Timing Analysis of Safety-Critical Automotive Software: The AUTOSAFE Tool Flow

M. Becker[1], S. Mohamed[2], K. Albers[3], P.P. Chakrabarti[2], S. Chakraborty[1], P. Dasgupta[2], S. Dey[2], R. Metta[4]
[1]TU München, Germany, [2]IIT Kharagpur, India, [3]Inchron GmbH, Germany, [4]Tata Consultancy Services, India

Abstract—**Automotive software applications implement a variety of control algorithms, with many of them being safety-critical in nature. A typical design flow starts with modeling these control algorithms using tools like MATLAB/Simulink. However, at this stage, a number of assumptions, like negligible sensor-to-actuator delay and instantaneous computation of the controller software, are often made. In particular, the details of the software implementation and the computing platform, both eventually defining the timing properties of the applications, are not accounted for. Such idealistic assumptions can cause a significant deviation of the control performance compared to what was proven at the modeling stage. This is usually addressed with multiple design iterations, which are costly and may lead to over-provisioned and thus poorly designed systems. In this paper we attempt to address this problem by proposing a design- and tool flow that integrates software- and platform-level timing information into the high-level modeling stage. We outline our proposed flow using concrete, industry-strength design tools.**

I. Introduction

Many automotive platforms today consist of 50-100 electronic control units (ECUs) that are connected by a heterogeneous communication architecture using buses like CAN, FlexRay and Ethernet. Such a platform runs several millions lines of software code, implementing various control algorithms spanning across safety-critical, driver assistance and comfort-related domains. The design flow for such a system starts with high-level modeling of the algorithms using tools like MATLAB/Simulink. At this stage – as is common in control theory – several idealistic assumptions, such as negligible sensor-to-actuator delay and instantaneous and perfectly periodic computation of the control law, are made. The control performance of the final implementation is then estimated based on these assumptions.

However, in such complex and distributed systems, the assumptions made at the model level are often not true. For example, the sensor-to-actuator delay of a distributed control loop is usually subject to bus communication delay, jitter due to arbitration, as well as jitter from scheduling effects on the ECUs. Consequently, the performance of the actual implementation deviates from the estimates, possibly violating requirements that have been proven at the modeling stage.

The information that would allow for a better estimate is only available later in the development process, when code is generated from these models and deployed on the distributed platform. This *semantic gap* between the *high-level models* and their *implementation* is often identified during integration tests, and fixed by going through several design iterations, until the performance is sufficient. However, those iterations are not only time-consuming, but also they leave the design without any analytic backup, hindering certification. In particular, a scientific basis for integrating software- and platform-level timing information into the high-level modeling stage is missing, with legacy and domain experience currently being the key influences. Software architectures like AUTOSAR are also not addressing this gap, instead they introduce even more abstraction of the processing platform.

On the other hand, elaborate tools modeling low-level platform behavior and allowing for precise simulation or timing analysis of distributed systems do exist, but little is known about approaches that can extract their timing information and use it in a meaningful way in the high-level models. In particular, such an integration is missing for commonly used tools like MATLAB and Simulink.

Closing the semantic gap would enable the automatic exploration of mapping and scheduling of control tasks on ECUs, and provide precise estimates of the final system performance, backed up by formal analysis. This paper lays down the blueprint of such a mechanism and presents the proof of concept for the proposed approach using industry-strength design tools, including Simulink [1], INCHRON Tool-Suite [2] and AutoGen [3]. The broad steps in our proposed workflow are as follows (see Figure 1):

1) Traditional MATLAB/Simulink modeling of the control system followed by code generation. A control-theoretic analysis provides bounds for possible sampling periods and tolerable deadline misses, which is captured in formal requirements to be fulfilled by the later implementation.
2) The control tasks are mapped onto ECUs in a chosen platform, which is possibly shared with other applications.
3) Perform worst-case execution time (WCET) analysis of each control task individually.
4) System-level timing analysis based on Real Time Calculus (RTC), accounting for platform-specific bus latency, task scheduling and preemption strategies. All feasible behavior of the distributed system is covered.
5) Automatic generation of timed automata models from the RTC result. Models can be generated at various levels of abstraction, matching the required level of precision.
6) Formally prove that the timed automata, and therefore the final system, admit only those behaviors that guarantee the control performance requirement of Step 1. For this purpose we use the UPPAAL model-checker.

The primary contribution of this paper is to lay down the foundations of the AUTOSAFE tool flow and present a proof-of-concept. The remainder of this paper presents insights to each of the above steps using a representative automotive study as a test case.

