View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

ALF – A LANGUAGE FOR WCET FLOW ANALYSIS

Jan Gustafsson[1], Andreas Ermedahl[1], Björn Lisper[1], Christer Sandberg[1], and Linus Källberg[1]

Abstract

Static Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A key component in static WCET analysis is the flow analysis, which derives bounds on the number of times different code entities can be executed. Examples of flow information derived by a flow analysis are loop bounds and infeasible paths.
Flow analysis can be performed on source code, intermediate code, or binary code: for the latter, there is a proliferation of instruction sets. Thus, flow analysis must deal with many code formats. However, the basic flow analysis techniques are more or less the same regardless of the code format. Thus, an interesting option is to define a common code format for flow analysis, which also allows for easy translation from the other formats. Flow analyses for this common format will then be portable, in principle supporting all types of code formats which can be translated to this format. Further, a common format simplifies the development of flow analyses, since only one specific code format needs to be targeted.
This paper presents such a common code format, the ALF language (ARTIST2 Language for WCET Flow Analysis).

1. Introduction

Bounding the Worst-Case Execution Time (WCET) is crucial when verifying real-time properties. A static WCET analysis finds an upper bound to the WCET of a program from mathematical models of the hardware and software involved. If the models are correct, the analysis will derive a timing estimate that is *safe*, i.e., greater than or equal to the WCET.

To statically derive a timing bound for a program, information on both the hardware timing characteristics, as well as the program's possible execution flows, needs to be derived. The latter includes information about the maximum number of times loops are iterated, infeasible paths, etc. The goal of a *flow analysis* is to calculate such flow information as automatically as possible. A precise flow analysis is of great importance for calculating a tight WCET estimate [18].

[1] School of Innovation, Design and Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden.
{jan.gustafsson,andreas.ermedahl,bjorn.lisper,christer.sandberg,
linus.kallberg}@mdh.se

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2279

The input to the flow analysis is a representation of the program to be analysed as, e.g., binary code, intermediate code, or source code. These alternatives have different pros and cons:

Binary code. This is the code actually being run on the processor, and thus the code for which the flow information is relevant. Analyzing the binary code therefore ensures that the correct flow information is found. However, information available in the source code may be lost in the binary, which can lead to a less precise flow analysis. To do a good job, an analysis tool will have to reconstruct the high-level program structure such as the control flow graph, function calls and returns, etc. The WCET analysis tools aiT [2] and Bound-T [7] analyze binary code, and include both a type of instruction decoding- and program-graph reconstruction phase [6, 19].

Source code. Source code can typically be analyzed more precisely, since there is much more information present. On the other hand, compiler optimizations can cause the control structure of the generated binary code to be different from the one of the source code. Therefore, to be used for WCET analysis the flow information must be transformed accordingly [10]. This requires compiler support, which current production compilers do not offer. On the other hand, source-code analysis can be used for other purposes such as deriving and presenting program flow information to the developer. Thus, flow analysis of source code is definitely of interest. Typically, when source code is analysed, the code is often translated to some type of intermediate code which is close to the source code. An example of a WCET analysis tool which works on source code level is TuBound [16].

Intermediate code. This kind of code is typically used by compilers, for the purpose of optimizing transformations, before the binary code is generated. It is often quite rich in information. The intermediate code generated by the parser is usually more or less isomorphic to the source code, while after optimizations it typically has a program flow which is close to the flow in the generated binary code. These properties make intermediate code an interesting candidate for flow analysis, since it can be analyzed for flow properties of both source and binary code. On the downside, the analysis becomes dependent on a certain compiler: code generated by another compiler cannot be analyzed. The current version of the WCET analysis tool SWEET analyzes the intermediate format of the NIC research compiler [4].

ALF is developed with flexibility in mind. The idea behind ALF is to have a generic language for WCET flow analysis, which can be generated from all of the program representations mentioned above. In this paper we describe ALF, its intended use, and some current projects where ALF will be used. For a complete description of ALF, see the language specification [3].

The rest of the paper is organized as follows: Section 2 describes the ALF language. Section 3 presents the intended uses of ALF, and in Section 4 we describe some of the current projects where ALF is being used. Section 5 presents some related work, and in Section 6 we draw some conclusions and discuss future work.

