Program Semantics in Model-Based WCET Analysis: A State of the Art Perspective*

Mihail Asavoae, Claire Maiza, and Pascal Raymond

Laboratoire Verimag
Centre Equation, 2 Avenue de Vignate, Gieres, France
{Mihail.Asavoae,Claire.Maiza,Pascal.Raymond}@imag.fr

Abstract

Advanced design techniques of safety-critical applications use specialized development model-based methods. Under this setting, the application exists at several levels of description, as the result of a sequence of transformations. On the positive side, the application is developed in a systematic way, while on the negative side, its high-level semantics may be obfuscated when represented at the lower levels. The application should provide certain functional and non-functional guarantees. When the application is a hard real-time program, such guarantees could be deadlines, thus making the computation of worst-case execution time (WCET) bounds mandatory. This paper overviews, in the context of WCET analysis, what are the existing techniques to extract, express and exploit the program semantics along the model-based development workflow.

1998 ACM Subject Classification B.8.2 Performance Analysis and Design Aids

Keywords and phrases Survey, WCET analysis, Program semantics, Model-based design, Infeasible paths

Digital Object Identifier 10.4230/OASIcs.WCET.2013.32

1 Introduction

Programming embedded and hard real-time systems requires careful considerations not only with respect to correctness criteria of the software product, but also to resource utilization (i.e. memory usage, power consumption or timing behavior). This implies to build the embedded and real-time applications in a systematic way and thus, to set the grounds for subsequent development of analysis tools. The worst-case execution time (WCET) analysis provides safe guarantees w.r.t. the timing behavior of hard real-time applications.

A popular solution in the direction of software systematization is called *model-based design* (MBD) and presents, in general, three components: a high-level specification language to develop the application (which is also called *model*); compilation support to further process the model; tool support for simulation (and in some cases analysis) purposes. We restrict our discussion on a particular MBD workflow which is currently used in both avionics and automotive domains and where the compilation support generates classical imperative code and then binary code as shown in Figure 1(C). In this setting, the application semantics is present in the high-level model, in the intermediate program and in the binary code. Apart from the semantics representation, the model-based workflow also has two semantics transfer levels: from model to imperative code and from imperative code to binary code. We discuss

* This work was partially supported by ANR under grant ANR-12-INSE-0001 and by EU COST Action IC1202:Timing Analysis On Code-Level (TACLe).

13th International Workshop on Worst-Case Execution Time Analysis (WCET 2013).
Editor: Claire Maiza; pp. 32–41

