Practical Experiences of Applying Source-Level WCET Flow Analysis on Industrial Code

Björn Lisper,[1] Andreas Ermedahl,[1] Dietmar Schreiner,[2] Jens Knoop,[2] Peter Gliwa[3]

[1] School of Innovation, Design, and Engineering, Mälardalen University, SE-721 23 Västerås, Sweden
[2] Institute of Computer Languages, Vienna University of Technology, A-1040 Vienna, Austria
[3] GLIWA GmbH embedded systems, Dollmann Str. 4, D-81541 München, Germany

Abstract. Code-level timing analysis, such as Worst-Case Execution Time (WCET) analysis, takes place at the binary level. However, much information that is important for the analysis, such as constraints on possible program flows, are easier to derive at the source code level since this code contains much more information. Therefore, different source-level analyses can provide valuable support for timing analysis However, source-level analysis is not always smoothly applicable in industrial projects. In this paper we report on the experiences of applying source-level analysis to industrial code in the ALL-TIMES FP7 project: the promises, the pitfalls, and the workarounds that were developed. We also discuss various approaches to how the difficulties that were encountered can be tackled.

This work was supported by the EU FP7 project ALL-TIMES (Integrating European Timing Analysis Technology, grant agreement no. 215068).

1 Introduction

Today, over 99 percent of all processors are *embedded* in products or systems. Many of these embedded systems are *real-time systems*, i.e., computer systems that must react within a certain time to events in their environment. Failure of such systems to meet their timing constraints could endanger human life and cause substantial economic losses. Thus, improved methods for timing verification are of paramount importance.

The purpose of *timing analysis* is to find out whether the timing constraints of a system are met. It is usually divided into *system-level analysis*, dealing with the combined execution of different programs ("tasks"), and *code-level analysis* dealing with the timing of individual tasks. The most important entity to estimate on code level is the *worst-case execution time* (WCET) of a task. WCET estimates are needed for many system-level analyses. For these to yield reliable

results, the WCET estimates must be *safe* meaning that they never will underestimate the true WCETs. Unsafe WCET estimates can also be useful, in situations like early design phases where they can help dimensioning the system.

WCET analysis assumes that the code is running to completion in isolation, without being interrupted. The two main kinds of analysis are *measurements* and *static analysis*. In general, measurements cannot give safe estimates and are thus suitable for less time-critical software. For time-critical software, where the WCET must be safely estimated, static analysis is preferable. *Hybrid analysis* combines elements of static and measurement-based analysis.

Static WCET analysis derives a WCET estimate analysing mathematical models of the code and the hardware. If the models are correct, then a static WCET analysis will always derive a safe WCET estimate. The static analysis is usually divided into three phases: a *flow analysis* where information about the possible program execution paths is derived, a *low-level analysis* where the execution times for atomic parts of the code, like basic blocks, are decided from a performance model of the target architecture, and a final *calculation* where the flow and timing information is combined to derive a WCET estimate.

The flow analysis must handle a possibly very large number of paths. Thus, approximations are needed. If the approximations are safe (representations include at least the possible paths) then a WCET analysis is still safe, although it might become pessimistic. Flow analysis research has mostly focused on *loop bound* analysis [11], since upper bounds on the number of loop iterations must be known in order to derive finite WCET estimates. Automatic methods to find these exist, but often some loop bounds must still be bounded manually. Flow analysis can also identify other types of program flow constraints like *infeasible paths* [10]. constraining the number of times different program parts can be executed together.

To be accurate, flow analysis should be performed on the binary level, for the code actually being executed. However, much information is often missing in this code. Therefore, flow analysis is sometimes performed on source code. The advantage is that there is much more high-level information in the code that can help obtaining a precise flow analysis. On the backside, the program flows in source and binary are sometimes not exactly the same, especially if the compiler applies heavy optimizations. Program flow constraints derived from the source code can then be unsafe for the program flows of the binary. But there are still many situations where this is tolerable, for instance when making rough timing estimates in early design phases. Also, the technique is applicable for safety-critical applications where safety standards demand that compiler optimizations are turned off.

In this paper, we report on some practical experiences of source-level flow analysis in the ALL-TIMES FP7 project. The purpose of this project was to create methodologies and toolchains for combinations of different timing tools and techniques, including source-level flow analysis. The techniques were finally tried out on an industrial project from the automotive area. We ran into a number of problems when attempting to analyse the source code for the project.

