Impact of Dependable Software Development Guidelines on Timing Analysis

Ingomar Wenzel*, Raimund Kirner*, Martin Schlager*, Bernhard Rieder*, Bernhard Huber*

***Abstract*— The knowledge of the worst-case execution time (WCET) of real-time tasks is mandatory to ensure correct timing behavior of real-time systems. However, in practice an exact WCET analysis is often intractable due to limitations in computability and analysis complexity of real-size programs.**

In this paper we analyze how development guidelines for dependable software support and simplify WCET analysis. We investigate three guidelines and their impact on WCET analyzability. *DO-178B* as a production guide for avionics software expresses requirements that are relevant for timing analysis. The *MISRA Guidelines* include C programming guidelines that improve the WCET analyzability of software. Finally, *ARINC 653,* a standard for software architectures of avionic systems, provides examples on how to simplify timing analysis already at the design level as early as in system design. The argument of this paper is that careful system design and programming improves the timing analyzability of real-time systems.

I. Introduction

Many innovative developments in the transportation sector (aerospace, automotive) require complex electronics and software systems. In the next decade a further increase in electronic and software functions can be expected. For instance, in the automotive domain, more and more comfort (seat positioning, vehicle handling), safety (active pre-crash, collision warning) and infotainment (Internet, digital TV) functions will be introduced.

However, according to figures of the German automobile association ADAC, currently more than 50% of all breakdowns of cars are directly related to defects of car electronics. In 2015, even more than 60% electronic related breakdowns are expected [5]. Thus, a sufficient degree of dependability is of utmost importance – in particular for safety critical functions.

In case the correct operation of a particular service does not only depend on the correct result in the value domain, but also on the timeliness of the computed result, such service is called a real-time service [8].

Due to the temporal constraints required for correct operation of a real-time system, predictability in the temporal domain is a stringent imperative to be satisfied. Therefore, it is necessary to determine the timing behaviour of the tasks running on a real-time computer system. Worst-case execution time (WCET) analysis is the research field investigating methods to assess the worst-case timing behaviour of real-time tasks.

A. Objective

The key purpose of this paper is to investigate the relationship of guidelines for safety critical software development of real-time systems with respect to the applicability of WCET analysis methods. The goal within this work is to gain deeper insights into the capability of performing WCET analysis for state-of-the-art real-time programs.

Imposed by dependability requirements for safety-critical hard real-time systems, a number of broadly accepted ”guidelines” exist. These guidelines are intended to allow or ease the process of validation and verification by introducing a number of restrictions on the software engineering processes. These ”restrictions” address various levels and aspects in the software life cycle. Some effects of these restrictions are supposed to influence the structure of the actually produced program code.

In this work, the impact of these restrictions and constraints on the real-time program code is analyzed by focusing on the relevant aspects of the code structure for WCET analyzability.

B. Classification of the Documents Investigated

In this paper, the following three types of documents are considered. For each type, one section in this paper describes the covered aspects within that type of document and their impact on WCET analysis:

- **Guidelines for Software Production** cover the whole software life cycle with the goal to ensure a desired level of confidence into dependability (*DO-178B* [7]).
- **Coding Guidelines** (also referred to as *Code Standards*[1]) impose constraints on the pragmatism of a language. Examples for coding guidelines are the *MISRA Guidelines* for the C language [9] or Spark ADA [4].
- **Architectural Standardizations** specify the application programming interface for real-time software. As an example, in this paper a brief overview about the contents of *ARINC 653* [2] is presented.

Figure 1 depicts the relationship between these different document classes.

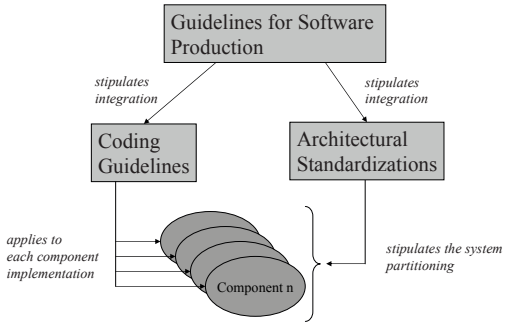

Fig. 1. Classification of Documents

*All authors are affiliated with Institute of Computer Engineering, Vienna University of Technology, Treitlstraße 3, 1040 Vienna, Austria. For further information, please contact ingo@vmars.tuwien.ac.at

[1]In *DO-178B* (Section 11.8, p52) code standards are introduced as one aspect of software life cycle management [7].

