Safety Standards and WCET Analysis Tools

Daniel Kästner **Christian Ferdinand**

AbsInt GmbH, Science Park 1, D-66123 Saarbrücken, Germany

http://www.absint.com

Abstract

In automotive, railway, avionics, automation, and healthcare industries more and more functionality is implemented by embedded software. A failure of safety-critical software may cause high costs or even endanger human beings. Also for applications which are not highly safety-critical, a software failure may necessitate expensive updates.

Contemporary safety standards – including DO-178B, DO-178C, IEC-61508, ISO-26262, and EN-50128 – require to identify potential functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. For ensuring functional program properties automatic or model-based testing, and formal techniques like model checking become more and more widely used. For non-functional properties like timing identifying a safe end-of-test criterion is a hard problem since failures usually occur in corner cases and full test coverage cannot be achieved. For code-level timing analysis this problem is solved by abstract-interpretation-based static analysis techniques which provide full coverage and yield provably correct results.

In this article we focus on static analyses of worst-case execution time, which are increasingly adopted by industry in the validation and certification process for safety-critical software. First we will give an overview of the most important safety standards with a focus on the requirements for non-functional software properties. We then explain the methodology of abstract-interpretation-based analysis tools and identify criteria for their successful application. The integration of static analyzers in the development process requires interfaces to other development tools, like code generators or scheduling tools. Using them for certification requires an appropriate tool qualification. We will address each of these topics and report on industrial experience.

1 Introduction

The use of safety-critical embedded software in the automotive, avionics and healthcare industries is increasing rapidly. Failures of such safety-critical embedded systems may create high costs or even endanger human beings. Also for applications which are not highly safety-critical, a software failure may necessitate expensive updates. Therefore, utmost carefulness and state-of-the-art techniques for verifying software safety requirements have to be applied to make sure that an application is working properly. To do so lies in the responsibility of the system designers. Ensuring software safety is one of the goals of safety standards like DO-178B, DO-178C, IEC-61508, ISO-26262, or EN-50128. They all require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals.

Classical software validation methods like code review and testing with debugging cannot really guarantee the absence of errors. Formal verification methods provide an alternative, in particular for safety-critical applications. One such method is *abstract interpretation* [2], which allows to obtain statements that are valid for all program runs with all inputs. Such statements may be absence of violations of timing or space constraints, or absence of runtime errors. Static analysis tools in industrial use can detect stack overflows, violation of timing constraints [20], and can prove the absence of runtime errors [4].

The advantage of static analysis based techniques is that they enable full test coverage, but at the same time can reduce the test effort. Identifying end-of-test criteria for non-functional program properties like timing, stack size, and runtime errors is an unsolved problem. In consequence, the required test effort is high, the tests require access to the physical hardware and the results are not complete. In contrast, static analyses can be run by software developers from their workstation computer, they can be integrated in the development process, e.g., in model-based code generators, and allow developers to detect runtime errors as well as timing and space bugs in early product stages.

From a methodological point of view, static analyses can be seen as equivalent to testing with full coverage and, as such, are candidates for meeting all testing requirements listed in the above-mentioned standards. Thus, in the areas where validation by static analysis is technically feasible and applied in industry, e.g., for worst-case execution time analysis, stack size analysis or runtime error analysis, it defines the state-of-the-art testing technology.

In the following we will give an overview of the most important safety standards with a focus on the requirements for timing. Then we explain the basic methodology of static worst-case execution time (WCET) analysis and present the underlying concepts of our aiT tool. Industrial experience is summarized in Sec. 7.

2 Safety Standards

Safety standards like DO-178B [18], DO-178C, IEC-61508 [14], ISO-26262 [15] and EN-50128 [1] require to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. These standards mention explicitly three important non-functional safety-relevant software characteristics: Absence of runtime errors, execution time, and memory consumption. In the following, we will focus on execution time. Depending on the criticality level of the software the absence of safety hazards has to be demonstrated by formal methods or testing with sufficient coverage.

