Using Timed Automata in Requirements Analysis for Engine Control Units

Sebastian Grobosch∗ Volker Kamin∗∗
Kriengkrai Krirkkawin∗ Stefan Kowalewski∗∗

VEMAC GmbH & Co. KG (lastname@vemac.de)
Embedded Software Laboratory RWTH Aachen University
(lastname@embedded.rwth-aachen.de)

Abstract: To reduce the development costs for engine control units in small series, uniform software platforms are used. On the customer side, however, non-uniform systems are required. In order to realize different customer projects a tool set is needed to add functions to a common software basis. The resulting software variants, which consist of modules arranged by the costumer, are key drivers for the selection of the microcontrollers used in the system designs. Each software module has real-time requirements that must be fulfilled for the whole system to work properly. Violations of these real-time requirements should become visible during requirements analysis to avoid expensive redesigns.

The state of practice is to oversize the target hardware based on experience in order to minimize the risk that not all requirements can be fulfilled with the chosen hardware. In this paper, we show how a combination of timed automata analysis and variant management supports consistency during the development process and guarantees compliance with real-time requirements.

1. INTRODUCTION

This paper presents a novel approach to combine timed automata analysis and variant management in order to help developers to select cost-optimal hardware platforms for real-time controllers. Our method supports developers during the requirements analysis step of the V-Model, when cost estimates have to be made although the complete design has not been developed yet.

One major issue during the development of engine control units (ECU) for small series are the costs for hardware and software development. These costs are spread among just a small amount of units. Compared to high volume series the effect on the unit price is conspicuous.

Therefore, small and medium enterprises (SME) try to use uniform hardware platforms that can be used for various projects. On these platforms, the microcontroller unit (MCU) that is the main cost factor can be easily replaced by any MCU within its family. This allows cost optimization by selecting an appropriate MCU without the need for a costly hardware redesign. Additionally, using the same platform for multiple projects is beneficial due to high volume series. On the software side, a uniform hardware platform allows the development of a framework with interchangeable software modules. These modules can be arranged flexible to cover a wide range of applications. The idea is to provide single software modules that can be selected by the customer out-of-the-box to generate the software for the uniform hardware platform.

The ECUs evaluated for our case study are commonly used to run, e.g., chain saws, go-carts, and small aircrafts. This way, the price per unit is lower than the price a conventional small series development would yield.

Our approach focuses on ECUs for small-power combustion engines that run on MCUs, but it can be applied to other fields as well due to its high degree of abstraction.

1.1 Problem Description

Customers demand cost estimates already during requirements analysis, although this requires the selection of a target hardware that, according to the V-Model (IABG [2009]), should be done in a later phase of the development process. Due to the huge amount of different combinations of the out-of-the-box software modules an intuitive selection is uncertain. For each interesting combination of functions, software engineers need to find a MCU that is able to realize this combination and which costs as little as possible. Currently, this is done by oversizing the microcontroller based on experience in order to minimize the risk of selecting a MCU that turns out to be unsuitable.

In order to reduce oversizing, which results in less attractive cost estimates, developers need to be able to verify in advance if a certain MCU is suitable. To support the developer during this estimation process, information about certain properties of out-of-the-box software modules should be accessible, e.g., worst-case execution times (WCET). An early prototype of the software helps to verify properties like needed program memory and maximum stack size. Furthermore, execution time limits for currently unimplemented new software modules requested by the customer can be defined before the implementation phase.
These early limitations should guarantee the unproblematic interaction with the other selected software modules running on the MCU.

For real-time requirements, however, this estimation cannot be done by just measuring the execution times of each function using the prototype. Adding maximum execution times can identify a MCU as unsuitable, but it cannot guarantee that all software components will finish their computations in time (i.e., meet their deadlines). In order to verify the fulfillment of all real-time requirements, all possible executions have to be considered. Therefore, a formal approach is necessary.

1.2 Overview

Following this introduction, Sect. 2 introduces related work. The variant framework and its process flow is the topic of Sect. 3. Next, Sect. 4 describes our abstract timed automata model and explains its components in detail. The evaluation of the approach is shown in Sect. 5. Finally, Sect. 6 gives a conclusion and an outlook on future work.

