Chapter 2
Testing Implementation Soundness of a WCET Analysis Tool

Reinhard Wilhelm, Markus Pister, Gernot Gebhard, and Daniel Kästner

2.1 Introduction

Timing verification of a set of hard real-time tasks to be executed on a given hardware platform attempts to prove that all tasks in the set when executed on that platform always respect their deadlines, i.e., each task finishes its execution within its deadline. Traditionally, timing verification is split into two subtasks: a *timing analysis* also known as *WCET analysis*, which statically determines upper bounds on the execution times of the tasks, and a *schedulability analysis*, which takes these upper bounds and attempts to verify that all tasks in the given set, assuming these upper bounds on their execution times, will respect their deadlines.

A preliminary version of this paper appeared in [16].

R. Wilhelm (✉)
Universität des Saarbrücken, Saarbrücken, Germany
e-mail: wilhelm@cs.uni-saarland.de

M. Pister · G. Gebhard · D. Kästner
AbsInt Angewandte Informatik GmbH, Saarbrücken, Germany
e-mail: pister@absint.com; gebhard@absint.com; kaestner@absint.com

J.-J. Chen (ed.), *A Journey of Embedded and Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-47487-4_2

2.1.1 Tool Qualification

WCET analysis is applied to time-critical and safety-critical embedded-system software in problem-aware parts of the embedded-systems industry. Such systems have to be developed in accordance with international safety norms, e.g., DO-178B/C, DO-254, IEC 61508, and ISO 26262. While there are differences between these norms, in particular regarding prescriptiveness and required level of rigor, they have many aspects in common. All of them include guidance on the use of software tools as a part of the development and verification process of safety-critical software.

The criticality level (also known as the design assurance level (DAL) or safety integrity level (SIL)) of a component determines the effort to invest and the methods required or recommended to deliver assurance of the correct functioning of the component. The criticality level is derived from the impact of a failure of the component on the functioning of the system. Similarly, the required activities to provide confidence in the correct functioning of a software tool depend on its criticality with respect to the overall system. For example, DO-178C, the current international standard for avionics systems, defines five different *tool qualification levels (TQLs)*. The TQL is determined by the potential tool impact and the design assurance level of the software. There are three tool-impact categories; the most critical, *Category 1*, applies to tools whose output becomes part of the airborne software. Similar considerations are also made in other norms, e.g., the ISO 26262 defines a *tool confidence level (TCL)* in a very similar way.

The overall goal of *tool qualification* is to provide confidence that the tool operates correctly, i.e., according to its functional specification, in the operational context of the tool user. In the following, we will focus on the tool qualification requirements of the avionics industry, which are the most rigid of the safety-critical industries. Certification of avionics systems is regulated by the international standard DO-178C [1]. WCET analysis tools fare under *verification tools*. Verification tools have no overly rigid certification requirements, unlike *development tools*: their impact category is *Category 2* or *Category 3*, mostly depending on whether the output of the tool is used to justify the elimination or reduction of other verification or development activities or not. A prerequisite for tool qualification is a specification of the tool functionality. The *tool operational requirements (TOR)* specify the tool functions and technical features, which are stated as low-level requirements on tool behavior under normal operating conditions. Another required input is the *verification test plan (VTP)*, which defines test cases demonstrating the correct functioning of all specified requirements of the TOR. Test-case definitions include the overall test setup as well as a detailed structural and functional description of each test case, i.e., how the individual test case works and what the expected result is.

Certification becomes more challenging through DO-333, the formal-methods supplement to DO-178C. It asks for a statement that a formal method including the underlying theory is *adequate* for solving the corresponding verification problem. This introduces and enforces *soundness* of the methods and tools.