2. ALF (ARTIST2 Language for WCET Flow Analysis)

ALF is a language to be used for flow analysis for WCET calculation. It is an intermediate level language which is designed for analyzability rather than code generation. It is furthermore designed

to represent code on source-, intermediate- and binary level (linked as well as unlinked) through relatively direct translations, which maintain the information present in the original code needed to perform a precise flow analysis.

ALF is basically a sequential imperative language. Unlike many intermediate formats, ALF has a fully textual representation: it can thus be seen as an ordinary programming language, although it is intended to be generated by tools rather than written by hand.

2.1. Syntax

ALF has a Lisp/Erlang-like syntax, to make it easy to parse and read. This syntax uses prefix notation as in Lisp, but with curly brackets "{", "}" as parentheses as in Erlang. An example is

```
{ dec_unsigned 32 2 }
```

which denotes the unsigned 32-bit constant 2.

2.2. Memory Model

ALF's memory model distinguishes between program and data addresses. It is essentially a memory model for relocatable, unlinked code. Program and data addresses both have a symbolic base address, and a numerical offset. Program addresses are called *labels*. The address spaces for code and data are disjoint. Only data can be modified: thus, self-modifying programs cannot be modelled in ALF in a direct way.

2.3. Program Model

ALF's program model is quite high-level, and similar to C. An ALF program is a sequence of declarations, and its executable code is divided into a number of function declarations. Within each function, the program is a linear sequence of statements, with execution normally flowing from one statement to the next. Statements may be tagged with labels. ALF has jumps, which can go to dynamically calculated labels: this can be used to represent program control in low-level code. In addition ALF also has structured function calls, which are useful when representing high-level code. See further Sections 2.6 and 2.7

A function named "main" will be executed when an ALF program runs. ALF programs without a main function cannot be run, but may still be analyzed.

2.4. Data Model

ALF's data memory is divided into *frames*. Each frame has a symbolic base pointer (a *frameref*) and a *size*. A data address pointing into a frame is formed from the frameref of the frame and an offset. The offset is a natural number in the *least addressable unit* (LAU) of the ALF program. The LAU is always declared: typically it is set to a byte (8 bits).

Frames can be either statically or dynamically allocated. Statically allocated memory is explicitly declared. There are two ways to allocate memory dynamically:

- As local data in so-called *scopes*. This kind of local data is declared like statically allocated data, but in the context of a scope rather than globally. It would typically be allocated on a stack.
- By calling a special function "dyn_alloc" (similar to malloc in C) that returns a frameref to a newly allocated frame. This kind of data would typically be allocated on a heap.

Semantically, framerefs are actually pairs (i, n) where i is an identifier and n a natural number. For framerefs pointing to statically allocated frames, n is zero. For frames dynamically allocated with dyn_alloc, each new allocation returns a new frameref with n increased by one. This semantics makes it possible to perform static analysis of dynamically allocated memory, like bounding the maximal number of allocations to a certain data area.

ALF's data memory model can be used both for high-level code, intermediate formats, and binaries, like in the following:

- "High-level": allocate one frame per high-level data structure in the program (struct, array, ...)
- "Intermediate-level": use one frameref identifier for each stack, heap etc. in the runtime system, in order to model it by a potentially unbounded "array" of frames (one for each object stored in the data structure). Use one frame for each possible register. If the intermediate model has an infinite register bank, then use one identifier to model the bank (again one frame per register).
- Binaries: allocate one frame per register, and a single frame for the whole main memory.

2.5. Values and Operators

Values can be:

- Numerical values: signed/unsigned integers, floats, etc.,
- Framerefs,
- *Label references* (lrefs), which are symbolic base addresses to code addresses (labels),
- Data addresses (f, o), where f is a frameref and o is an offset (natural number), or
- Code addresses (labels) (f, n), where f is an lref and n a natural number.

There is a special value undefined. This provides a fallback in situations where an ALF-producing tool, e.g., a binary-to-ALF translator, cannot translate a piece of the binary code into sensible ALF.

Each value in ALF has a *size*, which is the number of bits needed for storing it. ALF has integers of unbounded size: these are used mainly when translating language constructs into ALF that are not easily modelled with ALF's predefined operators. All other values have finite size, and they are *storable*, meaning that they can be stored in memory. Storable values are further subdivided into *bitstring* values, which have a unique bitstring representation, and *symbolic* values. The latter include

address values, since they have symbolic base addresses. Only a limited number of operations are allowed on symbolic values.