2. RELATED WORK

The scheduling analysis with monotonic rates by Liu and Layland [1973] is a good start for analyzing the real-time requirements. This approach is suitable when using task deadlines which are not equal to their periods. In our case study, task periods can differ from their deadlines, which is why we need a more general approach.

WCET analysis is a technique to gather timing information about single software functions (Kirner et al. [2004]). The interaction of different software modules is not included. The important property for schedulability analysis is the worst-case response time (WCRT).

Standard WCRT analysis approaches of real-time tasks use so called critical instant for the calculation (Bril et al. [2007]). So as not to overapproximate one needs to know which task constellation is possible due to the system behavior. Our approach of schedulability analysis with timed automata (Ahur and Dill [1990]) combines WCRT analysis with a realistic model of the software system to minimize this overapproximation.

A widely known static analyzer that can compute WCETs is\textsc{AbsInt} (Heckmann et al. [2004]). It uses binary code as input for its analysis and utilizes detailed hardware models to obtain a precise overapproximation. Unfortunately, it does not support the low budget microcontroller we focus on.

Static analysis is a widely known technology that has been successfully applied to embedded software. Abstract interpretation (Cousot and Cousot [1977]), for example, is used to combat stack overflows (Regehr et al. [2005]) and determine WCET (Lundqvist and Stenström [1999]).

A tool for schedulability analysis for real-time tasks using timed automata is\textsc{TIMES} (Amnell et al. [2003]). The tool solves the schedulability analysis by reachability analysis on timed automata. Unfortunately, the project development has stopped in a state that does not allow its integration in our approach.\textsc{TIMES} utilizes the timed model checker\textsc{Uppaal} (Larsen et al. [1997]) that is also used in our approach.

For handling different software modules the approach of variant management is widely known (Dalgarno et al. [2008]). While this kind of tools can combine different software modules to a resulting software system, they cannot verify the schedulability of the resulting system. A combination of different approaches is necessary.

To the best of our knowledge, there is no approach supporting developers in choosing target hardware during requirements analysis considering real-time requirements and variant management.

3. VARIANT FRAMEWORK

To support the developer during the cost estimation process, we choose variant management to provide an early C-code prototype of the software system and to generate a\textsc{Uppaal} model for schedulability analysis. To improve consistency throughout the development process, the variant management also stores data regarding each software module (e.g., WCET) and also manages test scenarios and test queries. Such improvements are expected to increase software quality and support reproducibility of variants.

Furthermore, they should improve maintainability and reusability of software modules provided.

3.1 Overview

As described before, for various variants of a software system the developers are required to provide a cost efficient product to the customer. These variants differ in software modules which are used to cover functional requirements. Figure 1 shows the process of variant management. The variant system combines various functions to a new system. Next to the system architecture including, e.g., the scheduler, functions such as special controllers for combustion engines, bus system drivers, and basic I/O drivers are provided. Each function is defined by a module which consists of its meta data and the concrete implementation.

The meta data, which is provided by a XML database, is passed to the variant management where it is used together with the variant model to generate the new variant. The XML data is than processed by a transformation engine to generate the resulting documents (C-code and \textsc{Uppaal} model).

3.2 Process Flow

The main idea using the variant management is to separate information about interfaces and the implementation from the architecture. This information is stored as meta data (see Fig. 1) in a database, which is realized as an XML file. The software architecture is represented as template files with defined placeholders used as interfaces to the XML database. The variant management combines this meta data with the appropriate template files to generate C-code and the\textsc{Uppaal} model. The C-code is compiled, linked, and flashed onto the MCU to obtain an early running prototype. With the generated\textsc{Uppaal} model, scheduling analysis (see Sect. 4) can be performed to
check whether the selected variant will run on the microcontroller. The transformation process is done with the help of XSLT (W3C [2010]) and SED (SED [2010]) that generates the output documents using the meta data and the template files.

The templates files for the C-code represent the software architecture of the microcontroller code. There are start-up files which initialize, e.g., the oscillator, memory and peripheral devices of the microcontroller hardware as well as setup vector tables that link all interrupts to specific C-code functions. Beyond these hardware dependent components, the template files also include the task scheduler and the entry point function (main()). Each module can use these template files by adding own C-code at defined placeholders in every .c and .h template file. In main() a placeholder is set up where all modules can insert their initializing routines, which are executed after the boot up of the microcontroller. There is also a placeholder where various I/O ports provided by the microcontroller need to be activated for later usage by the module. Further #include statements can be inserted to provide module functions to the rest of the system. In the interrupt vector table section a placeholder is used to define which interrupt service routine is executed when a specific interrupt is triggered. If a module provides such a routine, the function name needs to be inserted here. For the communication between different modules, global variables are used. These variables need to be defined at the appropriate placeholder.

