET with all input value combinations. **WC** gives the number of WCET calculations performed, and **BT** gives the number of WCET calculations due to back-tracking. We here (pessimistically) define a back-track as any WCET calculation performed outside the algorithm's best-case path. Thus, for the binary search the amount of back-tracking is

$$WC - ((\Sigma_{v \in IAnn} \lceil log(|v|) \rceil) + 1)$$

while for the extreme heuristic it is

$$WC - ((\Sigma_{v \in IAnn} 1) + 1)$$

The $+1$ term comes from the fact that we, in our evaluations, made an additional WCET calculation for the original input value space.

MnT and **MxT** gives the minimum and maximum analysis time, in seconds, used for any of the WCET calculations. **WT** gives the total time, in seconds, spent in WCET calculations, while **AT** gives the total analysis time in seconds (including file processing and shell-command runs). **FinW** gives the final WCET estimate obtained for the derived worst-case input value combination (for all terminating analyses we got the same **FinW** for both binary and extreme) and **RedW** gives the WCET reduction compared to **OrgW** in percent. **SimW** gives the timing, in clockcycles, obtained using SWEET's trace simulation facility for the derived worst-case input values, as described above.

We conclude that our methods are able to derive a WCET input combination for most of the analyzed programs. Further, the number of WCET calculations needed grow with the size of the input value space.

All programs experience varying WCET calculation time. In general, when the input value size decreases, the time for performing the WCET calculation also decreases. Thus, the first analysis generally consumes most time, while subsequent analyses are faster.

We see that the **Binary** and **Extreme** heuristics are highly dependent on the input value(s) that gives the worst-case behaviour. For most of the programs it is beneficial to use the extreme heuristic, but not for all (`jcomplex`, `task7`, and `task8` being notable exceptions).

D. Categories of analysis runs

When investigating our analysis traces we could distinguish three major categories of analysis runs. For the first category, examplified in Fig. 6(a), the **FinW** became equal to the **OrgW**. This means that there exists at least one path through the tree from the root to a leaf where all the branch-nodes holds the same WCET. Thus, the WCET input value can be derived with no, or rather small amount of back-tracking. All benchmarks with **RedW** equal to 0% belongs to this category, such as `lcdnum`, and `task1`, `task2`, and `task3`.

For the second category of analysis runs, examplified in Fig. 6(b), we get a reduction of WCET compared to the **OrgW**. Thus, overapproximations in our flow analysis, due to a large input value space, made the initial **OrgW** pessimistic. Moreover, there is a path through the tree from the root to the worst-case leaf which can be taken with rather little amount of back-tracking. Thus, even though nodes in this path get smaller WCETs when the input value size decreases, this is "compensated" by the fact that all other tree nodes have got even smaller WCETs. `task9` belongs to this category.

For the third category of analysis runs, examplified in Fig. 6(c), we experienced a large amount of back-tracking. As for category 2, we generate a overly pessimistic WCET for the initial input value space. Moreover, we also get a reduction in WCET when propagating further down the tree that is larger

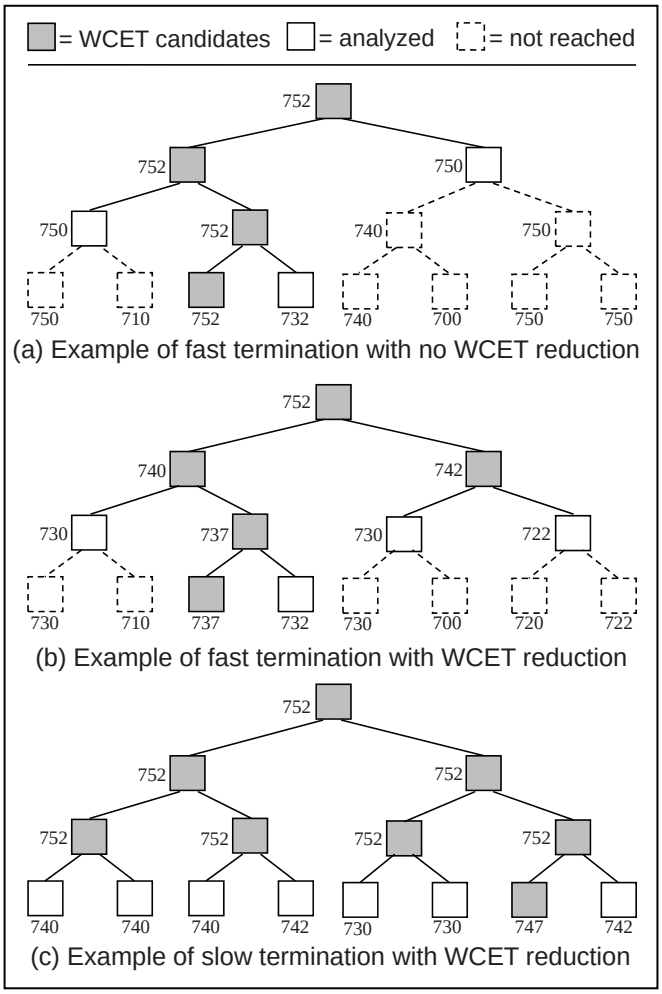

Fig. 6. Types of analysis runs encountered

than the WCET reduction for branches higher up in the tree. `jcomplex`, `task4`, `task5`, `task6`, `task7`, `task8`, and `task10` all belong to this category. When running these programs with their initially given annotations, only `jcomplex`'s analysis terminated within reasonable time[4]. By changing the processing order, making variables with smaller input domains get processed before variables with large input domains, we got the analysis results in Tab. II for `task7` and `task8`. For `task4`, `task6`, and `task10` we had to investigate printouts of previous analysis runs, and put the variables, that when partitioned gave the largest reduction, first in the processing order. Thus, we conclude, that the order in which variables are processed may have a large impact on the analysis time.