II. Taxonomy of Software Complexity for Timing Analysis

In this section we give a generic discussion about the hurdles of predicting the runtime behavior of code and describe how programming guidelines can improve the predictability of code. These concepts are the motivation why development guidelines for dependable software are useful to make WCET analysis feasible.

A. Code Patterns Challenging WCET Analyzability

In the following we describe code patterns that rise some challengenges on timing analysis [6].

1) Loops: Program loops are defined as jumps to previously executed program statements by the transition of so-called *back-edges* [1]. Loops are a challenge for program analysis since they can potentially cause an exponential increase of the number of execution paths through a code fragment. The analyzability of loops becomes worse if no upper bound for the iteration count can be calculated. In this case, manual code annotations are necessary to guide the analysis.

2) Irreducible Flow Graphs: Irreducible (or unstructured) flow graphs [1] cause problems for loop analysis. Irreducible flow graphs arise from using `goto` statements for entering a loop [6]. Using only structured programming implies that the control flow graph is free of irreducible control flow.

3) Function Calls: One hurdle of function calls on timing analysis is that the execution time within functions may depend on the calling context. Depending on the type of function call, additional difficulties may arise for external calls, function pointer calls (functions called at runtime may not be known statically) and library calls (in case no source code is available).

4) Dynamic Data Structures: The control flow of a program may depend on the content of data structures. This becomes challenging in case that data structures are constructed dynamically. Manual code annotations can be used to guide the WCET analysis tool in case it cannot determine the data structure automatically. However, for complex data structures it will become a cumbersome and error-prone task to manually annotate the control flow resulting from the computationally intractable space of potential instances of dynamic data structures.

B. Toward WCET-Analyzable Code

In this subsection we describe a software engineering technique to obtain more predictable code, which is called *wcet-oriented programming* [11], [14].

WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control flow decisions or, if this cannot be completely achieved, it restricts operations that are only executed for a subset of the input-data space to a minimum.

In case WCET-oriented programming is not able to avoid all input-dependent control flow the *single path transformation* can be used eliminating any input-dependent control flow [13], [12]. The main idea behind this approach is to transform control flow manipulating statements into predicated execution instructions (i.e., some instructions are conditionally executed but the timing of the instruction remains constant).

The major benefit of input-independent control flow is that the execution time jitter of the code is minimized and the WCET analysis becomes much more simple. However, the analysis of effects from data caches and task preemptions still remains challenging.

III. MISRA Guidelines

A. Overview on the MISRA C Rules

The Motor Industry Software Reliability Association (MISRA) issued “Guidelines for the Use of the C Language in Vehicle Based Software” in 1998 [9]. These guidelines describe a subset of the C language that is intended to be suitable for embedded automotive systems. The document contains a list of rules and recommendations for the safe usage of classes of C language statements within the domain of automotive software development. Recently, this document has been updated by “MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems” [10]. The new guidelines are a refined (mainly regarding the presentation style) version that is compatible to [9].

1) Objectives of MISRA C: The goal of the *MISRA Guidelines* consortium is to establish a safe subset of the C language for industrial usage. The static checking of conformance with these rules is emphasized. It is not the intention to promote the use of the C language in the automotive industry. Rather it is the goal to make existing development practice safer.

2) Adopting the MISRA C Subset: In order to claim compliance of **a product** with the *MISRA Guidelines* C subset, the following steps are required [9]:

1) A close *compliance matrix* to show how compliance has been enforced for each rule; i.e., it is ensured that each potential violation of rules has its assigned detection mechanism.
2) In some instances it might be necessary to deviate from rules. For these cases a *deviation procedure* is required that ensures a formal procedure to be used in order to authorize these deviations. The more serious the deviations are, the more technical competence should be required to justify the introduced risk.
3) The whole process (use of the subset, static checking tools, deviation procedure, documentation) should be stipulated within the quality management system.

B. The MISRA C Rules

The rules are grouped into several categories. A complete list including a brief description of all rules is provided in Appendix A of [9].

Out of 127 rules, 10 regulations have been identified to have an impact on WCET analysis.

1) Types: Rule 14 (required) ”The type `char` shall always be declared as unsigned char or signed char.” is useful due to the implementation specific compiler behavior when using the `char` data type.

Rule 15 (advisory) ”Floating point implementations should comply with a defined floating point standard.” and Rule 16 (required) ”The underlying bit representations of

floating point numbers shall not be used in any way by the programmer." may be useful when applying semantic analysis (e.g., abstract interpretation) in order to ease the correct simulation of the instruction's semantics.