We describe these problems, and the workarounds that we had to make to be able to proceed. The problems that we encountered seem to be quite general for source-level analysis of this kind of code, and we discuss a number of possible ways to tackle them.

The rest of this article is organized as follows: in Section 2 we give an account for related work. Section 3 describes the ALL-TIMES project briefly. In Section 4 we describe the target system for which we applied source-level flow analysis. Section 5 describes how the source-level analysis was used, and which aspects were tested. Section 6 describes our analysis tool SWEET, which was used in the case study, and its flow analysis. Section 7 reports on the results and experiences of the case study. Section 8, finally, wraps up and discusses possible directions for future work.

2 Related Work

There have been a number of studies of WCET analysis of industrial code. There are some reports on using commercial, static analysis WCET tools to analyze code for space applications [13,12,17], and in avionics industry [8,20,15]. In [4,18], time-critical parts of the commercial real-time OS Enea OSE were analysed. Experiences from WCET analysis of real-time communication software for automotive was reported in [3]. All these case studies concern analysis of binary code only. A problem detected in several of them is the need to provide manual annotations for program flow properties at binary level. This was often found to be cumbersome and error-prone.

[19] reports on WCET analysis of a number of tasks in the transmission control of an articulated hauler. This analysis was done on binary level. In a follow-up study [1], the program flow analysis was done on source level, and the results passed to the binary level analysis be hand. The experiment was successful in the sense that almost all program flow constraints on the binary level could be automatically derived on source level. Compiler optimizations changing the control structure did not pose any big problems for the translation. What did pose a problem was the fact that the tasks communicate through data stored in intricate data structures. Value annotations had to be provided for some of these values, which proved to be difficult for two reasons: the inability of the annotation language to express accurately the value fields in the data structures, and the difficulty to find proper value ranges for data stored and updated by several tasks. For this case study, there was access to all the source code. The code was strictly ANSI-C, and parsing did not pose any problems.

There are a number of general-purpose academic or commercial tools for source-level static analysis. Some examples are ASTRÉE [6], Coverity [2], Polyspace, and Klocwork. These tools can check for a variety of possible errors such as possible division by zero, array index out of range, or potential memory leaks to name a few. A comparative evaluation of Coverity, Polyspace, and Klocwork is found in [7].

In [2], a number of practical problems for source-level static analysis tools, when applied to industrial code, were reported. Among the experiences reported were the need to handle different environments for program development, problems parsing code accepted by the customers' compilers, and the need to reduce the ratio of false to true positives. Clearly these experiences apply to supporting source-level analyses for WCET analysis as well, and they are in line with what we report here.

3 The ALL-TIMES Project

ALL-TIMES (Integrating European Timing Analysis Technology)[4] is an FP7 small to-medium focused research project (STREP) with six partners: Mälardalen University (coordinator), Vienna University of Technology, AbsInt Angewandte Informatik GmbH, Gliwa GmbH, Symtavision GmbH, and Rapita Systems Ltd. The partners are either research groups, with academic tools applicable to timing analysis, or vendors of timing analysis tools. The project has brought the following main results:

- integrated methodologies for timing analysis,
- prototypes of integrated tool chains,
- new/improved code and timing analysis tools,
- new tool connections, including open tool interfaces, and
- a validation of the integrated tool chains.

Fig. 1 shows the different tools, and the different tool connections that have been created within the project. The connections enable the integrated tool chains that in turn support the integrated methodologies.

The tools have the following functions: SymTA/S is a system-level timing analysis tool performing tasks like schedulability analysis. aiT is a static WCET analysis tool, and RapiTime is a tool for WCET analysis and worst-case performance profiling which uses a hybrid method. T1 is a tool for timing- measurement, -analysis and -visualisation. SWEET is a static WCET analysis tool: however, only its program flow analysis part has been of concern in ALL-TIMES. SATIrE, finally, is a source-code program analysis and transformation workbench built on top of the Rose compiler framework[5].