Bitstring values are primarily numeric values. A bitstring value can be numerically interpreted in various ways depending on its use: as signed, unsigned, floating-point, etc. This is a complication when performing a value analysis of ALF, since this analysis must take the different possible interpretations into account. A *soft type system* [20] for ALF, which can detect how values are used and restrict their possible interpretations, is under development [3].

ALF has an explicit syntax for constant values, which includes the size and the intended interpretation. This is intended to aid the analysis of ALF programs.

Operators in ALF are of different kinds. The most important ones are the operators on *data of limited size*. These operators mostly model the functional semantics of common machine instructions (arithmetic and bitwise logical operations, shifts, comparisons, etc). ALF also has operators on *data of unbounded size*. These are "mathematical" operations, typically on integers: for instance, all "usual" arithmetic/relational operators have versions for unbounded integers. They are intended to be used to model the functional semantics for instructions whose results cannot be expressed directly using the finite-size-data operators. An example is the settings of different flags after having carried out certain operations, where ALF sometimes does not provide a direct operator.

Furthermore, ALF has a few operators on *bitstrings*, like bitstring concatenation. They are intended to model the semantics on a bitstring level, which is appropriate for different operations involving masking etc. There is a variety of such operations in different instruction sets, and it would be hard to provide direct operators for them all. ALF also has a *conditional*, and a *conversion function* from bitstrings to natural numbers. These functions are useful when defining the functional semantics of instructions.

Finally, ALF has a *load* operator that reads data from a memory address, and the aforementioned `dyn_alloc` function that allocates a new frame.

All these operators, except `dyn_alloc`, are side-effect free. This simplifies the analysis of ALF programs. Each operator takes one or several *size arguments*, which give the size(s) of the operands, and the result. ALF is very explicit about the sizes of operands and operators, in order to simplify analysis and make the semantics exact.

2.6. Statements

ALF has a number of statements. These are similar to statements in high-level languages in that they typically take full expressions as arguments, rather than register values. The most important are the following:

- A concurrent assignment (`store`), which stores a list of values to a list of addresses. Both values and addresses are dynamically evaluated from expressions.
- A multiway jump (`switch`), which computes a numerical value and then jumps to a dynamically computed address depending on the numerical value. For convenience, ALF also has an unconditional `jump` statement.

- Function `call` and `return` statements. The `call` statement takes three arguments: the function to be called (a code address), a list of actual arguments, and a list of addresses where the return values are to be stored upon return from the called function. Correspondingly, the `return` statement takes a list of expressions as arguments, whose values are returned. The address specifying the function to be called can be dynamically computed.

2.7. Functions

ALF has procedures, or *functions*. A function has a name and a list of formal arguments, which are frames. The body of a function is a scope, which means that a function also can have local variables. The formal arguments are similar to locally declared variables in scopes, with the difference that they are initialized with the values of the actual arguments when the function is called.

As explained in Section 2.6, a function is exited by executing a `return` statement. A function is also exited when the end of its body is reached, but the result parameters will then be undefined after the return of the call. Functions can be recursive.

Functions can only be declared at a single, global level, and a declared function is visible in the whole ALF program. ALF has lexical (static) scoping: locally defined variables, or formal arguments, take precedence over globally defined entities with the same name. This is similar to C.

2.8. The Semantics of ALF

ALF is an imperative language, with a relatively standard semantics based on state transitions. The state is comprised of the contents in data memory, a program counter (PC) holding the (possibly implicit) label of the current statement to be executed, and some representation of the stacked environments for (possibly recursive) function calls.