Since the required effort for tool qualification can be high, ideally the software qualification process is supported by a *qualification support kit (QSK)* supplied by the tool provider. It must include TOR and VTP and typically provides a validation suite, which allows users to execute the relevant test cases in the relevant operational context. TOR, VTP, and a test execution report become part of the certification package. Furthermore, it is typically required for a tool provider to supply *qualification software life cycle data* to demonstrate that the development process and the invested efforts to assure correctness, quality, and traceability are adequate for usage in a safety-critical system context. The qualification software life cycle is not covered in this article.

DO-178C exhales a test-based spirit: many verification activities are test based. Well-defined coverage criteria try to capture to which extent the behavior of the system under test has actually been exerted during testing. Note that in case of a static verification tool, test coverage does not apply to the code to be analyzed: a sound static-analysis tool provides full data and control coverage, i.e., it analyses all paths and takes into account all potential data values for its analyses. What is needed in case of the microarchitectural analysis, which is the focus of this article, is to demonstrate the correctness of the microarchitecture model used by the analyzer. To this end it is the instruction set architecture (ISA) and the set of paths through the execution platform that need to be covered. Huge sets of test traces in qualification suites are used at tool-qualification time to cover the sets of paths through the execution platform.

Note the difference to measurement-based WCET analyses. It is known that they are in general unsound. In order to provide a sufficient level of confidence in the real-time behavior of industrial-size code they need an unacceptably huge set of traces and accordingly an excessive effort at verification time. In the case of a static WCET analysis tool, the testing effort is applied at tool-qualification time when ample time is available.

2.1.2 Predictability

Timing predictability [3, 15] has long been recognized as essential for achieving precise results of timing estimation at reduced analysis effort. In the context of the current article, it is worth mentioning that it also reduces the number of test cases for the validation of an abstract architectural model. In general, an increase in the timing predictability of the underlying architecture leads to a decreasing number of different instruction flow paths through the processor pipeline since they feature less average-case performance-enhancing micro-optimizations like instruction and data queues and buffers, data forwards, etc. Such architectures show a more regular hardware design.

2.1.3 WCET Analysis

Performance-enhancing architectural components such as caches, pipelines, and speculation have made WCET analysis difficult. Execution times of consecutively executed instructions do not compose easily because instruction execution times are now dependent on the execution state in which they are executed. In the composition *A;B* the execution time of statement *B* depends on the execution state produced by executing statement *A*. The variability of execution times grows with several architectural parameters, e.g., the cache-miss penalty and the costs for pipeline stalls and for control-flow mispredictions. As approaches using exhaustive measurements are infeasible due to the size of the search space, abstraction is applied leading to an over-approximation of the set of potential executions. This over-approximation introduces remaining uncertainty in the results of the microarchitectural analysis, which grows with the same architectural parameters mentioned above unless the architectural platform is predictable [18], see Sect. 2.1.2.

2.1.4 The Central Idea: Proving Safety Properties

We needed to solve the WCET problem for architectures with state-dependent execution times. Figure 2.1 shows that this problem could be decomposed into many subproblems. The main problem, specific for WCET analysis, was the *microarchitectural analysis*, a combined cache and pipeline analysis. Let us describe the central idea behind this phase in our WCET analysis method [17], first in a conceptual way, i.e., not quite like it is implemented, later closer to how it is implemented:

- We define any architectural effect that causes an instruction to execute longer than its fastest execution time to be a *timing accident*. Typical such timing accidents are cache misses, pipeline stalls, bus-access conflicts, or branch mispredictions. Each timing accident is associated with a *timing penalty*. Timing penalties may be constant, but may also be execution-state dependent. A cache-miss penalty may be constant if the bus is always guaranteed to be free for the cache reload. If this guarantee cannot be given, however, its size depends on the execution state, namely whether the bus happens to be free.