With the new tool connections that have been implemented in the project, a number of integrated tool chains can be formed. An example is RapiTime passing estimated WCETs for tasks to SymTA/S through tool connection M. RapiTime, in turn, may have its analysis enhanced by source-level analyses. For instance, SATIrE may pass information about possible function pointer values through tool connection P, and SWEET may provide program flow constraints through tool connection T. (SWEET must then use SATIrE as a frontend using tool connection Q, see below.) In all, this constitutes a tool chain involving four of the tools in the project.

[4] Website: `www.all-times.org`
[5] `www.rosecompiler.org`

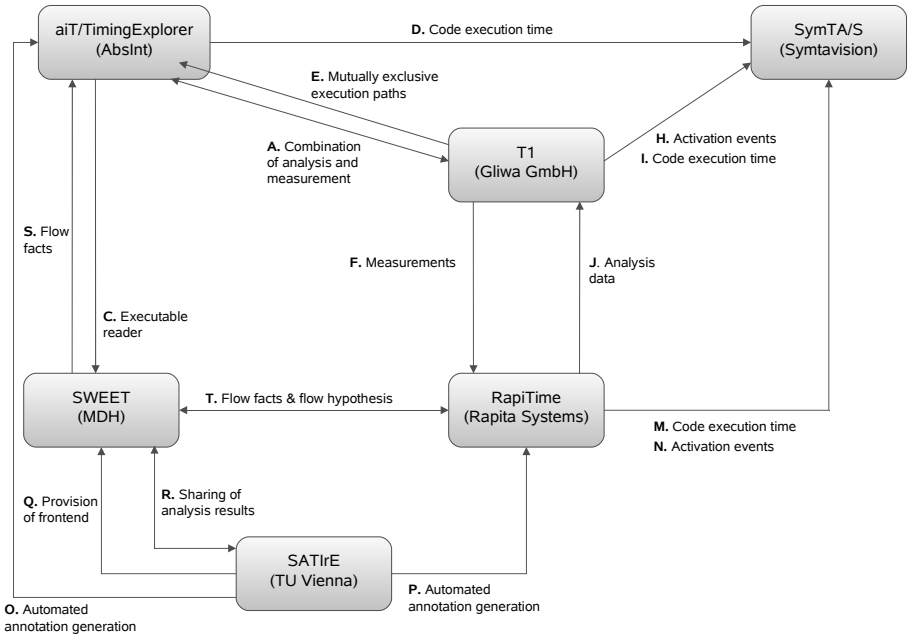


Fig. 1. ALL-TIMES integrations between timing analysis tools

An outspoken goal of ALL-TIMES has been to provide support for timing analysis in early, explorative design phases. Early indications of timing properties can help avoiding costly redesigns in situations where a late timing verification shows that the timing constraints are not met. For this purpose, a version of aiT called "TimingExplorer" was implemented. TimingExplorer sacrifices absolute safety of WCET estimates for analysis speed, and has means to do quick design space exploration varying the hardware configuration. It has the same interfaces as aiT, and can thus take part in various tool chains for providing early timing estimates. For instance, it can use information from source-level analyses performed by SATIrE and SWEET.

Another goal of ALL-TIMES has been to provide support for timing analysis from source-level analyses. Two kinds of program flow information were deemed to be of high interest: information about possible values of function pointers in different program points, and traditional program flow information such as bounds on the number of loop iterations, and infeasible path constraints. These analyses are provided by SATIrE and SWEET. SATIrE uses the C frontend of Rose to parse C source code before analysing it: it can then deliver function pointer sets to aiT and RapiTime using each tool's native format for manual source-level annotations. SATIrE also has an experimental loop bounds analysis. SWEET can compute a number of program flow constraints, ranging from simple loop bounds to complex infeasible path constraints, and can export these to aiT and RapiTime using the same annotation formats as SATIrE. SWEET's flow analysis analyses the ALF code format (see Section 6). SATIrE can convert

Rose's internal program representation to ALF through the "melmac" tool, and can thus act as a C frontend to SWEET. Thus, the source-level analysis of SWEET always uses SATIrE as a frontend.

The results of ALL-TIMES were validated using industrial code from the automotive domain, see Section 4. The validation included an estimation of the productivity increase brought by the technologies developed in ALL-TIMES. Two typical timing analysis scenarios were analysed, where for each step the time to perform it was estimated with and without the support of the ALL-TIMES technologies, respectively. The results indicate that the improved timing analysis support brought by ALL-TIMES can have a very significant impact on the efficiency of the timing analysis process.