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

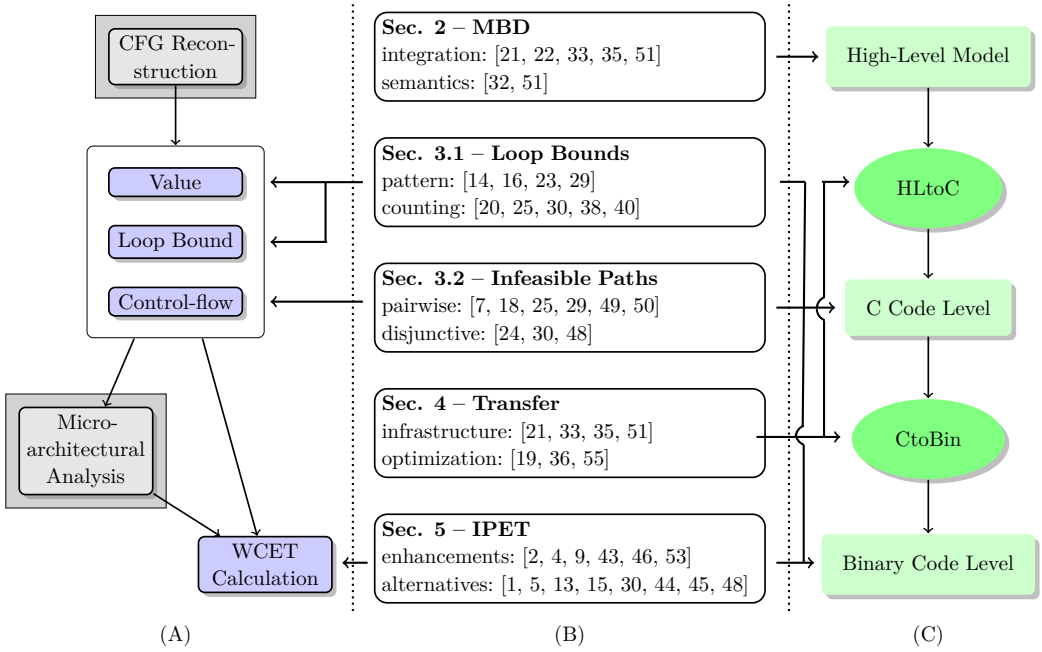

Figure 1 Static WCET analysis workflow (A), paper organization (B) and MBD workflow (C).

how the program semantics is expressed, extracted and exploited in such a workflow, when the analysis of interest computes WCET bounds.

The WCET analysis of a particular program is performed at the binary level and with knowledge about the underlying architecture. As it is summarized in [56] and shown in Figure 1(A), a typical workflow for a WCET analysis proceeds with the CFG extraction, a number of program flow- and processor-behavior analyses, and finally, the bound computation. The WCET analysis should provide safe and tight estimations of the actual WCET of a program. To address these, the WCET analysis workflow relies on a number of specific analyses, spawn from both the flow analysis (i.e. detection of loop bounds and infeasible paths) and the architecture analysis (i.e. cache and pipeline behavior prediction).

In this paper we present a survey study on how the WCET analysis workflow is projected on the MBD workflow, from a particular point of view – the separation of concerns at the level of program semantics manipulation. Due to the generality of the MBD framework and the multitude of contributions in the WCET analysis, as well as the current space limit, we restrict our presentation under a setting defined by the following constraints. First, we consider MBDs where the model is compiled into C code. Second, we consider the architecture-related analyses to be orthogonal to our investigation on the program semantics and thus are left out, as shown in Figure 1(A). This second restriction activates other intended omissions, from our survey: analyses for CFG extraction and for classification of load/store instructions. Third, we discuss the path-analysis problem from the popular implicit path-enumeration technique (IPET) point of view, classifying the approaches as enhancements or alternatives to it.

The works in [37, 56] survey, from certain angles, the state-of-the-art approaches in the WCET analysis field of research. The authors of [37] rely on the notion of the flow fact and classify then-existing WCET analysis approaches w.r.t. this notion. As a consequence, this allows comparisons between various approaches at the confluence of axis for the representation levels and the execution-time modeling. The survey in [56] is ampler and newer than [37],

covering both the methods and the existing tools in the WCET analysis field of research. The methods are classified into static and measurement-based ones, with the information presented at the level of WCET analysis subtasks. What we propose here is a specialized view, in the form of a separation of concerns from the program semantics perspective. In comparison, this paper is different because it covers (1) an up-to-date specialization of the representation levels axis from [37] (i.e. only the semantics levels in the model-based development frameworks with generation of C code capabilities) (2) an up-to-date specialization to the static-based methods from [56], in particular to the flow analyses, specialization which is presented on (3) a projection of the workflow of the model-based development frameworks. To summarize, our survey follows the organization of a modern software development framework for embedded and real-time applications and presents up-to-date works, exclusively from a program semantics perspective.

The organization of this paper follows the projection of WCET analysis workflow over the levels of the model-based development frameworks – Figure 1(B). We present in Section 2 the model-based frameworks as well as the current approaches towards WCET analysis on the general setting. In Section 3, we project the flow analysis of interest (i.e. loop-bound and infeasible path detection) at each programming language level in the development frameworks, while in Section 4 we discuss how to transfer information. We dedicate Section 5 to the path analysis problem, then we draw conclusions and discuss open problems, in Section 6.

