# Unified Approach to Static and Runtime Verification

Olga Thoss[1] Andreas Werner[1] Robert Kaiser[1] Reinhold Kroeger[1]

**Abstract:** Smart living environments are increasingly based on embedded information and communication technology. Generally, users are no technical experts and rely on the correct functioning of the system. Formal verification of a system's functional and non-functional properties is often regarded as the ultimate way to achieve the highest levels of trust as demanded for today's dependable systems. However, static verification, though sound in theory, is often impractical given the ever-increasing complexity of software and the non-deterministic nature of some mechanisms of the underlying hardware architecture. We argue that by supplementing static verification with runtime verification, a high level of trust can be achieved. In this paper, we report on an ongoing effort for tool-supported verification of functional and non-functional properties by combining static and runtime verification techniques.

**Keywords:** static verification; runtime verification; OS microkernel; SPARK; WCET; AQUAS

## 1 Introduction

Today's systems become more and more complex, and even domain experts are sometimes in doubt regarding their correct behaviour in rare and non-standard situations. Especially embedded systems incorporate increasing functionality and have to deal with a wide spectrum of sensors and actors interacting with the environment. Real-time properties requiring a guaranteed reaction of the system within a given limited time window are often associated as well, and safety of the users has to be ensured by law. Furthermore, these critical systems often have to face uncertainty which may originate from unknown device configurations at design time or unforeseen changes of the environment during operation. Uncertainty may also exist in control algorithms. For example, to guarantee a safe behaviour of trained AI algorithms in previously unseen situations is inherently difficult, if not impossible.

Under these conditions it is a complex and highly responsible task for developers to deliver a high level of trust in the developed software. This is commonly achieved through certification. To certify software for a given Safety Integrity Level (SIL), or ASIL level in the automotive systems context, it has to be thoroughly tested, specific models and analysis methods have to be used up to a formal, mathematical verification of required system properties. Today, all this has to happen before the system is actually used.

Due to the described complexity of current and future systems we do not believe that a full static verification and validation at design time is possible any longer to deliver the necessary trust. Instead, we have started to work on a methodology which distinguishes between

[1] RheinMain University of Applied Sciences, firstname.lastname@hs-rm.de

doi:10.18420/inf2020_93

verification activities carried out at design time and those at runtime. In summary, at design time static verification takes place, i.e. specified functional as well as non-functional or timing system properties are formally proven to the highest possible degree for a reasonable maximal effort. For properties which cannot be proven statically, sufficiently strong monitors are generated which are executed at runtime to monitor correct system behaviour. In the undesired case of detecting a property violation at runtime, the underlying system architecture is prepared to reconfigure the application to ensure acceptable behaviour. This adaptivity has to be supported by the application design. In total, a trusted self-adapting application seems to be reachable.

In the following Section 2, the functional properties, their verification and the approach for adaptation are considered. In Section 3, timing is taken into account as this is the most important non-functional property regarding real-time behaviour. In both sections, the current status of work is described. The paper closes with a summary and outlook. The considered use case and examples are taken from the AQUAS EU project [20b] to which we contribute.

## 2 Functional Verification

### 2.1 Static Verification

Typically, in the design phase a system model is constructed which specifies the overall system and its functional and non-functional properties and constraints. To simplify the designer's work, rather than using the industry standards SysML and OCL directly, we propose to use a DSL (Domain Specific Language) with an expressiveness to target the class of applications in mind. Such standard models shall then be generated from the DSL.

This generated system model will then be semi-automatically transformed into a set of views by exporting and transforming the model. Each view provides information concerning a specific aspect of the system. For the functional view and its verification, the SysML model is transformed into a SPARK interface definition, and functional OCL constraints into SPARK contracts with pre- and post-conditions (see Fig. 1). In addition to specified application constraints, other conformance rules, e.g. standards, company rules etc., can be taken into account as well, resulting in additional contracts or contract restrictions.