The second type of template files is used for the UPPAAL model. There, the prototypes of the tasks used are defined by a set of parameters (see sec. 4.2). These files also include the scheduler which is independent from the tasks used.

3.3 Meta Data

Every module consists of various data which is stored in the XML database. As described above, this data is used to generate C-code and the UPPAAL model together with the template files. The available properties are described as follows:

- Library inclusion: If the module needs special library functions (e.g., math.h) they can be included into the software system by using this property.
- External peripheral settings: During the boot-up process on the microcontroller several external peripheral functions are initialized. To use these peripherals in a module they have to be initialized first. This can be done by parameterize them with this property.
- Initialization routine: After the boot-up process every module can run a defined routine once to initialize the module, e.g., to set default values to variables or outputs.
- Global communication: To enable data exchange between different modules, this property can be used to specify global variables. Such variables are accessible for other modules.
- Scheduler data: The scheduler provides periodic tasks with different periods (100 ms, 500 ms and "as fast as possible"). Every module can add its own routines to the periodic task queues, which are than executed with the appropriate period.
- Interrupt behavior: Several interrupts are provided by the microcontroller. To use these interrupts every module can add routines to the appropriate interrupt service routine which is executed when the interrupt occurs. Furthermore this property is used to update the interrupt vector table to register an interrupt to the microcontroller.

Figure 2 illustrates the process of the transformation at the example of the global variables. Two modules A and B have been selected by the customer. The C-code template file contains the placeholder GLOBALVAR where the global variables of all modules should be inserted. Both modules want to use a global variable which is inserted into the C-code template during the transformation process. The resulting C-code file is than ready to compile.

4. ABSTRACT MODEL

In order to solve the problem of checking real-time requirements of software running on a given microcontroller as introduced above, we created an abstract model based on ideas of David et al. [2010]. We extended these approaches for the use of microcontroller systems. To model the behaviour of interrupts used in microcontrollers we used preemption and we introduced preemptible and non-preemptible tasks. Furthermore, we added the calculation of WCRT for each task. The system consists of one scheduler and multiple tasks, which we modeled using timed automata (Alur and Dill [1990]).

These components form a network of timed automata (Behrmann et al. [2004]), for which real-time properties can be verified using the model checker UPPAAL (Larsen et al. [1997]). First, this section gives an overview of the components of the abstract model and describes how they interact. Then, the scheduler and tasks are explained in detail.
4.1 Overview

The abstract model consists of multiple tasks and a scheduler controlling these tasks. Each component is modeled as a timed automaton. Tasks request computing time and the scheduler grants and revokes it. This is done by passing messages from one timed automaton to another forming a network of timed automata. Note that these messages do not represent actual communication between software components but might stand for events such as the activation of an interrupt service routine. Therefore, tasks only need to communicate with the scheduler and not with each other.

When a task needs computation time, it sends an activate message to the scheduler. At the same time it announces whether it is able to interrupt other tasks and whether it may be interrupted itself. If a task that is able to interrupt other tasks becomes active and the task that currently is assigned computation time allows itself to be interrupted, the current task is preempted. Otherwise, the current task keeps running until it sends a terminate message to the scheduler. Each time a task is preempted or terminates, the scheduler checks which active task has the highest priority and sends a start message to that task. A task that sends an activate message while no other task is running is started immediately.

Each task has two clocks. One stops the time from activation while the other one stops the computation time of that task. The former is used to verify that no task violates its deadline; the latter ensures that each task cannot use up more computation time than its WCET to eliminate spurious counterexamples. The scheduler does not stop time, but it ensures that only one task is granted computation time at a time and that priorities are considered properly. Using this level of abstraction, the model can be used for any kind of microcontroller while still considering microcontroller specific behavior (e.g., interrupt service routines). Due to the flexibility of UPPAAL these abstract models can be extended to represent more complex systems and to increase accuracy of the results.