  The property that the execution of an instruction at some program point will not cause a particular timing accident is then a safety property. The occurrence of a timing accident thus violates a corresponding safety property.
- We then use an appropriate method for the verification of safety properties to prove that for the instructions in the program some of the potential timing accidents will never happen. The goal is to prove as many of such safety properties as possible. Conceptually, the safety properties shown to hold could be used to reduce the worst-case execution-time bound for an instruction, which a naive, sound WCET analysis would have to assume, by the cost for the excluded

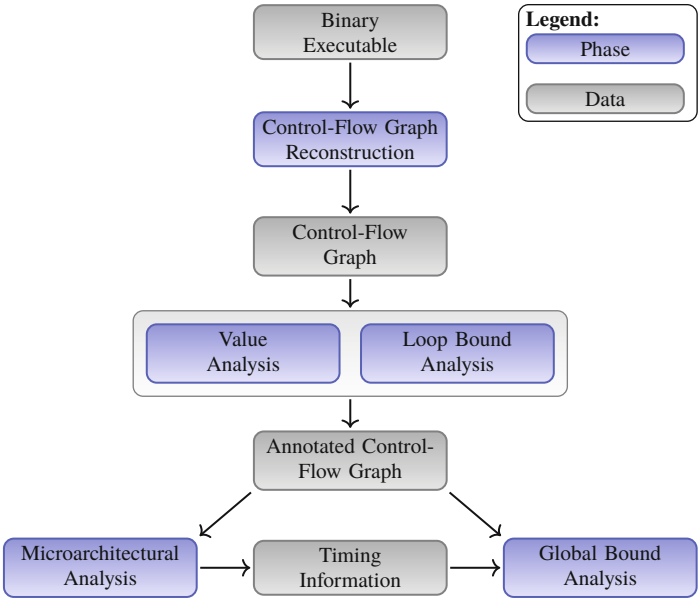

Fig. 2.1 The architecture of the aiT tool

timing accidents. In practice, pipeline analysis drives a cycle-wise transition, which considers the abstract execution state, e.g., makes no transition under a cache miss if a cache miss can be excluded.

- We then prove these safety properties by abstract interpretation (AI) [4] in the following way: Compute invariants at each program point, in our case an over-approximation of the set of execution states that are possible when execution reaches this program point. Derive the above mentioned safety properties, that certain timing accidents will not happen, from these invariants. For example, AI computes an abstract cache state at each program point, which overapproximates the sets of concrete cache states that may reach this program point. The abstract cache states are used to classify some memory accesses as definite hits. Another cache analysis that underapproximates the set of possible concrete cache states is able to predict definite misses. Predicted cache hits are then used to prove that the timing accident, this memory access will miss the cache, will never happen [8, 10].

  This method for the microarchitectural analysis was the main innovation that made our WCET analysis work for real-life architectures and scale to industrial-size software [6].

Now follows the description of the microarchitectural analysis that is closer to the implementation. Driver of this analysis is the pipeline analysis [14]. It goes through the instruction stream, instruction by instruction, and executes the current instruction in the current abstract execution state. This abstract execution

state contains uncertainty, i.e., it lacks information about some state components. Transitions to all potential successor states are performed whenever the transition to the next state depends on such a missing part of the state. The timing contributions of these transitions are accumulated until an instruction can be retired. In the end, upper bounds on the execution times of basic blocks are obtained that are coefficients in an integer linear program representing the control flow of the program [17]. Another type of result is described below.

2.1.5 Terminology

We consider only sound WCET analysis methods. *Soundness* means that a method and associated tool will always produce *conservative* WCET estimates, i.e., estimates that will never be exceeded in any execution. Being conservative is a Boolean property. Unfortunately, *conservative* is often used as a metric property, *more conservative* meaning *less precise*. However, calling results of an unsound method conservative is a misnomer. The really meant, other dimension, in addition to soundness, is *accuracy*. Accuracy of some WCET estimate, obtained by a sound method, expresses the degree of over-estimation, the difference between a WCET estimate and the real WCET. It does not make sense to talk about the accuracy of an unsafe estimate or an unsound method. In case of an unsound method it is not even clear whether a "more conservative" estimate moves towards the real WCET from below or is larger than the real WCET and moves further away from it. In general, WCET estimates are below, i.e., underestimate the real WCET, if end-to-end measurements are used. On the other hand, if piecewise measurements are applied whose results are combined to an estimate of the overall execution times, this often results in over-estimation of the real WCET.