2.9. An Example of ALF Code

The following C code:

```
if(x > y) z = 42;
```

can be translated into the ALF code below:

```
{ switch { s_le 32 { load 32 { addr 32 { fref 32 x } { dec_unsigned 32 0 } } }
                   { load 32 { addr 32 { fref 32 y } { dec_unsigned 32 0 } } } }
         { target { dec_unsigned 1 1 }
                  { label 32 { lref 32 exit } { dec_unsigned 32 0 } } } }
{ store { addr 32 { fref 32 z } { dec_unsigned 32 0 } }
        with { dec_signed 32 42 } }
{ label 32 { lref 32 exit } { dec_unsigned 32 0 } }
```

The `if` statement is translated into a `switch` statement jumping to the exit label if the (negated) test becomes true (returns one). The test uses the `s_le` operator (signed less-than or equal), taking 32 bit arguments and returning a single bit (unsigned, size one). Each variable is represented by a frame of size 32 bits.

3. Flow Analysis Using ALF

ALF will be able to offer a high degree of flexibility. Provided that translators are available, the input can be selected from a wide set of sources, like linked binaries, source code, and compiler intermediate formats: see Section 4.3 for some available translators. The flow analysis is then performed on the ALF code.

If a single, standardized format like ALF is used, then it is natural to implement different flow analyses for this format. This facilitates direct comparisons between different analysis methods. Also, since different analyses are likely to be the most effective ones for different kinds of translated input formats, the analysis of heterogeneous programs is facilitated. This can be useful in situations where parts of the analysed program are available as, say, source code, while other parts, like library routines, are available only as binaries. Also, different parts of the program with different characteristics may require different flow analysis methods, e.g., input dependent functions may require a parametric analysis [12]. Flow analysis methods can be used as "plug-ins" and even share results with other methods. Several flow analysis methods can be used in parallel and the best results can be selected.

ALF itself does not carry flow data. The flow analysis results must be mapped back as flow constraints on the code from which the ALF code was generated. An ALF generator can use conventions for generating label names in order to facilitate this mapping back to program points in the original code.

As a future general format for flow constraints, the *Program Flow Fact* (PFF) format is being developed to accompany ALF. PFF will be used to express the dynamic execution properties of a program, in terms of constraints on the number of times different program entities can be executed in different program contexts.

4. Current Projects Using ALF

4.1. ALL-TIMES

ALL-TIMES [5] is a medium-scale focused-research project within the 7th Framework Programme. The overall goal of ALL-TIMES is to integrate European timing analysis technology. The project is concerned with embedded systems that are subject to safety, availability, reliability, and performance requirements. These requirements often relate to correct timing, notably in the automotive and aerospace areas. Consequently, the need for appropriate timing analysis methods and tools is growing rapidly.

ALL-TIMES will enable interoperability of the various tools from leading commercial vendors and universities alike, and develop integrated tool chains using open tool frameworks and interfaces. The different types of connections between the tools are:

- provision of data: one tool provides some of its analysis data to a another tool
- provision of component: the component of one tool is provided to another partner for integration in the other tool
- sharing of data: this is a bidirectional connection where similar data is computed by both tools and the connector allows to exchange that information in both directions

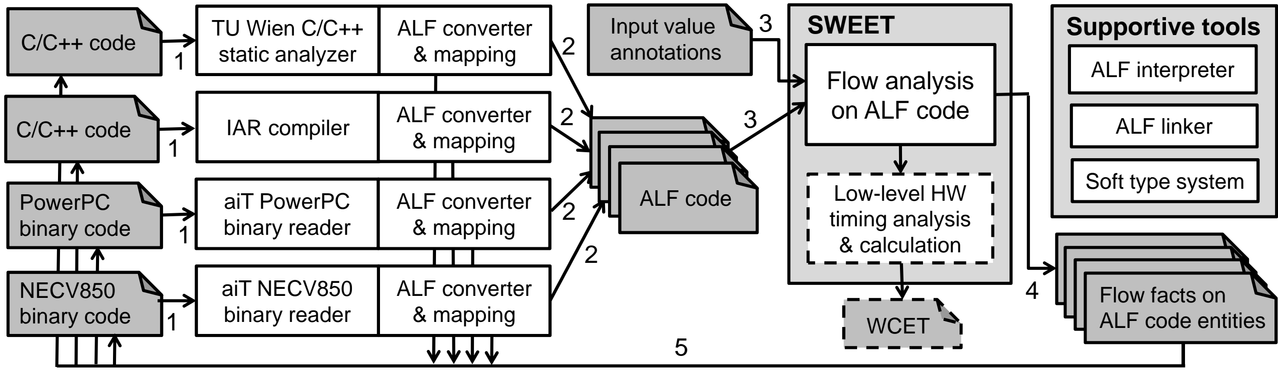

Figure 1. The use of ALF with the SWEET tool

- combination of features: the foreseen connection allows to combine features from different tools in both directions (either by exchanging data or components)

ALF is being developed within the ALL-TIMES project, where it will contribute to the interoperability between WCET tools.