In our abstract model, we utilize stop watches. We are, however, not concerned with decidability issues because UPPAAL guarantees termination by using overapproximations (see David et al. [2010]).

4.2 Tasks

The timed automata modeling tasks in our abstract model contain all the information that is needed to verify compliance with real-time properties. Each task is parameterized using six properties: ID, periodicity, WCET, deadline, preemptable and interrupt.

ID is a unique number that is used by the scheduler to address tasks. Currently, the function determining the priority of a task also uses ID because we have no need for dynamic priorities yet. In embedded systems priorities are, e.g., used to define the order in which interrupts are executed.

Periodicity specifies how frequently a task is activated. Some tasks are activated in fixed time intervals, e.g., 1 ms. Other tasks are activated as a reaction to an irregular external trigger. In the latter cases, worst-case periods are assumed in order to obtain valid overapproximations. Deadline is assigned the maximum amount of time that may pass after activation before a task has to finish its computation.

Preemptable and interrupt are microcontroller specific properties. Preemptable indicates whether a task is atomic or can be interrupted by other tasks. Interrupt declares whether a task is part of the main program or an interrupt service routine, which is activated by hardware when a certain event occurs. Usually, exactly one of these properties is true, but there are exceptions. On one hand, time critical parts of the main program may be protected from interrupts. On the other hand, interrupts, which by default cannot be interrupted, may explicitly allow other interrupt service routines to preempt them. The latter leads to so-called nested interrupts, which cause great problems when using static analysis and other formal methods that do not consider timing (Regehr et al. [2005]).

These six parameters are used to instantiate a template shown in Fig. 3. For clarity, we simplified the original timed automata. Initially, a task is Suspended, which is a urgent state. Urgency is a concept that ensures that the state is left immediately (i.e., no time may pass while any component is in an urgent state) and, therefore eliminates spurious behavior such as a component that simply does not announce its readiness. Using two internal states not depicted in Fig. 3 the task sends an activate message to the scheduler including ID, preemptable, and interrupt. At the same time, the activation time (AT) clock and execution time (ET) clock are reset and AT is started. The task is now Ready and waits for the start message from the scheduler allowing it to enter the Running state. As soon as the task reaches this state, ET is started. If the task is preempted, only ET is stopped and the task returns to the Ready state waiting for another start message. When ET reaches WCET of the task, it is forced to leave the Running state by an invariant. Internal states not shown in Fig. 3 ensure that such a task is not preempted, which would cause spurious counterexamples. Instead, it is checked whether the task finished its computation in time, i.e., whether AT is greater than deadline. If the deadline has been violated, the Error state is reached, otherwise the task enters EndPeriod sending a terminate message. In this state time passes until the next period of that task starts and an invariant forces the task to advance to its Suspended state again.

4.3 Scheduler

As mentioned above, the scheduler does not contain any information about real-time properties and, therefore, does not need to be parameterized. Its duty is to guarantee that tasks are assigned computing time in order of their priorities, that the CPU does not run idle while tasks are ready, and that at most one task is running at a time.

The scheduler model shown in Fig. 4 represents the behavior of the software scheduler and the hardware interrupt system. Again, the figure has been simplified for clarity. The scheduler model contains three main locations named Idle, Ready and Busy which represent the states of the
CPU. The *Idle* location is used when no CPU time is required. Here, time passes until a task sends an *activate* message. Using internal states *ID*, *preemptable*, and *interrupt* are received. In the *Ready* state, the scheduler determines which task to start using a priority function. Here, no time may pass, which is ensured by using a variant of the urgency concept. Therefore, a *start* message is sent immediately to the most important task, causing the scheduler to enter the *Busy* location. Whenever a task sends an *activate* message, a chain of urgent internal states are traversed. If an interrupt sent the message and the currently running task is preemptable, it is stopped and the scheduler enters the *Ready* state, yielding the start of that interrupt. Otherwise, the scheduler immediately returns to the *Busy* state letting the current task continue its computation.

While it is possible to merge *Idle* and *Busy*, we kept them separate in order to allow a more meaningful visualization using Gantt charts offered by UPPAAL TIGA (Behrmann et al. [2007]).