2 Model-Based Development Framework

The development of embedded real-time applications using MBDs [11] gained popularity in the last decade. The key element lays in the design environment – using a high-level specification language with mathematical background and graphical support, which enables rapid prototyping and a high level of design reusability. Moreover, a MBD tool provides controller analysis and synthesis, as well as deployment support. Application development in the automotive [10, 47], avionics [54] and aerospace [31] domains rely on popular tools like Scade Suite and Matlab Simulink/Stateflow. More precisely, Scade belongs to the synchronous languages family, which means it was designed to generate code, while Matlab served initally as a simulation tool, but it is now equipped with code generation facilities.

The synchronous paradigm is a deterministic parallel programming style, which compiles synchronous programs into classical sequential C code with bounded loops and memory usage – and for which it is mandatory to find a WCET bound. Synchronous programming languages can be classified into data-flow oriented (e.g. Lustre), control-flow oriented (e.g. Esterel) and mixed approaches (e.g. Scade Suite). The Lustre programming language and its formal semantics are presented in [26]. Semantically, a Lustre program transforms input streams of values into output streams of values and structurally, it represents a system of equations defining the variables (these are functions from time domain to value domain). The Esterel programming language and its formal semantics are presented in [6]. Semantically, an Esterel program reacts to input events (i.e. signals) by producing output signals, and structurally, it consists of specialized imperative statements to specify control operations (i.e. delay, signal emission, abortion etc). Both the Lustre and Esterel compilers [27, 17] generate sequential C code from the intrinsically-parallel synchronous program. The mixed approaches, represented by Scade Suite and Matlab Simulink/Stateflow provide powerful modeling languages to integrate data and control-related aspects. For data- and control-flow parts of the application, Scade uses Lustre and respectively Safe State Machines (graphical equivalent to Esterel), while Matlab uses the languages Simulink and respectively Stateflow.

The safety-critical applications, which are developed through MBDs, often require guarantees about their timing behavior. Particularly, the idealized "instantaneous" synchronous tick is implemented as a guarantee of an upper bound on the execution time. Therefore, it is required to integrate the WCET analysis techniques into the MBD workflow. There, we state the following two types of contributions: *integration methods* (w.r.t. the timing analyzer) and *semantics-specific methods* (w.r.t. the model or other program representation-level).

Integration methods. The integration methods simply embed the timing analyzer in the MBD workflow as tool support in the application development process. Timing analyzers are integrated in Scade Suite workflow [21] (using the existing traceability properties of the MBD), in Matlab Simulink [35, 51], in an Esterel-driven MBD [33] and in an automotive-specific model-based development framework, called Ascet [22].

Semantics-specific methods. The semantics-specific methods focus on the precision of the WCET analysis, transporting program semantics properties from the model level to the binary level. Existing approaches investigate the timing behavior of Matlab Simulink/Stateflow models [51] as well as of Esterel applications (during one tick and along multiple ticks) [32]. The program semantics information materializes into various path pattern types [32] or entailment relations and flow constraints [51]. The works in [8, 3] propose standalone timing analyses of synchronous programs, without a complete integration into the WCET analysis workflow. Both perform timing analyses of Esterel code which is executed during one tick, and also called worst case reaction time (WCRT) analyses.

3 Representation Level – Language

The model-based design presents several levels of program representation. In this paper, we consider those design platforms with C code as their representation language between the model and the binary level. For the WCET analysis techniques, it is (1) convenient to work on the low-level representation because of the architecture-related information, and (2) inconvenient because of the obfuscated program structure (and possibly the semantics), due to compilation influence. The MBD frameworks open the possibility to manipulate the program semantics at a convenient level. There are two complementary methods to obtain the program semantics properties of interest: add *manual annotations* or extract them using *dedicated analyses*. We briefly cover the former and then, elaborate on the latter.

Manual methods. In general, the MBD workflow provides annotation support through intermediate generated files (e.g. for Scade Suite models [51]) and it handles information about code locality. Nevertheless, it is possible to extend the given MBD workflow to accommodate specific WCET analysis annotations (e.g. for Matlab Simulink/Stateflow models [35]). For a more general view on existing annotation languages and tool support in the WCET analysis domain, we recommend the comprehensive survey in [34].