In the following, we give a short overview on the assessment of timing properties by the safety standards for avionics, space, automotive and railway systems, for general Electric/Electronic systems, and for medical software products.

2.1 DO-178B/DO-178C

Published in 1992, the DO-178B [18] ("Software Considerations in Airborne Systems and Equipment Certification"), is the primary document by which the certification authorities such as FAA, EASA, and Transport Canada will approve all commercial software-based aerospace systems. The purpose of D0-178B is "to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements." The criticality levels defined are Level A (most critical) to Level E (least critical).

The DO-178B emphasizes the importance of software verification. Verification is defined as a technical assessment of the results of both the software development processes and the software verification process. Sec. 6.0 of the DO-178B states that "verification is not simply testing. Testing, in general, cannot show the absence of errors." The standard consequently uses the term "verify" instead of "test" when the software verification process objectives being discussed are typically a combination of reviews, analyses and test. The purpose of the software verification process is to detect and report errors that may have been introduced during the software development processes. Removal of the errors is an activity of the software development processes. The general objectives of the software verification process are to verify that the requirements of the system level, the architecture level, the source code level and the executable object code level are satisfied, and that the means used to satisfy these objectives are technically correct and complete. At the code level the objective is to detect and report errors that may have been introduced during the software coding process. Non-functional safety properties are explicitly mentioned, including worst-case execution time.

The DO-178C, due to be finalized in 2011, will be a revision of DO-178B to bring it up to date with respect to current software development and verification technologies. It specifically focuses on model-based software development, object-oriented software, the use and qualification of software tools and the use of formal methods to complement or replace dynamic testing (theorem proving, model checking, and abstract interpretation).

2.2 IEC-61508 Edition 2.0

In 2010 a new revision of the functional safety standard IEC-61508 has been published, called Edition 2.0 [14]. It sets out a generic approach for all safety lifecycle activities for systems comprised of electrical and/or electronic and/or programmable electronic (E/E/PE) elements that are used to perform safety functions. The safety integrity levels are called SIL1 (least critical) to SIL4 (most critical). The timing properties are part of the software safety requirements specification and list response time and best case and worst-case execution time.

The IEC-61508 states that verification includes testing and analysis. In the software verification stage, static analysis techniques are recommended for SIL1 and highly recommended for SIL2-SIL4. Among these techniques, static worst-case execution time analysis is recommended in SIL1-4. Among the criteria to be considered for selecting specific techniques is the completeness and repeatability of testing, so where testing is used, completeness has to be demonstrated. The results of abstract-interpretation-based static analyses are considered a mathematical proof; their reliability is rated maximal (R3). The IEC-61508 also provides requirements for mixed-criticality systems: "Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. It shall be demonstrated either (1) that independence is achieved by both in the spatial and temporal domains, or (2) that any violation of independence is controlled. The justification for independence shall be documented". This has significant consequences for hardware selection and system configuration: it has to be ensured that there are no unpredictable timing-related interferences which might affect real-time functions. Cache-related preemption costs, pipeline effects, and timing anomalies have to be taken into account. For multicore processors it has to be shown that there are no inherent timing interferences between cores – which are quite common, e.g., due to collisions on shared memory buses between cores, or due to shared cache levels [3]. For achieving temporal independence the standard suggests deterministic scheduling methods. One sugges-

tion is using a cyclic scheduling algorithm which gives each element a defined time slice supported by worst-case execution time analysis of each element to demonstrate statically that the timing requirements for each element are met. Other suggestions are using time triggered architectures, or strict priority based scheduling implemented by a real-time executive [14].

2.3 ISO-26262

ISO-26262 (Road vehicles – Functional safety) [15] is the adaptation of the Functional Safety standard IEC 61508 for Automotive Electric/Electronic Systems. It has been published as an international standard in August 2011, replacing the IEC 61508 as formal legal norm for road vehicles.