4 The Target System

The target system that was used for the final validation in ALL-TIMES is an automotive embedded control unit. For this unit, the project could obtain access to the source code. This was crucial for the validation of the source-level tools and -connections. The unit has a Freescale MPC 564 processor. It has no hardware trace facilities, and it uses the ETAS ERCOSEK operating system. The source code consists of ca. 685k lines of code, divided on 1297 C files (.c), 768 C header files (.h), and 28 assembly files (.a). The C files range in size from less than 1 KB to 1997 KB (22910 lines of code). The source code is compiled with the Windriver compiler (Diab Data).

In addition to the source code, the project had access to a test environment including compiler, debugger, and hardware.

5 Source Code Analysis Validation

The source code analyses developed within ALL-TIMES were validated as parts of larger tool chains, involving also the WCET analysis and system-level timing analysis tools in the project (see Section 3). The tool chains were validated using two scenarios, in the following way. The first validation scenario was an "early design exploration" scenario, where TimingExplorer is used to select an appropriately upgraded hardware platform for an existing application. For this purpose, a set of approximate WCETs of the tasks were computed for each investigated hardware configuration, and they were subsequently sent to SymTA/S for a schedulability analysis. Computing bounded WCETs require that all loops in the code are bounded. TimingExplorer does perform an automatic loop bounds analysis: however, this analysis is done on the binary code, and so, for the target code of the validator, TimingExplorer was unable to bound 102 loops. The source-level analysis validation for this scenario was to see how many loops SWEET could additionally bound by a source-level analysis, and pass the results to TimingExplorer to avoid the time-consuming process of manually annotating the loop bounds.

The second scenario was a "late stage verification" scenario, where SATIrE, SWEET, and RapiTime are used to analyse/measure the code and determine the WCETs of the tasks in the target system. RapiTime produces traces that are visualised using one of the so-called "trace-viewers" provided with T1 or SymTA/S. SymTA/S also performs a scheduling analysis to verify schedulability under worst-case conditions.

Here, it was tested how well SATIrE could determine function pointer sets, which RapiTime needs to cut down the set of possible execution paths when calling a function that is indirectly referenced through such a pointer. SWEET's loop analysis was used in a different way than for the first scenario. RapiTime measures execution times for program fragments, and combines this information into a WCET estimate using program flow constraints. The validator has no hardware tracing facilities, and thus the code has to be instrumented for measuring execution times. The instrumentation has to be sparse since otherwise buffers would overflow and data would be lost. Therefore it is interesting to identify parts of the code with little variability of the execution time, since such parts are prime candidates not to instrument. Loops that always execute the same number of times are likely to have low execution time variability. Since SWEET can compute both upper and lower iteration bounds for loops, it was tested how many loops SWEET could find where these bounds were equal (and thus the loop always executes the same number of times). This information could help reduce a time-consuming inspection of the code to find those parts of the code where instrumentation could be reduced.

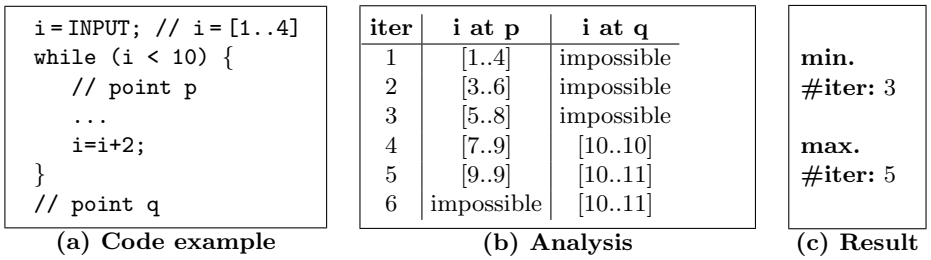
6 SWEET, and its Flow Analysis

SWEET[6] is a WCET analysis research tool from MDH. It has the standard WCET analysis tool architecture with a flow analysis, a low-level analysis, and a final WCET estimate calculation.

The flow analysis of SWEET is the part currently being actively maintained and developed, and this is the part of SWEET that has been of concern for the ALL-TIMES project. The analysis is able to produce a number of program flow constraints, ranging from simple upper and lower loop iteration bounds to complex infeasible flow constraints. The analysis is context-sensitive as well as *input-sensitive*, the latter meaning that the analysis can take restrictions on input values into account when computing the program flow constraints.