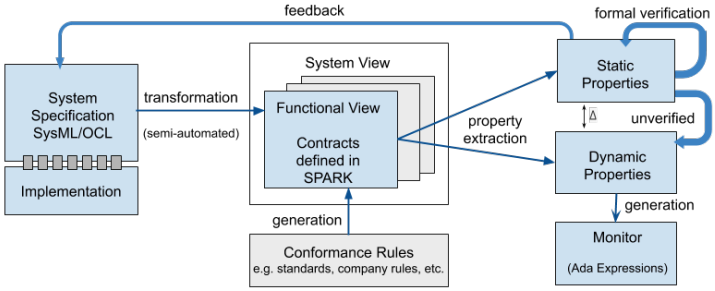


Fig. 1: Functional Verification During the Design Phase

During the design phase, the verification of selected functional properties can be carried

out solely at the contract or API level by taking advantage of a hierarchical system structure, deducing higher-level contracts from lower layers and assuming elementary contracts as facts [LNR80]. Thus, it is possible to provide early feedback regarding correctness of the system specification. Later, the assumed facts have to be proven by verification of the corresponding implementation.

### 2.2 Runtime Verification

Runtime verification is regarded as the discipline of computer science dealing with techniques to monitor systems during runtime in order to detect violations of given correctness properties [LS09]. More recent research, however, also considers controlling the system via feedback as belonging to runtime verification as well [LS09; Ru16]. Augmenting a functional system with a corresponding management or control system allows for autonomous behaviour of the system during operation.

As previously explained, not all contracts can be statically verified at design time. Our methodology extends the verification activities to the system runtime. The amount of possible static analysis and needed dynamic verification depends on the specific system and the complexity of its constraints. During the design phase, our methodology aims to separate as many properties or partial predicates as possible that can be verified statically and to automatically derive the complementary properties or predicates that have to be ensured and verified at runtime. Concerning the application level, necessary monitors will be generated from the unverified SPARK contracts during the design phase and executed by the runtime architecture.

In principle, monitors may be associated with all critical parts of the system which may be a source of uncertainty, like the application itself, the operating system and the hardware, but also the environment, especially when considering embedded systems. If a monitor assertion fails, an event is signalled to the runtime system (see Fig. 2).

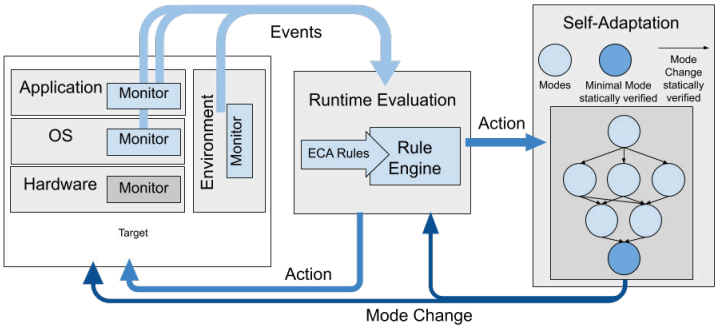


Fig. 2: Architecture for Runtime Verification

Thus, by extending the verification process to runtime, the methodology supports finding a manageable approach to verification of complex systems also covering uncertainty. This is especially important when static verification is based on pre-conditions, whose outcomes

can only be evaluated during runtime, e.g. if the pre-conditions are depending on user input or environmental factors.

As a disadvantage, incomplete static verification of system properties may result in property violations at runtime. Thus, provisions for events originating from failing runtime verification may be required. This is considered in the following section.

### 2.3 Adaptation Architecture

In the runtime system, signalled events will be received by an Event Condition Action (ECA) rule engine. Based on an application-dependent statically defined rule set, appropriate actions can be taken for detected runtime violations. Such actions can lead to a simple reconfiguration, like changing control parameters, or may require a more complex adaptation of the system.

As a basis for adaptation, the well-known concept of operational modes will be used. At each point in time, the system runs in a certain mode determining the provided functionality. At design time, a mode change graph will be developed, defining the set of operational modes and allowed changes. Also, the mode change algorithm must be statically verified for correctness. At runtime, mode changes may take place in response to signalled events. Thus, the system is guaranteed to be able to enter a well-defined state in case of a runtime verification failure, thus supporting graceful degradation.