Although the standard addresses *functional* safety the ISO-26262 also takes *non-functional* requirements into account. Concerning software validation it requires to identify functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals.

The ISO-26262 demands that the timing constraints of time-critical functions have to be covered by the specification of the software safety requirements. Here, both the *worst-case execution time* at the code level, and the *response time* at the system level has to be considered. Temporal constraints also are a part of the software architectural design (Sec. 7.4.5), especially the worst-case execution time. An important requirement is to establish "appropriate scheduling properties" which is highly recommended for all ASIL levels. Since without knowledge of upper bounds on the worst-case execution time no safe schedulability analysis is possible the availability of upper bounds of the WCET can be considered such a basic scheduling property. Furthermore, freedom of interference has to be ensured and, as in the IEC-61508, the software is subject to the hightes ASIL level used when temporal independence between the safety functions cannot be established (Sec. 7.4.10). Also during unit testing and integration testing upper bounds on the execution time have to be established.

While the ISO-26262 does not enforce specific testing and verification methods, the importance of static verification is emphasized; e.g. it is considered one of the three goals of the software unit design and implementation stage. Static analysis is explicitly listed among the methods for software unit design and implementations and variations of static analysis are (highly) recommended for all ASIL levels (Sec. 8, Table 9).

2.4 CENELEC prEN 50128

The CENELEC EN-50128 [1] also has been subject to a revision which was published in 2011. It provides a set of requirements with which the development, deployment and maintenance of any safety-related software intended for railway control and protection applications shall comply. It addresses five software safety integrity levels and identifies and lists appropriate techniques and measures for each level of software safety integrity.

Static analysis based on abstract interpretation, e.g., worst-case execution time analysis, belongs to the recommended testing and verification techniques. It is highly recommended for SIL 3/4, recommended for SIL 1/2 and should be applied throughout the development process: in the software validation stage, the software integration test, software/hardware integration test, and software component test.

2.5 Regulations for Medical Software

Standards relevant for medical software are the EN-60601 and IEC-62304. The EN-60601 formulates requirements for the software lifecycle and risk management [5]. The standard IEC-62304 describes a lifecycle for software development with a focus on maintenance and on component-oriented software architectures [6].

Beyond these standards country-specific requirements have to be respected. In the following we will shortly discuss the American and German regulations. The presentation of the US regulations follows [9]. Software validation is a requirement of the Quality System regulation (cf. Title 21 Code of Federal Regulations (CFR) Part 820, and 61 Federal Register (FR) 52602, respectively). Validation requirements apply to software used as components in medical devices, to software that is itself a medical device, and to production software. Verification means "confirmation by examination and provision of objective evidence that specified requirements have been fulfilled" [8]. In a software development environment, software verification is confirmation that the output of a particular phase of development meets all of the input requirements for that phase. While software testing is a necessary activity, in most cases software testing by itself is not considered sufficient to establish confidence that the software is fit for its intended use. Additional verification activities are required, including static analysis.

In Europe, the validation of medical software has to follow the EU-directive 2007/47/EC [19], which, in Germany has been incorporated into national law in 2010 [17]. It states that software in its own right, when specifically intended to be used for medical purpose, has to be considered a medical device. For devices which incorporate software or which are medical software in themselves, the software *must* be validated *according to the state of the art*.

3 Abstract Interpretation

Static data flow analyses compute invariants for all program points by fixed point iteration over the pro-

gram structure or the control-flow graph. The theory of abstract interpretation [2] offers a semantics-based methodology for static program analyses. The concrete semantics is mapped to an abstract semantics by abstraction functions. While most interesting program properties are undecidable in the concrete semantics, the abstract semantics can be chosen for them to be computable. The static analysis is computed with respect to that abstract semantics. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster. By skillful definition of the abstract domains a suitable trade-off between precision and efficiency can be attained.

For program validation there are two essential properties of static analyzers: soundness and safety. A static analysis is called *sound* if the computed results hold for any possible program execution. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an overapproximation of the concrete semantics. Imprecisions can occur, but they will always be on the *safe* side.