The main analysis method of SWEET is called *abstract execution* (AE) [10]. AE can be seen as a fully context-sensitive abstract interpretation, where each loop iteration (or recursive function call) is analysed separately. It is therefore a "deep" analysis method, based on the semantics of the program. Alternatively, AE can be seen as a kind of symbolic execution where program variables store abstract values, and where operators and functions are given their interpretations in the abstract value domain. Fig. 2 illustrates how AE works for a simple

[6] `www.mrtc.mdh.se/projects/wcet/sweet.html`



(a) Code example **(b) Analysis** **(c) Result**

Fig. 2. Example of abstract execution

loop. Note the input-sensitivity: the restricting interval for the initial value of `i` will directly influence the resulting iteration bounds for the loop. A simpler, non-input-sensitive analysis would not be able to bound this loop. The current implementation of AE in SWEET uses the abstract interval domain for numerical values, but AE can in principle use any abstract domain.

The computed flow constraints are arithmetic constraints on so-called *execution counters*, virtual variables that count the number of executions in different program points. (In Fig. 2, "**#iter**" is such a counter.) SWEET has a general method to collect information about possible executions into different kinds of program flow constraints. The resulting constraints can be directly used in the final WCET estimate calculation.

The analysis method of SWEET is general and not a priori tied to any language or instruction set. To take advantage of this generality, SWEET's flow analysis analyses the code format ALF [9]. ALF is an intermediate format designed to be able to faithfully represent code on different levels, from source to binary level. It therefore contains high-level concepts, such as functions and function calls, as well as low-level concepts like dynamic jumps. Thus, SWEET can analyse code both on binary and source level provided that a translator into ALF is present. As mentioned, the melmac tool is such a translator that maps the internal format of Rose into ALF, thereby enabling SATIrE to be used as a C frontend to SWEET.

7 Results and Experiences

We now describe our experiences from doing source-level analysis on the validator (automotive ECU) code in the ALL-TIMES project. We discuss the problems encountered for the source-level program flow analysis performed by SWEET. Most of the problems are common to SATIrE's analyses as well, and indeed they should be common to "deep", semantics-based source-level analyses in general.

The steps necessary to perform SWEET's flow analysis are basically the following. This scheme should hold for source-level analysis in general:

1. identify the source files that are needed for the analysis,
2. convert the files to a format suitable for the static analyses to be applied,

3. "link" the converted files to enable inter-procedural program analyses,
4. perform the program analyses, and
5. map the results back to the source code, in a format suitable for other tools.

We now describe our experiences for each step in turn.

7.1 Step 1: Identify Needed Source Files

Academic benchmarks for WCET analysis research tend to consist of small, self-contained programs where each program to analyse resides in a single source file. Not so in real projects: they may consist of thousands of files, where the code that needs to be analysed is scattered over many different files.

The first step is to *identify the entry points for the code to analyse*. For WCET analysis, this typically amounts to finding the start functions for the tasks, or runnables, in the system. If the application uses an operating system, then the tasks can usually be found from some of its configuration files. This is the method that we used. It requires knowledge of how the specific operating system used is configured.

Second, the files containing code that is needed for the analysis are to be identified. This may sound trivial: a brute force approach that should work would seemingly be to include all source files in the project. But this requires knowledge of where the source files are located, which typically has to come from the build tool (which, if unlucky, may be some set of obscure scripts). A complication is that some source files may include assembler, or that they are simply missing (only binaries available): more on the consequences of this in Section 7.4.

A further complication, which we encountered in the project, is *late binding*. Functions with the same name may be defined in different compilation units, and first at linking time it is selected which of these functions to actually call, by including the corresponding object file. Late binding seems to be very common for embedded codes. With late binding, the source code simply does not contain sufficient information which source files to include, information from the build tool is needed.

To summarize: Step 1 in general requires information both from the operating system configuration and the build tool, and a source-level analysis tool must somehow acquire it.

7.2 Step 2: Converting Source Files

For SWEET's flow analysis, the conversion is done in two steps: first, the source files are parsed into the internal format of Rose, and then a translation to ALF is done by the melmac tool.