---

**Fig. 5. Response time**

5. EVALUATION

To evaluate the approach described above, we generated two different variants with the help of pure::variants (Beuche [2008]) and instantiated corresponding UPPAAL models. For the evaluation of the scheduling analysis, we used the following two scenarios. The first one is the evaluation of the UPPAAL model. We test the behavior of preemption and prioritization of a module combination. The second scenario calculates the WCRT for a variant with UPPAAL and compares this result to the measurements of the real system.

For the first scenario, we used two modules for the variant. The first module consists of the task TRA that is a periodic task (period: 1000 µs, deadline: 1000 µs, execution time: 929 µs, preemptible: true, interrupt: no). The second module provides the task INTO which is an interrupt task (execution time: 258 µs, preemptible: true, interrupt: true). This task depends on the current engine speed, so the period can differ. We assume the period to be worst-case (5000 µs), which corresponds to an engine speed of 12000 RPM. After the generation process done by the variant management, the C-code is compiled and uploaded to the ECU. Using the integrated timer on the microcontroller, the response times of TRA are measured as shown in Fig. 5. The experiment was performed at two different engine speeds (6000 RPM and 12000 RPM). It is conspicuous that every 5 ms or 10 ms respectively the response time of TRA is increased by the execution time of INTO. This behaviour will force the deadline of TRA to be violated.

To prove this behaviour, we used the verification engine with the generated UPPAAL model and the following queries. With "A[][ not deadlock" we checked if a system wide deadlock can occur. The verifier outputs that this property may be satisfied. Note that the use of stopwatches creates an over-approximation of the state space. If the query is satisfied, it is guaranteed that the system is schedulable under all circumstances. But the query is not satisfied. This means that a counter-example exists in the overapproximation. In order, to check if this counter-example is a feasible run of the original system, we checked if the *Error* states are reachable. Therefore, we used the query "A[][ not TRA.Error" for task TRA and "A[][ not INTO.Error" for task INTO, respectively. For INTO this property was satisfied, but the *Error* state of TRA may be reached. So, we checked the WCRT of both tasks. Using the query "sup: INTO.wcrt,TRA.wcrt", we got both WCRT times by using the supremum function for additional clocks, named *wcrt*, in our model. The WCRT of TRA is higher than its deadline, which is a violation of the real-time requirements. Therefore, the system is...
not schedulable. Uppaal needed less than a minute on a standard PC to verify these queries.

The second scenario consists of a variant with two tasks: A periodic task which is called every 100 ms. This task is preemptible. The second task has a higher priority and a period of 500 ms. Fig. 6 shows the response time of the 100 ms task over a period of 5 s. The response time varies between 1700 µs and 2250 µs depending on the occurrence of the task with the 500 ms period. During the 5 s of measurement the maximum response time is measured to 2250 µs. Comparing this value with the WCRT calculated with the Uppaal model (2401 µs) the difference is 151 µs which is about 6% of the Uppaal result. One reason for this difference could be the inaccuracy of the input times for the Uppaal model. The execution times used above are measured and not calculated. Therefore, the Uppaal model gives an overapproximation of the WCRT of the chosen task.

6. CONCLUSION AND FUTURE WORK

This paper described the problem of selecting optimal MCUs for certain combinations of software modules during the requirements analysis. Furthermore, this paper introduced a software framework to support the developer during the cost estimation process with a variant management and a scheduling analysis. The variant management is able to provide an early C-code prototype of the software system next to an instantiated Uppaal model for the scheduling analysis. Additionally, it stores all important information about a software module such as the tasks parameters, the C-code, and the test cases, in one central database. Hence, maintainability and consistency throughout the whole development process is improved. We introduced an abstract model based on timed automata for scheduling analysis that can be used to verify real-time requirements of a software system. This verification allows developers to refrain from further oversizing the MCU. Additionally, early limitations of execution times for new software modules can be defined and evaluated within the whole software system. We evaluated this approach in our case study using an ECU and showed that it can be used to verify the fulfillment of real-time requirements.

In future, we will extend our abstract model by replacing the static priority function with a dynamic one to support other scheduling strategies and with it open our approach to more fields of applications. Furthermore, an evaluation using more complex functions will show the scalability of our approach. A more precise computation of meta data for single tasks might improve the overall precision. Therefore, we plan to use static analysis on reusable code fragments to obtain better meta data.

REFERENCES


W3C. Xsl transformations 2.0, 2010. URL http://www.w3.org/TR/xslt20/.