For `task5` we could not get the analyses to terminate within reasonable time, even by changing the processing order. The WCET reported for this task is therefore the largest WCET value stored in the priority queue when the analyses were halted.

The huge amount of back-tracking for `jcomplex` can be explained by the fact that `jcomplex` has a very complex input-dependant execution behaviour. Basically, each concrete value combination of the two input variables results in a different execution time. Thus, for such type of (rather artificial) programs our methods may experience a lot of back-tracking.

To summarize: for benchmarks belonging to category 1 our methods are able to rather quickly derive the WCET input values. For benchmarks belonging to category 2 and 3, it might take longer time to derive the WCET input values. For the third category, we could use our methods to get a tighter WCET, even without having to wait for the analysis to terminate. Moreover, for some benchmarks we were able to manually "convert" the program from the third to the second category. The WCET reduction for benchmarks belonging to category 2 and 3 for which our analyses terminated ranged from 4.6% (`task6`) to 87.7% (`jcomplex`).

VII. Conclusions and future work

This paper has introduced several novel methods to derive the concrete input values that cause the WCET of a program, which – though important – has hardly been addressed before. Our evaluations show that for most of our benchmarks the WCET input variables can be relatively quickly derived, even for those with very large input value space. We have also shown that our methods can be used to get tighter WCET estimates, even without having to wait for the analysis to terminate.

For future work we plan to investigate classical search methods, such as simulated annealing, to find a good initial partition of the input value space. Moreover, we would like to investigate methods originating from the constraint-programming community to derive good ordering of variables for our search algorithms. We would also like to investigate the causes of overapproximation in our WCET analysis. With higher precision in our initial WCET estimates the amount of backtracking should decrease. We would also like to investigate the usage of our methods to quickly narrow down the search space for measurement-based WCET analysis approaches.

VIII. Acknowledgments

The authors would like to thank Volvo CE [21] for letting us use their industrial task codes in our experimental evaluations.

References

[1] Dani Barkah, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Christer Sandberg. Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In *Proc. 20^{th} Euromicro Conference of Real-Time Systems*, July 2008.

[2] Guillem Bernat, Antoine Colin, and Stefan Petters. pWCET: a tool for probabilistic worst-case execution time analysis. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, 2003.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4^{th} ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[4] Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In *Proc. 5^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2005)*, 2005.

[4] We used 4 hours as an upper time limit for our analyses.

[5] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[6] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala, Sweden, April 2002. ISBN 91-554-5228-0.

[7] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[8] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Transaction on Computers*, 54(9):1104–1122, September 2005.

[9] Christian Ferdinand and Reinhold Heckmann. Worst-case execution time – a tool provider's perspective. In *11^{th} IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC2008)*, pages 340–345, 2008.

[10] Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, and Mikael Nolin. Clustering worst-case execution times for software components. In *Proc. 7^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2007)*, Pisa, Italy, July 2007.

[11] Johan Fredriksson, Thomas Nolte, Mikael Nolin, and Heinz Schmidt. Contract-based reusable worst-case execution time estimate. In *Proc. 14^{th} International Conference on Real-Time Computing Systems and Applications (RTCSA'07)*, August 2007.

[12] Jan Gustafsson and Andreas Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, July 2008.

[13] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS'06)*, December 2006.

[14] Kim. Marriott and P. J. Stuckey. *Programming with constraints : an introduction / Kim Marriott and Peter J. Stuckey*. MIT Press, Cambridge, Mass. :, 1998.

[15] Stefan Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real-Time Computer Systems, Technische Universiät München, Germany, September 2002.

[16] Jan Reineke and Daniel Grund. Sensitivity analysis of cache replacement policies. Technical Report 36, Automatic Verification and Analysis of Complex Systems (AVACS) - project, March 2008. ISSN: 1860-9821.

[17] Daniel Sehlberg, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Steffen Wiegratz. Static WCET analysis of real-time task-oriented code in vehicle control systems. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *Proc. 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, Paphos, Cyprus, November 2006.

[18] Jan Staschulat, Rolf Ernst, Andreas Schulze, and Fabian Wolf. Context sensitive performance analysis of automotive applications. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*, pages 165–170, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Lili Tan. The worst case execution time tool challenge 2006. Technical report, AbsInt GmbH, 2006.

[20] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[21] Volvo CE (construction equipment) homepage, 2008. `www.volvo.com/constructionequipment`.

[22] Mälardalen WCET benchmarks homepage, 2006. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[23] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in Europe (DATE'05)*, volume 1, pages 606–611, March 2005.

[24] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[25] F. Wolf, J. Kruse, and R. Ernst. Timing and power measurement in static software analysis. *Microelectronics Journal, Special Issue on Design, Modeling and Simulation in Microelectronics and MEMS*, 6:91–100, 2002.

[26] The Whole-Program Optimization project homepage, 2001. `www.astec.uu.se/etapp3/`.