SWEET needs to generate flow constraints in the annotation formats of aiT and RapiTime. Both these annotation formats relate the flow constraints to locations in the source code. Therefore, in addition to the code conversion, melmac

generates a so-called *map file* relating code locations in ALF with the corresponding locations in the original source code.

The two first steps presented problems. The Rose compiler framework uses a commercial C/C++ frontend which is distributed with Rose as a binary, so it cannot be changed. It turned out that the parser could not process some of the target system's source files. We had to rewrite some source files by hand to get them through the parser. Some source files also contained inlined assembler, which the frontend could not process either. We dealt with this mainly by commenting out the assembler, although this is not a strictly safe solution. Other constructs causing problems were compiler specific pragmas, and preprocessor directives. The problems encountered were very similar to those reported in [2].

It also turned out that Rose itself had problems to handle some features of the target system's C code. We had to rely on the Rose developers to for fixing the problems, which took time.

The morale of this is that an industrial strength source analysis tool, at least for C, must be able to adapt to the variety of C dialects defined by the C compilers that are in use. It is hard to accomplish this without being in full control of the translation chain.

7.3 Step 3: Link the Converted Files

When analysing code that stretches several files or compilation units, a global namespace has to be created where symbols have an unambiguous meaning. This process can be seen as a kind of linking [14], but on source-code level.

Source-level linkers exist. One example is `cilly`, from the CIL framework [5], which merges a number of C file into one large C file. However, `cilly` changes the line numbers of statements in a way that cannot be easily traced. The original line numbers are required for flow constraint annotations in the annotation format for aiT, so if they are lost then such annotations cannot be generated. For this and other reasons, we decided to create or own linker for ALF code instead.

Our ALF linker works similarly to `cilly`, and merges all the ALF files into one big file. However, it also maintains the relation between program code locations in C and ALF files by creating a global map file from the merged ALF files. To differ between local entities with the same name, the ALF linker has to perform some "name mangling" to create unique names.

A problem, similar to the late binding problems described in Section 7.1, was caused by the fact that some header (.h) files contained full function definitions. Since the C preprocessor's semantics of including header files is to copy them verbatim, the result was the creation of several instances of an identical function with the same name in different C files. The C preprocessor would then be used to select which copy to include in the end. However, since the C files were translated into ALF one by one, the result would be several ALF files with identical function declarations. The ALF linker, righteously, did not accept this. Our solution was to change the linker to eliminate all identical copies of a function declaration but one.

7.4 Step 4: Performing the Flow Analysis

The linked ALF file should in principle be ready for analysis. Alas, a successful flow analysis for real industrial codes typically requires additional information not present in the available source code. This information must then somehow be provided, or approximated.

When analysing the possible program flows of a task, typically some bounds must be known for inputs that may affect the program flow (see Fig. 2 for an example). Inputs can be either arguments to the start function of the task, if any, and values of global variables when the start functions begin executing.

Especially important is to restrict the possible values of input pointers. If the abstract value of a pointer is TOP (any address), and an assignment is done using the pointer value as target, then the analysis must assume that the written value may possibly be written to any data address. This typically destroys the information needed to identify any interesting program flow constraints.

SWEET provides value annotations for arguments and variables, which allow the user to provide simple interval bounds for these. Inputs that are not annotated will assume the abstract value TOP (any value is possible). However, the problem remains how to identify the important inputs and find proper value ranges for these.

Since we had little time to make elaborate annotations in the ALL-TIMES project, we would mostly let input values default to TOP. We suspect that this affected the precision of the analysis adversely in many cases.

For systems that allow parallel or pre-emptive execution of tasks accessing shared data areas, the additional complication arises that during the execution of a task, a variable can be reassigned a new value by some other task. Such shared variables must be treated differently in the analysis. If there is no knowledge about the possible preemption points, then the value of such a variable must be considered possible to change, due to the actions of some other task in the system, at any program point. An analysis that does not take this into account is potentially unsafe. As far as we know, all current WCET analysis tools ignore this problem.

C has a keyword "volatile", which is a hint to the compiler that the variable might be changed by some other activity in the system. A variable might be marked volatile for many other reasons, though, and conversely any "non-volatile" global variable could potentially be shared between tasks. Analyses that identify shared data and compute safe value bounds for these are conceivable: however, we had no time to develop such analyses in the project. SWEET currently deals with the problem by forcing volatiles to assume the abstract value TOP throughout the program, but as explained this is potentially unsafe.