In case of safety-critical systems, not every mode must ensure safety. A so-called "Minimal Mode"providing a safe state for the system is assumed to exist, whose functionality is statically verified. Due to the verified mode change algorithm, the minimal element can be reached in any case. Thus, a minimal level of service is ensured under all conditions.

### 2.4 Current Status

For initial evaluation, the functional part of the methodology was applied to the design and implementation of a queueing system and its use inside the scheduler of our microkernel Marron. Marron was developed by our group to serve as a template for a future verified microkernel. Besides scheduling, it features strict separation between user and kernel space, interrupt handling and inter-task communication. The queues where designed directly in SysML and constraints were defined in OCL. An appropriate DSL will be specified later, when more experience has been made. The use of SysML or OCL features was manually restricted to fit SPARK 2014 capabilities, and the transformation was done manually for now. The specification was verified based on the SPARK API. Necessary facts that have to be verified by the implementation were indicated by the `assume` pragma. Finally, the implementation was verified separately.

Based on the verified queueing system, a graduate student developed and verified the Marron scheduler in SPARK. The student had no prior experience with Ada, SPARK or formal verification in general. The goal of this experiment was also to evaluate the efficiency of our approach by measuring the effort needed to develop a verified operating system component.

The student spent a total effort of 450h over a course of six months, including literature work, project management, documentation, etc. He was able to verify 216 out of 224 verification conditions (VCs). The measured efforts spent on implementation and verification, as well as the relative effort in minutes per line of code are shown in Table 1.

| | **hours** | **loc** | **ratio (min/loc)** |
|---|---|---|---|
| **implementation** | 62.25 | 296 | 4.7 |
| **verification** | 95.25 | 330 | 19.31 |

Tab. 1: Effort analysis for implementation and verification of an OS scheduler in SPARK.

Regarding runtime verification of the remaining eight statically unverified VCs, the pragma `Assertion_Policy` was simply used to execute all contracts as assertions during runtime, but no exhaustive tests were yet carried out, nor have the predicates for runtime verification been optimized. The adaptation architecture has not been implemented yet.

## 3 Non-Functional Verification

Non-functional properties in the context of this work refer to the timing behaviour of a system. In order to reason about the timing of a system, a notion of time, a specification of timing properties and constraints and also a verification environment are needed. To formally verify timing behaviours means to find a mathematical proof showing that, under all conditions, the system will behave temporally as specified.

### 3.1 Modelling and Verification of Timing Behaviour

There is a long history of formal languages that can be applied to specify diverse aspects of timing behaviours [Wa04]. The classical event-oriented temporal logics such as Linear-Time Propositional Temporal Logic (LPTL) or Computation Tree Logic (CTL) only model the temporal order of events (e.g. before, after, always, never, eventually, ...), but do not provide a notion of real, physical time, as is needed to model real-time systems. One possible language to start with is called Timed CTL* (TCTL*). It is the foundation of the UPPAAL verification framework [20d], which is based on model checking techniques. However, such an approach often leads to state explosion or undecidability problems, making it impractical for complex real systems. Our method wants to avoid these limitations by keeping the human in charge as director for the proof, aided by semi-automated theorem provers like Coq [MT18].

### 3.2 WCET Estimation

In order to check whether a real-time program temporally behaves as specified in a model, it is necessary to know the actual execution times of relevant program sections, and to associate states in the model with program states.

The actual execution time of any piece of code can vary each time the code is executed. In real-time systems, the *Worst Case Execution Time* (WCET) is a commonly used concept to abstract from these variations. A good overview of the classification and techniques for WCET calculation can be found in [Ca19]. The paper differentiates between static, measurement-based or a combined approach to determine the WCET, each in a deterministic

or probabilistic variant. The static deterministic approach, called Static Deterministic Timing Analysis (SDTA), uses symbolic execution on an accurate model of the hardware. This approach is only practical for systems which are amenable to modelling, but it is well trusted and well established in industry. However, as today's multicore hardware achitectures frequently do not fulfil the assumptions made for their modelling, newer methods combine these approaches with probabilistic ones such as Extreme Value Theory (EVT). These are subject of ongoing research [Ca19].