Automated methods. We focus on the following two subtasks performed on the CFG representation of the program, the loop bound analysis and the control-flow analysis (with the infeasible path detection). Most of the existing solutions work at the binary level.

3.1 Loop Bounds Detection

The model-based design framework should generate certified C code, which implies it to be deterministic and traceable, without dynamic allocation, with checked dynamic accesses etc.

As a result, such C code contains only bounded loops, usually in the form of for-statements. However, complex models could produce preemptive conditions to break out of the loops, making the initial loop bound a grossly overestimation of the actual number of iterations. The WCET analysis requires knowledge of both loop and recursion bounds. Therefore, a loop bound analysis attempts to automatically infer such bounds, or in other words to discover inductive invariants over loop counters. The general procedure consists of three steps: express how loop variables change across iterations, solve the resulting expressions (i.e. obtain their closed form) and finally, project the results over the loop counters [41]. Because this general procedure is undecidable, the state-of-the-art approaches rely on pattern-based heuristics at the level of loop structure and/or loop data. We classify the loop bound analyses w.r.t. whether they employ *pattern-matching* or *counting*, and further projected on how the loop bound is expressed.

Pattern-based methods. We consider the loop bound analyses which use patterns at the level of code constructs [29, 16, 23] and/or encode the closed forms expressions of interest [16]. The code-centered analyses considers specific patterns of loop tests (e.g. comparisons of variables to constant values) and captures how loop variables change across iterations through data-flow analysis [29, 23, 14] or abstract interpretation [16]. The property-centered analyses uses pattern matching in a different way: the results of an abstract interpretation-based analysis match user-defined patterns representing closed forms expressions [16]. The results of these loop bound analyses are expressed as summations [29], intervals [23, 14] or both [16].

Counting-based methods. We consider loop bound analyses which symbolically accumulates knowledge about (i.e. count) the number of loop iterations. The key element of such an analysis is the loop counter – a program or an analysis-specific variable. More specifically, the loop counter can be: a symbolic variable with an interval domain [25, 40], a number of program states (modulo equivalence classes) [20], a Presburger set-representation of a symbolic variable [30], a parameter of a recurrence equation [38] or a formula [12]. The counting methods are: derivations from abstract interpretation (i.e. abstract execution) [25], combinations of abstract interpretation with other methods (i.e. program slicing) [40, 20] or SMT-based invariant generators [38]. The results of these loop bound analyses are expressed as summations [38], intervals [25, 40, 20] and those of [30] could be disjoint sets of values (i.e. specific bounds or intervals).

3.2 Infeasible Paths Detection

The ability to detect infeasible execution paths greatly influences the precision of a WCET analysis. The sequence of instructions which define a program execution may be characterized by the sequence of decisions taken at conditional statements. A program execution is infeasible when it cannot be exercised, regardless of the input data. A common way to identify the infeasibility is to detect conflicting pairs of conditional statements. Note that, in the context of WCET analysis, the infeasibility expressed as conflicting pairs hides a more practical aspect – it could be easily encoded in the popular IPET formulation of the path analysis. Nevertheless, there exists a conceptually orthogonal approach which is capable to detect more expressive transition (e.g. disjunctive) invariants. The application development through MBDs produces code with infeasible paths, coming from multiple sources: the model semantics, the specificities of the high-level language or the code generation techniques. For example, the control-flow aspects (e.g. specialized instructions or finite state machines) of the high-level language are translated into conflicting tests (to isolate impossible behaviors). Also, the underlying scheduling mechanism to generate deterministic C code could produce

repeating tests. Next, we classify the infeasible path detection analyses w.r.t. the result type: *conflict-pairs invariants* and *transition (e.g. disjunctive) invariants*.