The second problem is that vital source code often is missing. For the validator, the big problem turned out to be that important parts of the code came from a subcontractor, and we had no access to its source code. Apparently this is a common situation in the automotive domain, and the trend towards componentising the software and use third-party code seems to become stronger. This

means that missing source code will be a significant problem for source-level analysis in the future.

We tackled this problem in two ways. First, we added a mode to SWEET where it will assume a certain default semantics for unknown entities. Unknown variables will be set to TOP, which is a safe overapproximation but may lead to very imprecise results especially of the variable holds pointer values. The default semantics for unknown functions is that their return value is TOP, and that they do not affect any global variables (no side-effects). The latter assumption is clearly unsafe, but without knowledge of the function a safe analysis will have to assume that it can set any global variable to anything, which will render the resulting analysis extremely imprecise.

Second, we extended the annotation language of SWEET to give it some limited capability to express the semantics of unknown functions. These annotations allow to specify bounds on return values, and for possible side effects on given global variables.

Missing code, and uncertainty about the nature of volatile-declared variables, posed a problem for the source-level validation. As mentioned, TimingExplorer failed to bound 102 loops, located in 50 different C functions. Of these, only 19 functions could be translated into ALF (mostly due to that the source code was missing). 24 of the not bounded loops were present in those functions. Two of these 19 functions refer to unknown entities, and 18 functions refer to volatiles. Of the total number of functions that could be translated into ALF, about 49% contained neither undefined entities nor volatiles (and should thus be possible to analyse with safe results, assuming that no non-volatile globals can be affected from outside the analysed task).

There were 18 start functions for tasks in the system. Of these 10 referred to undefined functions, and 16 to undefined data. Nine contained loops for which we had ALF code, and which TimingExplorer could not bound. For a number of reasons, we could not obtain any bounds for any of the not bounded loops by analysing the start functions. This was due to imprecision from unknown entities (especially unknown pointers), but also presence of volatiles whose safe treatment required them to be set to TOP, and infinite `for(;;);` loops that pose problems for the abstract execution algorithm used by SWEET. Thus, we resorted to analyzing functions deeper in the call tree, containing (possibly indirect) calls to the functons with the interesting loops. This had the advantage of avoiding to analyse much problematic code. On the other hand, calling context information was lost which would decrease the precision. By analysing these functions, we were able to bound three of the previously unbounded loops by a safe analysis (w.r.t. undefined entities and volatiles), and two more by a potentially unsafe analysis making default assumptions about undefined entities, and treating volatiles as "ordinary" variables. We believe that the low number of interesting loops that we were able to bound mainly is due to imprecision from undefined entities, and lack of knowledge about the calling contexts.

7.5 Step 5: Map Results Back to Source Code

The last step is to map flow constraints back to TimingExplorer or RapiTime using their native source-level annotation formats. Also here, we encountered some practical problems. One problem is that some precision was lost in the translation to these formats: for instance, SWEET can generate highly context-sensitive flow constraints which cannot be expressed to the same degree in them. Another issue is that SWEET derives loop bounds as bounds on the number of times the loop header executes, while both annotations formats use the number of times that the loop body executes. This forced us to make adjustments in the generated loop annotations.

8 Conclusions

We have identified a number of problems with source-level program flow analysis. Some problems are due to important information not being present in the source code, some to weak language standards (allowing different compilers to define different dialects), and some to missing source code. Most problems should be common to deep, semantics based source-level analysis in general. Thus, solutions are important not only for WCET analysis, but also for general source-level analysis.

It is not easy to change industrial practices. However, we can still make a "wish list". One thing on the list would be standardised formats for build processes, which an analysis tool could read to find the relevant source code to analyse. On the task level, clean standards for task communication are very desirable. Metadata should be provided, in standardised formats, that describes the task communication as well as the task entry points. Metadata should also be available to describe possible value restrictions on inputs to the system, like values read from sensors.

It should also be possible for third-party code providers to ship metadata describing the semantics of the delivered code. This metadata could then be used by an analysis tool when the source code is not provided. Examples of useful metadata are input-output relations for functions, possible side effects, and value restrictions for global variables. Function side effects can be expressed in types, and found by a type-based analysis [16].