In WCET analysis, soundness means that the computed WCET bound holds for any possible program execution. Safety means that the only imprecision occurring is overestimation: the WCET must never be underestimated.

4 WCET Analysis: Worst-Case Execution Time Prediction

Many tasks in safety-critical embedded systems have hard real-time characteristics. Failure to meet deadlines may be as harmful as producing wrong output or failure to work at all. Yet the determination of the Worst-Case Execution Time (WCET) of a task is a difficult problem because of the characteristics of modern software and hardware [22].

Embedded control software (e.g., in the automotive industries) tends to be large and complex. The software in a single electronic control unit typically has to provide different kinds of functionality. It is usually developed by several people, several groups or even several different providers. Code generator tools are widely used. They usually hide implementation details to the developers and make an understanding of the timing behavior of the code more difficult. The code is typically combined with third party software such as real-time operating systems and/or communication libraries.

Concerning hardware, there is typically a large gap between the cycle times of modern microprocessors and the access times of main memory. Caches and branch target buffers are used to overcome this gap in virtually all performance-oriented processors (including high-performance micro-controllers and DSPs). Pipelines enable acceleration by overlapping the executions of different instructions. Consequently the execution behavior of the instructions cannot be analyzed separately since it depends on the execution history. Cache memories usually work very well, but under some circumstances minimal changes in the program code or program input may lead to dramatic changes in cache behavior. For (hard) real-time systems, this is undesirable and possibly even hazardous. Making the safe yet – for the most part – unrealistic assumption that all memory references lead to cache misses results in the execution time being overestimated by several hundred percent.

The widely used classical methods of predicting execution times are not generally applicable. Software monitoring and dual-loop benchmarks modify the code, which in turn changes the cache behavior. Hardware simulation, emulation, or direct measurement with logic analyzers can only determine the execution time for some fixed inputs. They cannot be used to infer the execution times for all possible inputs in general.

In contrast, abstract interpretation can be used to efficiently compute a safe approximation for all possible cache and pipeline states that can occur at a program point in any program run with any input. These results can be combined with ILP (Integer Linear Programming) techniques to safely predict the worst-case execution time and a corresponding worst-case execution path. A survey of methods for WCET analysis and of WCET tools is given in [23].

AbsInt's timing verifier aiT [11] computes a safe upper bound for the WCET of a task, assuming no interference from the outside. Effects of interrupts, IO and timer (co-)processors are not reflected in the predicted runtime and have to be considered separately within system-level timing analysis. The main input of aiT is the binary executable. The analysis does not require any code modification and does not rely on debug information. The results are independent from flaws in the debug output and refer to exactly the same code as in the shipped system. aiT determines the WCET of a program task in several phases [12], which makes it possible to use different methods tailored to each subtasks [21]. First, the control-flow graph (CFG) is reconstructed from the input file, the binary executable. Then value analysis computes value ranges for registers and address ranges for instructions accessing memory; a loop bound analysis determines upper bounds for the number of iterations of simple loops. Subsequently, a cache analysis classifies memory references as cache misses or hits [10] and a pipeline analysis predicts the behavior of the program on the processor pipeline [16]. Finally the path analysis determines a worst-case execution path of the program [21].

The results of aiT are reported as annotations in call graphs and control-flow graphs (cf. Fig. 1,2), and as report files in text format and XML format. The overall WCET bounds/estimations for sequential code pieces

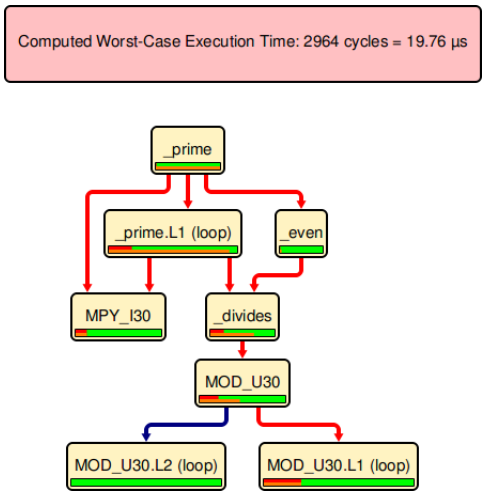

Figure 1: Call graph with WCET results

can also be communicated to the system-level analyzer SymTA/S [13], which computes worst-case response times from the sequential WCETs, taking into account interrupts and task preemptions.