Conflict-pair invariants methods. The methods to detect conflicting conditional statements are, in general, based on abstract interpretation methods [29, 18, 25, 7, 49], but search-based techniques are possible [50]. The general workflow has a value analysis phase, which defines possible values for program variables, and an extraction phase to produce relations between program statements (i.e. test-test or assignment-test). A number of specialized techniques aims at increasing the precision of the analyses: program slicing techniques [25, 49] or symbolic propagation [7]. The techniques in [25, 29, 50] discover infeasible paths in the context of the IPET technique. Moreover, these analyses are performed at the binary level, in the MBD hierarchy.

Disjunctive invariants methods. The methods to discover more infeasible paths use, either an expressive flow facts encoding (i.e. Presburger sets) [30], or techniques to expose the infeasibility [48, 24], via unfolding the set of program paths. This set is represented as a graph [48] (and explored with path pruning and graph refining techniques) or as a regular expression [24] (and explored with static analysis techniques). All these approaches [48, 24, 30] discover disjunctive invariants, which w.r.t. the dominant IPET technique (in the WCET analysis community), are difficult to express/exploit.

4 Representation Level – Transfer

We investigate how a particular MBD workflow with two levels of information transfer integrates the WCET analysis workflow. In general, transferring timing specific information (i.e. annotated or computed flow facts) from the high-level language to the imperative code level is well studied. The MBDs are supported by compilers which generate C code in a systematic way [27, 17, 52] and therefore offer good traceability information (e.g. the *KCG* compiler of Scade [52]). However, the second transfer level, from C to the binary level is more tricky because the general-purpose compilers such as *gcc*, feature code and data optimizations which affect the traceability. There are several classifications of traceability: depending on the direction of semantics transfer (i.e. forward and backward), modifications of the underlying tool support (i.e. deep and surface) or the presence of compiler optimizations [55]. We elaborate next on the classification based on the required infrastructure modification, then we overview existing approaches for traceability through compiler optimizations.

Infrastructure-based classification. Several approaches, e.g for Scade Suite [21, 51] rely on the available traceability information to integrate a timing analyzer into the MBD workflow. From an implementation point of view, traceability through annotations does not require modifications of the underlying structures. The Scade workflow uses XML files to transfer program location-based annotations. MBDs like Matlab Simulink [35] or Esterel [33] use modified infrastructure to improve the existing traceability. This type of traceability is achieved through the code structure and addresses the needs to transfer scope-based flow facts (e.g. loop bounds). Another application is to reconstruct the longest path returned by the timing analyzer, building a forward traceability chain as annotated ASTs [33].

Optimization-based classification. A more difficult problem is to transfer the flow facts for the WCET analysis, through compiler optimizations. The general strategy is to identify classes of optimizations and to model, case by case, the code transformations implied by the optimizations. The existing approaches [19, 36] require specialized languages to express flow

facts and their transformation. To integrate these languages, the compiler is either directly modified [19] or wrapped and manipulated by additional software infrastructure [36].

5 Path Analysis

The WCET analysis produces the timing bound (i.e. the longest execution path) after a path analysis phase. The longest path search implies that all the execution paths should be considered and it requires an underlying semantics model of them. We enumerate: control flow graph [39], abstract syntax tree [13], Kripke structure [42], timed automaton [15]. Nevertheless, an implicit path enumeration technique (IPET) formulation of the path analysis is the most popular approach for WCET analysis. We classify the path analyses into IPET-based algorithms (including enhancements of the original technique) and alternative approaches (syntax-directed schema, model checking or graph transformation).

IPET. The control flow graph (CFG) captures, in a compact way, the flow of the particular program, abstracting away data aspects. While there are several ways to represent the CFG, the WCET analysis considers the nodes as basic blocks (single-entry single-exit sequences of statements). The basic block representation of the CFG is used to encode the program paths as an ILP problem, and perform path analysis as ILP solving. The approach is called the implicit path enumeration technique (IPET) [39]. The ILP problem consists of two kinds of constraints: structural (or flow) constraints, to express input-output flow relations for basic blocks, and functional constraints, to handle loops (i.e. given as loop bounds) and to improve the precision (i.e. encode infeasible path). Specialized techniques extract ILP constraints from the CFG [18] or from other graph-based representations of the program [46, 53].