WCET analysis can be seen as the search for a longest path in the state space spanned by the program under analysis and by the architectural platform. Most real-time software is written as to guarantee termination. Its state space can thus be easily abstracted to a finite abstract state space, which is still too large to be exhaustively explored. We can, therefore, not expect to identify the real WCET, but only safe upper bounds to all execution times, which we will call WCET estimates. (Safe) over-approximation is used in several places. In particular, an abstraction of the execution platform is employed by the WCET analysis. How to convince oneself (or the certification authorities) of the correctness of this architectural model is the main subject of the next section.

2.2 Validation of Our WCET Analysis Tool

The claim that our WCET analysis tools produce safe results is a strong one and often disputed by some proponents of unsound WCET analysis methods. Their

argument is, to develop an error-free instantiation of the, in principle, sound WCET analysis technology is so difficult, that one might use a simpler unsound method in the first place. The main complaint is the complexity of the abstract architectural models. So, what is the basis for our claims?

Several analyses in the tools are instances of abstract interpretation [4], a scientific method with a strong underlying theory, relating analysis results to semantic properties of analyzed programs. Value and loop bound analysis, c.f. Fig. 2.1, are more or less standard abstract interpretations. The difference is that these analyses are performed on the binary level and not on the source level. Still, adequacy of these analyses is easily accepted. The instantiation of the abstract-interpretation framework for the microarchitectural analysis of a given execution platform, however, is far from trivial. In particular, it contains an abstraction of the execution platform. How does one make sure that such an abstraction is conservative? This will be explained in Sect. 2.2.3.

Let us give short descriptions of the different component analyses alongside the particular validation activities before we come to the validation of the central component.

2.2.1 Control-Flow Graph Reconstruction

The reconstruction of the control-flow graph (CFG) from a binary executable means to compute a safe approximation of the inter-procedural control flow of the executable [13]. This is achieved by the following two steps after having loaded the executable:

1. Classification of the loaded byte stream to identify individual assembly instructions and
2. Recursive reconstruction of the control flow based on this assembly-instruction classifications.

For Step 1, a specification of the instruction encoding is required. Instruction-set-architecture manuals provide this information, which is then used to implement instruction identification in the binary decoder of the aiT tool chain. To validate the implementation, we perform the so-called *decode* tests. For each supported instruction (in each supported addressing mode) we write a test case providing a reference as the expected result of the decoding. The decoding result is then compared to this reference.

In Step 2, the decoder uses the identified instruction stream to compose a safe control-flow approximation. To validate this, we compile a representative set of control structures (in a high-level language like C) and decode the resulting executable to compare the reconstructed control flow with a reference result.

2.2.2 Value Analysis

The value analysis determines safe approximations of the values in processor registers and memory cells for every program point. These approximations are used to determine bounds on the iteration number of loops and information about the addresses of memory accesses. The value analysis is based on the instruction semantics of the underlying target architecture. Like the instruction encoding, architecture manuals provide this information.

To validate the instruction-semantics implementation, we create a test case for each instruction and define pre- and post-conditions according to the expected effect of the particular instruction. These conditions are expressed by user annotations, which are read by the value analyzer. Pre-conditions are used to generate the machine state needed to execute the tested instruction. The post-conditions define the expected state after having executed the instruction under test.

2.2.3 Microarchitectural Analysis: Trace Validation

The microarchitectural analysis combines a cache and a pipeline analysis. It is an abstract interpretation of the program's execution on the underlying cache and pipeline architecture. The execution of a program is abstractly executed by feeding instruction sequences from the control-flow graph to the timing model, which then computes the changes of the abstract execution state at cycle granularity and keeps track of the elapsing clock cycles. The correctness proofs of the method have been conducted by Thesing [14] based on the theory of abstract interpretation.