4.2. ALF and SWEET

ALF will be used as input format for the WCET analysis tool SWEET (SWEdish Execution Time tool), which is a prototype WCET analysis tool developed at Mälardalen University [13].

Figure 1 describes the intended use of ALF in conjunction with SWEET. Step 1 represents the reading of the input program code, represented in different formats and levels. Step 2 describes the output in the generic ALF format. Step 3 shows the inputs to the flow analysis, which (in Step 4) outputs the results as flow constraints (flow facts) on ALF code entities. Using the mapping information created earlier, these flow facts can be mapped back to the inputs formats (Step 5). The four different translators to ALF are described shortly below.

4.3. Translators to ALF

A number of translators to ALF code are currently being developed. All three types of sources will be covered; source code, intermediate code, and binary code.

C/C++ source code to ALF translator. This C/C++ to ALF translator is developed within the ALL-TIMES project. It is based on the SATiRE framework [17], developed by the Compilers and Languages Group of the Institute of Computer Languages at TU Vienna. SATIrE integrates components of different program analysis tools and offers an infrastructure for building new code analysis tools. The C to ALF translator is available on the MELMAC web site [14]. The mapping of flow constraints back to the source code will use the Analysis Results Annotation Language (ARAL) format, which is a part of the SATiRE framework.

IAR intermediate code to ALF translator. This translator converts a proprietary intermediate code format, used by the embedded systems compiler vendor IAR Systems AB [8], into ALF.

Binary code to ALF translators. Two binary translators are currently under development at Mälardalen University, within the ALL-TIMES project. They translate CRL2 to ALF. CRL2 is a data format maintained by AbsInt GmbH [1], which is used by the aiT WCET analysis tool. CRL2 is mainly used to represent various types of assembler code formats with associated analysis results. By using CRL2 as source, the existing binary readers of aiT can be used to decode the binaries.

The first translator[1] translates CRL2's representation of NECV850E assembler code to ALF, and the second translator[2] converts PowerPC code into ALF. To allow SWEET's analysis results to be given back to CRL2, a mapping between CRL2's and ALF's different code and data constructs will be maintained. For both translators the generated flow constraints will be transferred to aiT using the AIS format, which is the annotation format used by aiT.

4.4. ALF interpreter

An ALF interpreter is currently being developed at Mälardalen University[3]. The interpreter will be used for debugging ALF programs (or, indirectly, translators generating ALF), and other purposes.

4.5. ALF file linking

An embedded systems project may involve a variety of code sources, including code generated from modelling tools, C or C++ code, or even assembler. To form a binary executable program, the source files are compiled and linked together with other object code files and libraries [11]. The latter are often not available in source code format [15].

To handle this issue ALF must support the "linking" of several ALF files into one single ALF file. Since ALF is supposed to support reverse engineering of binaries and object code, it cannot have an advanced module system with different namespaces. Instead ALF provides, similar to most object code formats [11], a set of exported symbols, i.e., `lrefs` or `frefs` visible to other ALF files, as well as a set of symbols that must be imported, i.e., `lrefs` or `frefs` used but not declared in the ALF file.

Forthcoming work includes the development of an *ALF linker*, i.e., a tool able to "link" ALF files generated from many different code sources. It will here be interesting to see if it is also possible to "link" (maybe partial) flow analysis results for individual ALF files to form a flow analysis result valid for the whole program.

5. Related Work

Generic intermediate languages have been used for other purposes. One example is the hardware description language TDL [9] which was designed with the goal to generate machine-dependent post-pass optimizers and analyzers from a concise specification of the target processor. TDL provides a generic modelling of irregular hardware constraints that are typical for many embedded processors. The part of TDL that is used to describe the semantics of instruction sets resembles ALF to some degree.