IPET enhancements. The overall method for path analysis through ILP solving suffers from two drawbacks: the timing bounds are as precise as the quality of the functional constraints and the size of the ILP problem directly affects the computation time of the results. Solutions for the former are presented in Section 3; next we focus on existing techniques to solve the ILP problem more efficiently. There are three complementary techniques: modular solving [4], problem size reducing [43] and parametric analysis [2, 9]. A modular solution identifies ILP sub-problems as CFG regions with single-entry single-exit properties, for which the locally computed results replaces the region. The size-reducing solution uses CFG transformations to combine conditionals and to reduce the number of program paths. The parametric solution produces, after specific analyses (for parametric dependencies between program variables and parametric expressions for loop bounds etc), a symbolic ILP problem.

IPET alternatives. Different path analysis methods project the representation of all program paths on syntactic and semantics artifacts. We enumerate the following solutions: syntax-directed [13, 44, 45], path-based [48], state-based [42, 15], graph-based [1] and special annotations [5, 30]. A syntax-directed approach uses timing schema for programming constructs and the path exploration is the AST traversal. A path-based WCET computation searches for the longest path among the previously computed bounds for different program paths. A model-based approach uses representations for the program states and the path exploration is performed with model checking techniques. A graph-based approach applies graph algorithmics to transform and/or traverse the CFG of the program. The annotation-based technique, which bridge the gap between the program semantics and the timing model [5, 30], transforms the path analysis into (one or more) constraint solving problems. With respect to the MBD, the path analysis is represented outside the MBD workflow, as an external procedure.

6 Concluding Discussions

This paper offers a broad view on a series of techniques for WCET analysis, with an emphasis on how the program semantics is manipulated. Moreover, this paper advocates for a separation of concerns at the level of program semantics – i.e. the separation into extract, express and exploit phases. We distinguish the following two directions of interest. First, in the context of the MBD workflow, it is necessary to manipulate the program semantics at various levels in the design chain. As such, the WCET analyzers would not only be integrated into MBDs [21, 35], but produce tighter results [51, 33], based on the model semantics. Second, the IPET technique dominates many approaches for WCET analysis, and, as such, many specializations of WCET analysis subtasks exist. The detection of loop bounds [16, 28] or infeasible paths [18, 29, 50] generate flow facts which are directly expressible into integer linear programming. As such, WCET analysis benefits from alternative approaches on flow facts generation [48, 24] or path analysis [30, 1].

Design and implement applications for embedded and hard real-time systems have several benefits when using a model-based design environment. First, the high-level language permits modular development, has formal semantics and capability for imperative code generation. From a WCET analysis perspective, (1) it allows annotations at the design level as opposed to cumbersome instrumentation at the low-level and (2) the resulting imperative code features good traceability because it adheres to certain certification criteria. Second, because the imperative code is systematically constructed, it opens new possibilities to apply specific discovery, express and transfer the flow properties to the WCET analysis level – the binary code. Third, the gap between the high-level driven MBD workflow and the low-level driven WCET analysis workflow requires a bi-directional transfer of the program semantics.

References

1 E. Althaus, S. Altmeyer, and R. Naujoks. Precise and efficient parametric path analysis. In *LCTES*, 2011.

2 S. Altmeyer, C. Humbert, B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *RTCSA*, pages 367–376, 2008.

3 S. Andalam, P. Roop, and A. Girault. Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs. In *DATE*, 2011.

4 C. Ballabriga and H. Cassé. Improving the wcet computation time by ipet using control flow graph partitioning. In *WCET*, 2008.

5 G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *RTP*, 2000.

6 G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program. (SCP)*, 19(2):87–152, 1992.

7 R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI*, pages 146–158, 1997.

8 M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *ENTCS*, 203(4):65–79, June 2008.

9 S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. *Journal of Systems Architecture*, pages 614–624, 2011.

10 S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper. Applying static wcet analysis to automotive communication software. In *ECRTS*, pages 249–258, 2005.