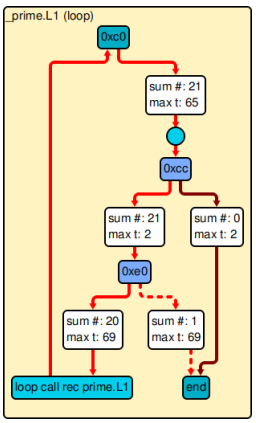

Figure 2: Basic-block graph with WCET results

aiT is available for various microcontrollers with the following cores: ARM7, Infineon C16x/ST10, Texas Instruments C33, Motorola HC11, HCS12/STAR12, Intel i386, i486, LEON2, LEON3, Motorola M68020, Freescale MPC 5xx, MPC603e, MPC55xx, MPC755, Infineon TriCore, NEC V850.

In general, the availability of safe worst-case execution time bounds depends on the predictability of the execution platform. Especially multi-core architectures may exhibit poor predictability because of essentially nondeterministic interferences on shared resources which can cause high variations in execution time. [3] gives a more detailed overview and suggests example configurations for available multi-cores to support static timing analysis.

5 Integration in the Development Process

Static analysis tools are not only applicable at the validation stage but also at the development stage. One advantage of static analysis methods is that no testing on physical hardware is required. Thus the analyses can be called just like a compiler from a workstation computer after the linking stage of the project. aiT can be used in batch mode facilitating the integration in a general automated build process. This enables developers to instantly assess the effects of program changes on WCET bounds. Potential problem/defects are detected early, so that late-stage integration problems can be avoided.

In general, static analysis tools can be smoothly coupled with other development tools. There are couplings of aiT with model-based code generators like Esterel SCADE [7], dspace TargetLink (currently in development), and ETAS ASCET. The timing analysis can be invoked from the modeling level, and the analysis results can be reported back to the modeling level. As an example, with the SCADE integration, the worst-case execution time can be displayed for each model component and each SCADE operator. Also a tool coupling to the system-level scheduling analyzer SymTA/S (e.g. SymTA/S [13]) is available. With this coupling, both the code-level and the system-level timing aspects are seamlessly covered.

6 Tool Qualification

Many safety standards require a dedicated tool qualification to demonstrate the a given software tool works correctly in the operational context of the user. aiT has successfully been qualified as analysis tool according to DO-178B. The qualification process can be automated to a large degree by *Qualification Support Kits*.

Qualification kits are available for various aiT versions/targets. The kits consist of a report package and a test package. The report package lists all functional requirements and contains a verification test plan describing one or more test cases to check each functional requirement. The test package contains an extensible set of test cases and a scripting system to automatically execute all test cases and to evaluate the results. The generated reports can be submitted to the certification authority as part of the certification package.

7 Industrial Experience

In recent years tools based on static analysis have proved their usability in industrial practice and, in consequence, have increasingly been used by avionics, automotive and healthcare industries. In the following we report on experiences gained with the aiT WCET Analyzer.

Since the exact WCET is usually unknown for typical real-life applications, statements about the precision of aiT are hard to obtain. For an automotive application running on the MPC 555, the results of aiT are 5 – 10 % above the highest execution times observed in a series of measurements (which may have missed the real WCET). For an avionics application running on the MPC 755, Airbus has noted that aiT's WCET of a task typically is about 25 % higher than some measured execution times for the same task, the real but unknown WCET being in between [20]. Measurements at AbsInt have indicated overestimations ranging from 0 % (cycle-exact prediction) till 10 % for a set of small programs running on M32C, TMS320C33, and C166/ST10.