The cache analysis described in [2, 5, 7] is incorporated into the pipeline analysis. At each memory access, where the concrete hardware would query and update the contents of the cache(s), the cache analysis applies the corresponding abstract cache effects to the abstract cache state.

The result of the microarchitectural analysis is either an upper execution-time bound for every basic block or a *prediction graph*. In the first case, these upper bounds are the coefficients in an integer linear program that represents the control flow of the program. This is the version usually described in publications about static WCET analysis, as it presents a clean work distribution. However, it has the disadvantage that too much information is lost at basic-block boundaries, namely the precise matching of final states at predecessor blocks to initial states at successor blocks. This loss of information entails a loss in precision. The prediction graph avoids this loss of precision. It consists of abstract states as nodes and edges for the transition between states and represents the evolution of the abstract execution states at processor-clock granularity and beyond basic-block boundaries. Note that in the description of trace validation a prediction event graph appears, which is the prediction graph extended by event annotations at its edges.

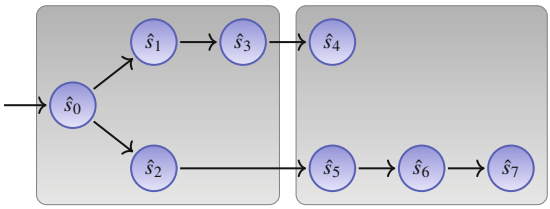

Fig. 2.2 Evolution of abstract hardware states $\hat{s}_i$. Each edge denotes a single cycle transition in the abstract state space. The gray boxes span the set of states that belong to the same basic block

As an example consider the prediction graph of Fig. 2.2, where the longest path is four transitions long, i.e., it takes four processor cycles to complete the program. Adding up the length of each longest path per basic block (denoted by the gray boxes) would neglect that there is no connection between the abstract states $\hat{s}_3$ and $\hat{s}_5$ and thus yield a worst-case estimate of five processor cycles.

Due to the complexity of the abstract architectural model, validation of the pipeline analysis cannot be done solely by testing the abstract implementation of individual instructions as we do it for CFG reconstruction and value analysis.

2.2.3.1 Semi-Automatic Derivation of the Abstract Architecture Model

Nowadays, hardware circuits are automatically synthesized from formal hardware specifications like VHDL or Verilog. Besides a formalization of the functional details, such specifications implicitly contain an execution model that also reflects the timing behavior of the whole system. It was a tempting idea to derive a pipeline analysis from the formal hardware model such that analysis and synthesized circuit share the same basis [11, 12].

However, the semi-automatic derivation of a timing model approach has not proven effective in the industrial context. Even if the hardware manufacturers grant access to their formal models (which is often not the case), the derivation process requires to fully understand the design, which might be a complex task for a complete processor including peripheral devices. Additionally, the quality of the resulting analysis depends on the coding style of the hardware model [11]. Results are excellent if the code features minimal dependencies between processes, a clear logical separation of different functionality into different processes/subprograms and a sequential logic design. Ideally, the code reflects the structural composition of the processor pipeline with explicit control signals to steer the flow of instructions and data. Models not adhering to those design principles complicate state abstractions and thus result in prohibitively resource-consuming analyses.

2.2.3.2 Trace Validation

For the reasons given above, the abstract architectural models are *hand-crafted* by human experts based on the available hardware reference documentation, which sometimes contains errors and usually lacks relevant details. Reverse engineering based on specific runtime measurements needs to fill this gap. Even if it were semi-automatically derived from a specification, the implementation of the microarchitectural analysis would still need to be validated. *Trace validation* checks for safe over-approximations of the predictions by matching observable hardware events recorded during concrete executions of instruction sequences against predictions of those events produced by the microarchitectural analysis. This is done for a sufficiently large set of instruction sequences that structurally covers the possible instruction flows (wrt. the different functional units, instructions, dependencies between instructions, etc.) of the processor pipeline.