2) ControlFlow: This class of rules covers aspects that influence the structuring of the code.

Rule 52 (required) "There shall be no unreachable code."

Rule 56 (required) "The `goto` statement shall not be used.", Rule 57 (required) "The `continue` statement shall not be used." and Rule 58 (required) "The `break` statement shall not be used (except to terminate the cases of a switch statement).' ensure that the code does not contain unstructured loops (Subsection II-A.2).

3) Functions: Rule 69 (required) "Functions with variable numbers of arguments shall not be used." eases in the construction of the timing graph because the execution time required for the function call itself remains constant.

Rule 70 (required) "Functions shall not call themselves, either directly or indirectly." forbids recursion with the argument that recursion may cause stack overflow errors.

4) Pointers and Arrays: Rule 104 (required) "Non-constant pointers to functions shall not be used." supports to derive/constrain the set of functions that may include the function actually called by a function pointer reference.

5) Standard Libraries: Rule 118 (required) "Dynamic heap memory allocation shall not be used." argues that the dynamic memory instructions (like `calloc`, `malloc`, `realloc` and `free`) cause a number of undefined and implementation-defined behavior. Note, that there may exist library functions that implicitly call memory allocation functions. Thus, these functions shall not be used, too.

C. Summary on MISRA Guidelines

The impact of the rules on WCET analysis is on the code structure level (with respect to the features relevant for WCET analyzability identified in Subsection II-A).

The *MISRA Guidelines* pose a number of restrictions on the original C language subset. Only few rules could be identified to directly influence the WCET analyzability. The implications of these rules are, however, of significant importance for WCET analysis because they enforce program code that is easier to analyze. Especially (i) the rule prohibiting the use of dynamic memory allocation excludes the use of all algorithms using dynamic data structures and (ii) the limited usability of control flow changing instructions prohibits loop constructs that would be hard to analyze. However applying the *MISRA Guidelines* does still not guarantee WCET analyzability.

IV. DO-178B: Software Considerations in Airborne Systems and Equipment Certification

A. Overview on DO-178B

The purpose of DO-178B is to provide guidelines for the production of software for airborne systems that perform their intended function with a level of confidence that complies with airworthiness requirements [7]. This goal is intended to be achieved by:

- Objectives for the software life cycle processes,
- Description of *activities* and *design considerations* to achieve these goals,
- Descriptions of the evidence that indicates that these objectives have been accomplished.

B. Organization of DO-178B

The guideline *DO-178B* is organized as follows. First, system aspects relating to software development are described (e.g., link between system and software processes, failure conditions and software level classification, system architecture). Important for validation is the introduction of software criticality levels (Figure 2).

Software Level	Failure condition categories	Description
Level A	Catastrophic	Prevent continued safe flight and landing
Level B	Hazardous/Severe-Major	Large reduction in safety margins, physical distress such that the flight crew could not be relied on to perform their tasks correctly
Level C	Major	Signifcant increase in workload of crew, discomfort of occupants including injuries
Level D	Minor	No significant reduction in spacecraft safety
Level E	No Effect	Do not affect the operational ability of the aircraft

Fig. 2. Software Criticality Levels

Next, the software life cycle and the software planning process is outlined. Then, the typical software development process (requirements process, design process, coding process, integration process) and the verification process are described. Verification consists of software reviews/analyses and software testing processes. Next, the software configuration management process, the software quality assurance process, and certification are considered. Finally, a collection of information on the software life cycle data (e.g., design standards, code standards, etc.) is presented.

C. Implications on WCET Analysis by DO-178B

In *DO-178B* safety related requirements are part of the system requirements which are inputs to the software life cycle processes. These requirements usually include performance requirements. However, no explicit reference to WCET analysis is provided. The system requirements are subject to the processes described in *DO-178B*. The next reference regarding correct timing behavior can be found in Chapter 6 of *DO-178B* [7]. Since testing generally cannot show the absence of errors, verification includes a combination of reviews, analyses and tests. On the one side in this section, the objectives for verification are described. On the other side, corresponding activities ensuring these objectives are proposed.

D. Summary of DO-178B

DO-178B does not exhaustively stipulate issues regarding timing analysis. However, especially in the verification section timing analysis is explicitly mentioned.

V. Avionics Application Software Standard Interface (ARINC 653)

An important paper considering the role of timing analysis in the certification of integrated modular avionics is [3].

The "Supplement 1 to ARINC Specification 653" [2] expands and clarifies the content of ARINC 653. In particular, the document focuses on avionics operating system partition management, intrapartition communication, and health monitoring.