8 Conclusion

The quality assurance process for safety-critical embedded software is of crucial importance. The cost for system validation grows with increasing criticality level to constitute a large fraction of the overall development cost. The problem is twofold: system safety must be ensured, yet this must be accomplishable with reasonable effort.

Tools based on abstract interpretation can perform static program analysis of embedded applications. Their results are determined without the need to change the code and hold for all program runs with arbitrary inputs. Especially for non-functional program properties they are highly attractive, since they provide full coverage and can be seamlessly integrated in the development process.

We have presented the static WCET analyzer aiT. It allows to inspect the timing behavior of (time-critical parts of) program tasks. It takes into account the combination of all the different hardware characteristics while still obtaining tight upper bounds on the WCET of a given program in reasonable time. aiT has been used by Airbus in the development of various safety-critical applications for the A380 (and other airplanes).

aiT can be used as analysis tools for the certification according to development standards like DO-178B or ISO 26262. The tool is used by many industry customers from avionics and automotive industries and its applicability has been proved in industrial practise. The tool qualification process can be automated to a large extent by dedicated Qualification Support Kits.

Acknowledgement

The work presented in this paper has been supported by the European FP7 projects INTERESTED, ALL-TIMES, and PREDATOR.

References

[1] CENELEC DRAFT prEN 50128. Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, 2009.

[2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[3] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza (Burguière), J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

[4] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, pages 437–451, 2007.

[5] DIN EN 60601-1-4. Medizinische elektrische Geräte – Teil 1–4: Allgemeine Festlegungen für die Sicherheit – Ergänzungsnorm: Programmierbare elektrische medizinische Systeme, 2001.

[6] DIN EN 62304. Medical device software – Software life cycle processes, 2006.

[7] Esterel Technologies. SCADE Suite. http://www.esterel-technologies.com/products/scade-suite.

[8] FDA. US Food and Drug Administration. CFR – Code of Federal Regulations Title 21. Part 820. Quality System Regulation. http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfCFR/CFRSearch.cfm, 2010.

[9] General Principles of Software Validation; Final Guidance for Industry and FDA Staff. U.S. Department of Health and Human Services, Food and Drug Administration; Center for Devices and Radiological Health, Office of Device Evaluation, Office of In Vitro Diagnostics; Center for Biologics Evaluation and Research, Office of Blood Research and Review, 2005.

[10] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[11] C. Ferdinand and R. Heckmann. Worst-case execution time – a tool provider's perspective. In *11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing ISORC 2008, Orlando, Florida, USA*, May 2008.

[12] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, First Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[13] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. *IEEE Proceedings on Computers and Digital Techniques*, 152(2), Mar. 2005.

[14] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[15] ISO 26262-WD. Road vehicles – Functional safety, 2009.

[16] M. Langenbach, S. Thesing, and R. Heckmann. Pipeline modeling for timing analysis. In *Proceedings of the 9th International Static Analysis Symposium SAS 2002*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2002.

[17] Medizinproduktegesetz in der Fassung der Bekanntmachung vom 7. August 2002 (BGBl. I S. 3146); zuletzt geändert durch Artikel 12 des Gesetzes vom 24. Juli 2010 (BGBl. I S. 983).

[18] Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.

[19] Richtlinie 2007/47/EG des Europäischen Parlaments und des Rates vom 5. September 2007 zur Änderung der Richtlinien 90/385/EWG des Rates zur Angleichung der Rechtsvorschriften der Mitgliedstaaten über aktive implantierbare medizinische Geräte und 93/42/EWG des Rates über Medizinprodukte sowie der Richtlinie 98/8/EG über das Inverkehrbringen von Biozid-Produkten.

[20] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[21] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, Dec. 1998.

[22] R. Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1 – 14–23. CRC Press, 2005.

[23] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.