Figure 2.3 shows the trace-validation workflow. An instruction sequence is executed on the actual hardware, or its execution is simulated using a VHDL model, to obtain an *observed event trace*. The microarchitectural analysis is modified to predict those events and annotate them to the edges of the generated prediction graph. In this fashion the microarchitectural analysis of an instruction sequence generates a *prediction event graph* that describes an over-approximation of all possible event traces that could occur while executing the instruction sequence. The observed trace of events, the reached execution state, and the consumed time are

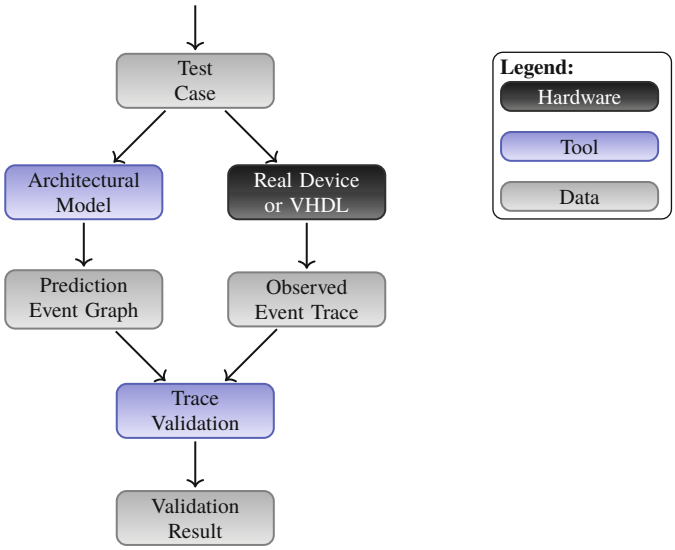

Fig. 2.3 Trace validation according to [9]. The instruction sequences together with the generated prediction graphs annotated by state and timing information are part of the Qualification Support Kit

checked for containment in the prediction graph. Trace validation is successful if the sequence of traced events is found in the prediction graph, and their predicted execution time does not underestimate the observed execution time.

The granularity at which the comparison takes place strongly depends on the debug facilities provided by the hardware. At best, timer interrupts are used to stop execution after each execution cycle. This way, the execution of instruction sequences is extended cycle by cycle to observe actual execution states and execution times.

The behavior of some components of the architectural state, such as the cache state, is unfortunately not directly observable. These need to be indirectly observed through executions that are forced to lead to cache hits and cache misses.

Thus a tremendous effort is required to cover both all instructions and all architectural components. This is essentially achieved by triggering many different architectural states through the execution of dedicated test cases.

The validation suite of the AbsInt static WCET tool aiT may contain several hundred individual test cases, even for a simple DLX-like architecture like the ARM Cortex-M4. For multi-core architectures, such as the TriCore TC275, which features three different cores, several thousand test cases are necessary to cover all architectural features.

How many test cases are required to cover the whole architectural behavior correlates to the complexity of the analyzed hardware, i.e., with the number of available instructions of the instruction set architecture, the number of components of the pipeline architecture like functional units, internal buffers, queues, memories, buses, and their states. Often unexpected (undocumented) hardware behavior is exposed while trying to understand existing test cases. This leads to additional test cases. Hence, the number of test cases that are sufficient in order to cover the (timing) relevant hardware behavior cannot be easily quantified in advance.

2.3 Conclusion

The AbsInt WCET analyzer aiT uses a combination of sound methods to derive safe upper bounds on execution times. Their implementation is quite complex, such that it is natural to query the soundness of the implementation of the technology. We describe the validation efforts employed to convince ourselves, the customers, and the certification authorities of the soundness of the implementation. The European Aviation Safety Agency (EASA), obliged to follow the strictest certification rules, those of DO178-C, has accepted AbsInt's aiT as a validated WCET analysis tool for several time-critical subsystems in the Airbus A380 and A350 planes.