A. System Overview of ARINC 653

The purpose of *integrated modular avionics* (IMA) is to support the independent execution of avionics applications. The best way to achieve this is to partition the system, i.e. a functional separation usually for fault-containment and in order to simplify verification, validation, and certification [2].

The basic entity is a partition that contains one application consisting of one or more tasks and comprising its own context and configuration. Partitions are executed strictly cyclically, i.e. the processor time allocated to a partition is statically determined. A minimum interface for application partitions to access other system components is provided by the application/executive (APEX) interface.

The software that resides on the hardware platform consists of:

1) **Application partitions** contain the application code specific for a particular application.
2) The **OS kernel** provides the API to the Operating System (OS).
3) **System partitions** provide interfaces to the outside world.
4) **System specific functions** comprise hardware drivers, test-functions, etc.

The expected benefits of this architecture are portability, reusability, modularity, and integration of multiple-criticality software.

B. Partition Timing

The system integrator knows the timing requirements of the partitions and is responsible that these requirements are satisfied. It is the responsibility of the application developer to configure the processes within an application. A detailed list of responsibilities is listed in [2].

C. Impact on WCET Analysis

ARINC 653 clearly outlines the necessity for time partitioning in an integrated architecture. This partitioning is prescribed at the architecture level, the concrete implementation and timing assessment on the target platform is left open to the application developer (and the respective architectural part to the system integrator).

The code structure is influenced by providing standardized operating system calls (that can be expected to fulfill well specified timing requirements).

VI. Conclusion

In order to establish a meaningful term ”WCET analyzability” we outlined three important aspects of WCET analyzability. In Subsection II-A we introduced requirements for analyzability, a more normative approach has been presented in Subsection II-B.

We investigated the impact of various guidelines for safety critical software engineering of real-time systems on WCET analyzability. Using three currently used and practical important guidelines we put these three guidelines in a common context and analyzed the topics stipulated by each of them. *DO-178B* considers the whole process of safety critical software development. The *MISRA Guidelines* limit the language subset to be used in safety critical automotive applications. *ARINC 653* specifies an architecture for IMA. Common to all three documents is - more or less - some impact on the WCET analyzability.

Regarding the *MISRA Guidelines* some of the rules ease WCET analysis by restricting the language subset. However, these rules are not sufficient to guarantee WCET analyzability (Section III).

DO-178B does not exhaustively stipulate how timing requirements have to be taken into account when verifying hard real-time systems. Rather timing constraints are considered as application requirements and therefore have to undergo the same processes as functional requirements. Although the one or other hint is given, a lot of effort has to be invested in the processes assessing correct timing behavior of the applications.

ARINC 653 describes the architectural framework for IMA. Time partitioning is an essential part of the architecture. By enforcing the use of the APEX interface for applications, the degree of freedoms within applications is restricted, which in turn eases the timing analysis process.

Concluding, when considering all restrictions on the code structures imposed by the sum of these guidelines, safety critical real-time programs can be analyzed by state-of-the-art WCET analysis methods. The shift towards integrated architectures provides an interesting option and a clear interface for WCET analysis methods.

For future guidelines we suggest to include more detailed methodologies and processes related to timing analysis, especially to enforce the consideration of timing issues from the beginning of system design.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] APEX Working Group. *Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface*, 2003.

[3] N. Audsley, I. Bate, and A. Grigg. The role of timing analysis in the certification of IMA systems. In *Certification of Ground/Air Systems Seminar (Ref. No. 1998/255)*, pages 6/1–6/6, 1998.

[4] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[5] V. der Automobilindustrie (VDA). HAWK2015 – Herausforderung Automobile Wertschöpfungskette. Henrich Druck + Medien GmbH, Schwanheimer Strasse 110, D-60528 Frankfurt am Main, 2003.

[6] J. Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Real-Time Technology and Applications Symposium*, pages 46–55, 1999.

[7] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[8] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 3rd edition, 1997.

[9] MISRA The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 1998.

[10] MISRA The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*, Oct 2004.

[11] P. Puschner. Algorithms for dependable hard real-time systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[12] P. Puschner. The single-path approach towards WCET-analysable software. In *Proc. IEEE International Conference on Industrial Technology*, pages 699–704, Dec. 2003.

[13] P. Puschner and A. Burns. Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[14] P. Puschner and R. Kirner. Avoiding timing problems in real-time software. In *Proc. IEEE Workshop on Software Technologies for Future Embedded Systems*, pages 75–78, May 2003.