### 3.3 Current Status

We evaluated the AbsInt tool for SDTA named aiT [20a] with a small application from the AQUAS space usecase and compared the WCET bounds with real measurements. The application cyclically receives a message from a serial communication interface, encrypts the message and sends it out over another serial communication interface. This application was executed on top of two different operating systems: (1) the library-based RTEMS kernel [20c] designed for microcontrollers and mostly used in avionic and space systems, and (2) our own microkernel Marron equipped with a small RTEMS adaptation layer which currently only implements the interface subset needed by the application. As target hardware, we use the TI TMS570 microcontroller with two ARM Cortex-R4 in lock-step mode. Marron was originally designed to run on Cortex-A multicore processors, but these more complex processors are not supported by aiT. The ARM Cortex-R4 was the smallest processor on which our system runs without modification.

| | **AIS2 code** | **Infeasible routines** | **Analysis times** | **aiT WCET** | **Max Exec. Time** |
|---|---|---|---|---|---|
| **RTEMS** | 659 loc | 36 | 40 s | 0.659 ms | 0.321 ms |
| **Marron** | 476 loc | 23 | 4 s | 0.580 ms | 0.247 ms |

Tab. 2: WCET and Measurement Results

The results of the WCET analysis are presented in Table 2. Annotations of the source code using the AbsInt AIS2 language are instructions to the AbsInt tools directing the static analysis. Declaring routines as infeasible means that the developer is sure they are never executed by the analysed code (e.g. POSIX and kernel error handling) and thus, they do not contribute to the estimated WCET.

We also compared the WCET estimations with real measurements of the execution times of the application. The execution time measurement starts upon reception of the first byte and ends when the last byte is sent out, thus being the same code sequence as for the WCET analysis. Both versions were compiled for ARM Thumb Code and with optimisation set to -Og (i.e. weak, "debug-friendly" optimisation). For the measurements we use the ARM Performance Monitoring Unit which measures the number of elapsed CPU cycles. All measured data was buffered in SRAM, as SRAM accesses are deterministic on the used platform. The buffering overhead was determined in a separate measurement and subtracted for compensation. For the static analysis, the buffering overhead was excluded by appropriate annotations. The measurement overhead itself was measured to be 60 CPU cycles based on 10 x 1000 single measurements. The aiT tool estimated the WCET for one

measurement to be 77 CPU cycles. The density function of the measured execution times for

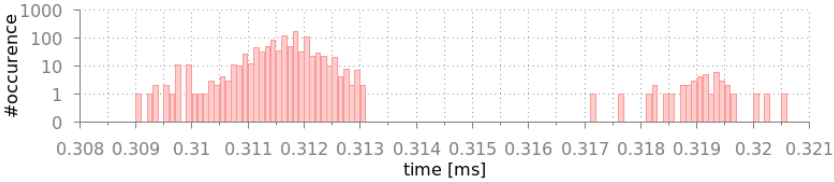


Fig. 3: Measurement Results for RTEMS

the RTEMS version is shown in Fig. 3, for the Marron version in Fig. 4. RTEMS execution times below 0.3131ms are caused by the communication through a software queue between the application and the receiver interrupt. The execution times above 0.317ms are the result of the same communication delayed by the system timer interrupt. This interference does not appear in the WCET analysis, as the tools do not model task communication or processor interrupts. The Marron RTEMS adaptation layer does not provide software queues for