The natural place to standardise metadata formats is in component-oriented frameworks such as AUTOSAR. However, also without them, there are improvements to be done. One interesting option is to develop mixed source/binary-level analyses, that use the binaries when sources are not available. Using a multilevel language as ALF for the analysis, which allows translations from both source- and binary level code, is then clearly an advantage. Analysis tools also need better annotation languages to let the user provide precise information when needed.

Finally, note that most problems described above will not appear for a binary level analysis of a statically linked executable. All the code will then be available,

and all ambiguities will have been resolved by the compiler and linker. This compensates for the disadvantages of binary level analysis to some degree. However, we think that source-level analysis offers distinct advantages warranting efforts to make it applicable to real embedded production codes.

References

1. Barkah, D., Ermedahl, A., Gustafsson, J., Lisper, B., Sandberg, C.: Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In: Proc. 20^{th} Euromicro Conference of Real-Time Systems (Jul 2008)
2. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Comm. ACM 53(2) (Feb 2010)
3. Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying static WCET analysis to automotive communication software. In: Proc. 17^{th} Euromicro Conference of Real-Time Systems, (ECRTS'05) (Jul 2005)
4. Carlsson, M., Engblom, J., Ermedahl, A., Lindblad, J., Lisper, B.: Worst-case execution time analysis of disable interrupt regions in a commercial real-time operating system. In: Pettersson, P., Yi, W. (eds.) Proc. 2^{nd} International Workshop on Real-Time Tools. Copenhagen (Aug 2002)
5. CIL - infrastructure for C program analysis and transformation (2008), `manju.cs.berkeley.edu/cil`
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) Proc. 14th European Symposium on Programming. Lecture Notes in Comput. Sci., vol. 3444, pp. 21–30. Springer-Verlag, Edinburgh (Apr 2005)
7. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools (extended version). Technical report, Linköping University (Jan 2008), `http://www.ep.liu.se/ea/trcis/2008/003/`
8. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Proc. 1^{st} International Workshop on Embedded Systems, (EMSOFT2000), LNCS 2211 (Oct 2001)
9. Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., Källberg, L.: ALF – a language for WCET flow analysis. In: Proc. 9^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2009). pp. 1–11. Dublin, Ireland (Jun 2009)
10. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: 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) (Dec 2006)
11. Healy, C., Sjödin, M., Rustagi, V., Whalley, D., van Engelen, R.: Supporting timing analysis by automatic bounding of loop iterations. Journal of Real-Time Systems 18(2-3), 129–156 (May 2000)
12. Holsti, N., Långbacka, T., Saarinen, S.: Using a worst-case execution-time tool for real-time verification of the DEBIE software. In: Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457) (Sep 2000)
13. Holsti, N., Långbacka, T., Saarinen, S.: Worst-case execution-time analysis for digital signal processors. In: Proc. EUSIPCO 2000 Conference (X European Signal Processing Conference) (2000)
14. Levine, J.: Linkers and Loaders. Morgan Kaufmann (2000), iSBN 1-55860-496-0

15. Montag, P., Goerzig, S., Levi, P.: Challenges of timing verification tools in the automotive domain. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06). Paphos, Cyprus (Nov 2006)
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2^{nd} edition. Springer (2005), iSBN 3-540-65410-0
17. Rodriguez, M., Silva, N., Esteves, J., Henriques, L., Costa, D., Holsti, N., Hjortnaes, K.: Challenges in calculating the WCET of a complex on-board satellite application. In: Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis (WCET'2003). Porto (Jul 2003)
18. Sandell, D., Ermedahl, A., Gustafsson, J., Lisper, B.: Static timing analysis of real-time operating system code. In: Proc. 1^{st} International Symposium on Leveraging Applications of Formal Methods (ISOLA'04) (Oct 2004)
19. Sehlberg, D., Ermedahl, A., Gustafsson, J., Lisper, B., Wiegratz, S.: Static WCET analysis of real-time task-oriented code in vehicle control systems. In: Margaria, T., Philippou, A., Steffen, B. (eds.) Proc. 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06). Paphos, Cyprus (Nov 2006)
20. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software. In: Proc. of the IEEE Int. Conf. on Dependable Systems and Networks (DSN-2003) (Jun 2003)