11 P. Caspi, P. Raymond, and S. Tripakis. Synchronous languages. In *Handbook of Real-Time And Embedded Systems*. Chapman and Hall, 2007.

12 J. Coffman, C. Healy, F. Mueller, and D. Whalley. Generalizing parametric timing analysis. In *LCTES*, pages 152–154, 2007.

13 A. Colin and I. Puaut. A modular & retargetable framework for tree-based wcet analysis. In *ECRTS*, pages 37–44, 2001.

14 C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *WCET*, 2007.

15 A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *WCET*, pages 113–123, 2010.

16 M. de Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *RTCSA*, 2008.

17 S. A. Edwards. Compiling esterel into sequential code. In *DAC*, pages 322–327, 2000.

18 J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, pages 163–174, 2000.

19 J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution times analysis for optimized code. In *ECRTS*, 1998.

20 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

21 C. Ferdinand, R. Heckmann, T.L. Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In *ERTS2*, 2008.

22 C. Ferdinand, R. Heckmann, H.-J. Wolff, C. Renz, O. Parshin, and R. Wilhelm. Towards model-driven development of hard real-time systems. In *AASSD*, 2006.

23 C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New developments in wcet analysis. In *Program Analysis and Compilation*, pages 12–52, 2006.

24 S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

25 J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, 2006.

26 N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

27 N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *PLILP*, pages 207–218, 1991.

28 C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loops iterations. *RTS*, 18(2-3), May 2000.

29 C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), August 2002.

30 N. Holsti. Computing time as a program variable: a way around infeasible paths. In *WCET*, 2008.

31 N. Holsti, T. Långbacka, and A. Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. In *DASIA*, 2000.

32 L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury. Context-sensitive timing analysis of esterel programs. In *DAC*, pages 870–873, 2009.

33 L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty. Performance debugging of esterel specifications. In *CODES-ISSS*, 2008.

34 R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Journal on Software and System Modeling*, 10(3), 2011.

35 R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *ECRTS*, 2002.

36 R. Kirner, P. Puschner, and A. Prantl. Transforming flow information during code optimization for timing analysis. *Journal on Real-Time Systems*, 45(1-2), 2010.

37 R. Kirner and P. P. Puschner. Classification of wcet analysis techniques. In *ISORC*, pages 190–199, 2005.

38 J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for wcet analysis. In *PSI*, pages 227–242, 2012.

39 Y.-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(12), 1997.

40 P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, 2009.

41 F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In *CC*, 1998.

42 A. Metzner. Why model checking can improve wcet analysis. In *CAV*, pages 334–347, 2004.

43 H. S. Negi, A. Roychoudhury, and T. Mitra. Simplifying wcet analysis by code transformations. In *WCET*, 2004.

44 C. Y. Park and A. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, 24:48–57, 1991.

45 P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst. (RTS)*, 1(2):159–176, September 1989.

46 P. Puschner and A. Schedl. Computing maximum task execution times - a graph-based approach. *Real-Time Systems*, 13(1), 1997.

47 D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz. Static wcet analysis of real-time task-oriented code in vehicle control systems. In *ISoLA*, pages 212–219, 2006.

48 F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES*, 2001.

49 I. Stein and F. Martin. Analysis of path exclusion at the machine code level. In *WCET*, 2007.

50 V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC*, pages 358–363, 2006.

51 L. Tan, B. Wachter, P. Lucas, and R. Wilhelm. Improving timing analysis for Matlab Simulink/Stateflow. In *ACES-MB*, 2009.

52 Esterel Technologies. *Scade Language Reference Manual*, 2011.

53 H. Theiling. Ilp-based interprocedural path analysis. In *EMSOFT*, pages 349–363, 2002.

54 S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, pages 625–632, 2003.

55 A. Vrchoticky. Compilation support for fine-grained execution time analysis. In *LCTES*, 1994.

56 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.