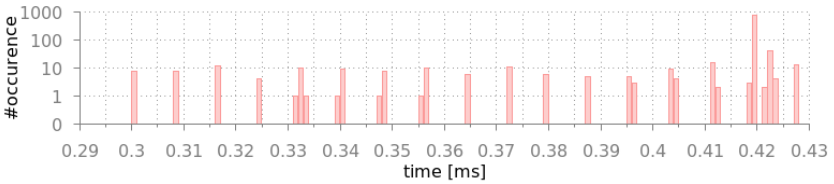


Fig. 4: Measurement Results for Marron

signalling, the only buffering mechanism used are the hardware FIFOs built into the serial interfaces. If no new data is available, the layer waits for the interrupt through a system call in user space. Execution times below 0.45ms result from rare cases of beneficial interference between the application and the system timer interrupt: If the application is interrupted before the hardware FIFO is checked, it is possible that the execution time will be shorter because new data arrived while the system timer interrupt was being processed. In this case, the data is ready upon exit from the timer interrupt and the receiving task does not need to wait for a receive interrupt.

Comparing the measurement results for RTEMS and Marron with the correspnding WCETs in Table 2 we can detect an overestimation of the WCET. This can be reduced up to a certain degree with many more annotations. Up to this point there were 120 hours spent in WCET analysis including a training period and meetings.

## 4 Conclusion and Outlook

In this paper, we presented the early stage of a method aiming at tool-supported verification to complex embedded systems, considering both functional properties and timing as a non-functional property. For functional properties, simple examples were used to evaluate parts of the methodology with promising results. However, more complex, realistic problems need to be considered and the methodology needs to be developed further. For timing properties, static WCET estimations were compared against measured execution times. It

turns out that static methods are difficult to apply to today's increasingly complex multicore hardware architectures, especially when these were not designed for determinism. In order to model worst-case behaviour for such architectures, very pessimistic assumptions need to be made, correspondingly leading to pessimistic WCET estimations.

To deal with these problems, monitors observing execution times could be generated from the timing specification and serve as a basis for supplementing static WCET estimation with runtime verification. This would lead to a similar approach as presented above for functional verification. Such a method would belong to the class of measurement-based probabilistic timing analysis methods in the sense of [Ca19]. A unified approach for the verification of functional as well as timing properties, supplementing static verification with runtime verification such that even complex systems remain controllable seems to be feasible.

**Acknowledgements:** This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737475. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Spain, France, United Kingdom, Austria, Italy, Czech Republic, Germany. This project has also received funding from the Federal Ministry of Education and Research (BMBF) under agreement No 16ESE0157. We would like to give special thanks to the people from AbsInt Angewandte Informatik GmbH for their support and Thales Alenia Space for the usecase application.

## References

[20a] AbsInt aiT, Apr. 2020, URL: https://www.absint.com/ait/.

[20b] AQUAS EU Project, 2020, URL: https://aquas-project.eu.

[20c] RTEMS Real Time Operating System, Feb. 2020, URL: https://rtems.org/.

[20d] UPPAAL, 2020, URL: http://www.uppaal.org/.

[Ca19] Cazorla, F. J.; Kosmidis, L.; Mezzetti, E.; Hernandez, C.; Abella, J.; Vardanega, T.: Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey. ACM Comput. Surv. 52/1, Feb. 2019.

[LNR80] Levitt, K. N.; Neumann, P. G.; Robinson, L.: The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software. In: Report 500-67. SRI International, Menlo Park, NBS, 1980.

[LS09] Leucker, M.; Schallhart, C.: A Brief Account of Runtime Verification. Journal of Logic and Algebraic Programming 78/5, pp. 293–303, May 2009.

[MT18] Mahboubi, A.; Tassi, E.: Mathematical Components, Creative Commons License, 2018, URL: https://math-comp.github.io/mcb/.

[Ru16] Rufino, J.: Towards integration of adaptability and non-intrusive runtime verification in avionic systems. ACM SIGBED Review 13/, pp. 60–65, Mar. 2016.

[Wa04] Wang, F.: Formal verification of timed systems: a survey and perspective. Proceedings of the IEEE 92/8, pp. 1283–1305, Aug. 2004.