References

1. Rtca/do-178c software considerations in airborne systems and equipment certification (2013)
2. M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, Cache behavior prediction by abstract interpretation. in *Proceedings of the Third International Symposium on Static Analysis, SAS'96*, ed. by R. Cousot, D.A. Schmidt. Aachen, September 24–26, 1996. Lecture Notes in Computer Science, vol. 1145 (Springer, Berlin, 1996), pp. 52–66. https://doi.org/10.1007/3-540-61739-6_33
3. P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, W. Yi, Building timing predictable embedded systems. ACM Trans. Embedded Comput. Syst. **13**(4), 82:1–82:37 (2014). https://doi.org/10.1145/2560033
4. P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, ed. by R.M. Graham, M.A. Harrison, R. Sethi, Los Angeles, January 1977 (ACM, New York, 1977), pp. 238–252. https://doi.org/10.1145/512950.512973
5. C. Ferdinand, Cache behaviour prediction for real-time systems. Ph.D. thesis, Saarland University, Saarbrücken (1997)
6. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, R. Wilhelm, Reliable and precise WCET determination for a real-life processor, in *International Workshop on Embedded Software*. Lecture Notes in Computer Science, vol. 2211 (2001), pp. 469–485
7. C. Ferdinand, R. Wilhelm, On predicting data cache behavior for real-time systems, in *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ed. by F. Mueller, A. Bestavros. Lecture Notes In Computer Science: Languages, Compilers, And Tools For Embedded Systems, vol. 1474 (Springer, Montréal, 1998), pp. 16–30. https://doi.org/10.1007/BFb0057777
8. C. Ferdinand, R. Wilhelm, Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst. **17**(2–3), 131–181 (1999)
9. G. Gebhard, Static timing analysis tool validation in the presence of timing anomalies. Ph.D. thesis, Saarland University (2013). http://scidok.sulb.uni-saarland.de/volltexte/2013/5558/
10. M. Lv, N. Guan, J. Reineke, R. Wilhelm, W. Yi, A survey on static cache analysis for real-time systems. Leibniz Trans. Embedded Syst. **3**(1), 5:1–5:48 (2016). https://doi.org/10.4230/LITES-v003-i001-a005
11. M. Pister, Timing model derivation—pipeline analyzer generation from hardware description languages. Ph.D. thesis, Saarland University (2012)
12. M. Schlickling, Timing model derivation—static analysis of hardware description languages. Ph.D. thesis, Saarland University (2013)
13. H. Theiling, Control flow graphs for real-time systems analysis. Ph.D. thesis, Universität des Saarlandes, Saarbrücken (2002)
14. S. Thesing, Safe and precise WCET determinations by abstract interpretation of pipeline models. Ph.D. thesis, Saarland University (2004)
15. L. Thiele, R. Wilhelm, Design for timing predictability. Real-Time Syst. **28**(2–3), 157–177 (2004). https://doi.org/10.1023/B:TIME.0000045316.66276.6e
16. R. Wilhelm, Mixed feelings about mixed criticality (invited paper), in *Proceedings of the 18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018*, ed. by F. Brandner, July 3, 2018, Barcelona. OASICS, vol. 63 (Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, 2018), pp. 1:1–1:9. https://doi.org/10.4230/OASIcs.WCET.2018.1

17. R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, S. Wilhelm, Static timing analysis for hard real-time systems, in *Proceedings of the 11th International Conference Verification, Model Checking, and Abstract Interpretation, VMCAI 2010*, ed. by G. Barthe, M.V. Hermenegildo, Madrid, January 17–19, 2010. Lecture Notes in Computer Science, vol. 5944 (Springer, Berlin, 2010), pp. 3–22. https://doi.org/10.1007/978-3-642-11319-2_3
18. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, C. Ferdinand, Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans. CAD Integr. Circuits Syst. **28**(7), 966–978 (2009). https://doi.org/10.1109/TCAD.2009.2013287