[1] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0875
[2] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0917
[3] http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0830

6. Conclusions and Future Work

ALF can be seen as a step towards an open framework where flow analyses can be available and freely used among several WCET analysis research groups and tool vendors. This framework will ease the comparison of different flow analysis methods, and it will facilitate the future development of powerful flow analysis methods for WCET analysis tools.

There may be other uses for ALF than supporting flow analysis for WCET analysis tools. For instance, ALF can be used as a generic representation for different binary formats. Thus, very generic tools for analysis, manipulation, and reverse engineering of binary code may be possible to build using ALF. Possible examples include generic binary readers that reconstruct control flow graphs, and tools that can reconstruct arithmetics for long operators implemented using instruction sets for shorter operators. The latter can be very useful when performing flow analysis for binaries compiled for small embedded processors, where the original arithmetics in the source code must be implemented using an instruction set for short operators.

It would be simple to define an XML syntax for ALF. We might do so in the future, in order to facilitate the use of standard tools for handling XML. Future work also includes further development of the PFF format for flow facts and the format for input value annotations.

Acknowledgment

This research was supported by the KK-foundation through grant 2005/0271, and the ALL-TIMES FP7 project, grant agreement no. 215068.

References

[1] ABSINT. aiT tool homepage, 2008. `www.absint.com/ait`.

[2] FERDINAND, C., HECKMANN, R., AND FRANZEN, B. Static memory and timing analysis of embedded systems code. In *3rd European Symposium on Verification and Validation of Software Systems (VVSS'07), Eindhoven, The Netherlands* (Mar. 2007), no. TUE Computer Science Reports 07-04, pp. 153–163.

[3] GUSTAFSSON, J., ERMEDAHL, A., AND LISPER, B. ALF (ARTIST2 Language for Flow Analysis specification. Tech. rep., Mälardalen University, Västerås, Sweden, Jan. 2009.

[4] GUSTAFSSON, J., LISPER, B., SANDBERG, C., AND BERMUDO, N. A tool for automatic flow analysis of C-programs for WCET calculation. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)* (Jan. 2003).

[5] GUSTAFSSON, J., LISPER, B., SCHORDAN, M., FERDINAND, C., GLIWA, P., JERSAK, M., AND BERNAT, G. ALL-TIMES - a European project on integrating timing technology. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods (ISOLA'08)* (Porto Sani, Greece, Oct. 2008), vol. 17 of *CCIS*, Springer, pp. 445–459.

[6] HOLSTI, N. Analysing switch-case tables by partial evaluation. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)* (2007).

[7] HOLSTI, N., AND SAARINEN, S. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'2002)* (2002).

[8] IAR Systems homepage.
URL: `http://www.iar.com`.

[9] KÄSTNER, D. TDL: A hardware description language for retargetable postpass optimizations and analyses. In *Proc. 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, Lecture Notes in Computer Science (LNCS) 2830. Springer-Verlag, 2003.

[10] KIRNER, R. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.

[11] LEVINE, J. *Linkers and Loaders*. Morgan Kaufmann, 2000. ISBN 1-55860-496-0.

[12] LISPER, B. Fully automatic, parametric worst-case execution time analysis. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)* (Porto, July 2003), J. Gustafsson, Ed., pp. 77–80.

[13] MÄLARDALEN UNIVERSITY. WCET project homepage, 2008. `www.mrtc.mdh.se/projects/wcet`.

[14] MELMAC. MELMAC homepage, 2009.
`http://www.complang.tuwien.ac.at/gergo/melmac`.

[15] MONTAG, P., GOERZIG, S., AND LEVI, P. Challenges of timing verification tools in the automotive domain. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[16] PRANTL, A., SCHORDAN, M., AND KNOOP, J. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET'2008)* (Prague, Czech Republic, July 2008), R. Kirner, Ed., pp. 141–148.

[17] SATIRE. SATIrE homepage, 2009.
`http://www.complang.tuwien.ac.at/markus/satire`.

[18] SEHLBERG, D., ERMEDAHL, A., GUSTAFSSON, J., LISPER, B., AND WIEGRATZ, S. Static WCET analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[19] THEILING, H. *Control Flow Graphs For Real-Time Systems Analysis*. PhD thesis, University of Saarland, 2002.

[20] WRIGHT, A. K., AND CARTWRIGHT, R. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst. 19*, 1 (Jan. 1997), 87–152.