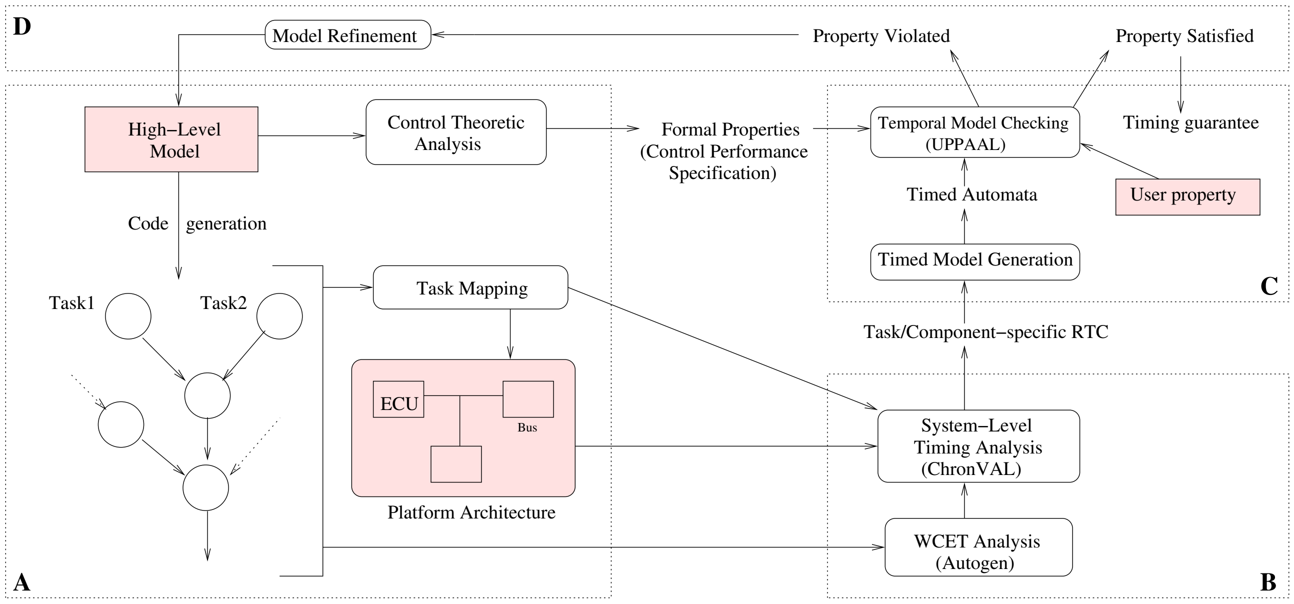


Fig. 1: AUTOSAFE Workflow (simplified figure only showing the verification parts; simulation omitted).

II. The Tool Flow

Our overall tool flow, as shown in Fig. 1, consists of the following primary components, A) a high-level model, code generation and task mapping front-end, B) an architecture-sensitive timing analysis engine, C) a back-end for automata-based verification of timing properties and D) a back-annotation mechanism of timing properties to the high-level model.

A. From High-Level Specification to Task Mapping

The development process starts with a specification of the functional behavior of the system. Towards this, the developer may chose any suitable modeling framework or tool which can generate an implementation of the modeled system in the form of C code.

The system may consist of numerous interdependent, sequential or parallel *tasks*, interacting with each other to produce the desired behavior. Moreover, the system may be composed from multiple independent task *clusters*. For example, our approach allows simultaneous modeling and verification of both an anti-lock braking system and an adaptive cruise control, which do not have any interdependence, but may be sharing resources, such as ECUs and buses. At this stage of the development, neither the synchronization nor the communication method between those tasks has to be specified; only their data flows.

After the functional modeling is complete, we generate C code for each such task separately. Additionally, a formal specification for each task is emitted, which is used later on during the analysis. This can be seen as a set of desired attributes that shall be achieved by the implementation. The attributes of this specification depend on the nature of the high-level model, and the properties that are verified. Examples are the execution pattern of the task (periodic, sporadic, event-triggered), task deadlines, or even a performance metric that can be evaluated when timing data is available.

As a next step, the developer creates a model of the distributed system architecture, which specifies the existing ECUs, their processors and OS types, their interconnections (e.g., FlexRay buses) and other architectural elements.

Finally, the developer decides for a *mapping* of the tasks onto the ECUs, that is, the binding of the functional parts to the architectural blocks. Naturally, it is possible to map multiple tasks onto the same ECU, or to distribute interdependent tasks over a network of ECUs, as often the case in automotive systems. At the moment, this step takes place manually. An automatic design space exploration is in principle possible by iterating our entire workflow, see section II-D.

At this point, all the necessary information (platform definition, communication paths of distributed software, the actual software implementation, etc.) is captured and available to perform a detailed temporal and functional analysis of the distributed system. In the next section, we explain how this information is used to perform an arbitrarily precise timing analysis of the chosen system.

B. Timing Analysis

The goal of the timing analysis is to obtain a complete picture of the end-to-end timing of the system. This includes both the latency of computation (execution times, task scheduling, interrupts, etc.) and the latency of communication (forming messages, bus arbitration, transmission time, etc.).

The timing analysis consists of two steps: a) *task-level timing analysis* to obtain the worst-case execution time (WCET) of each task in the system and b) *system-level timing analysis* to compose the tasks, evaluate effects of sharing resources and finally obtaining the end-to-end delay.

Task-Level Timing Analysis: First, we compute the worst-case execution time (WCET) of each individual task in the system. The WCET of a task represents the maximum execution time that the task takes to finish computation on the specific ECU it is mapped onto, considering all possible inputs and the architectural features of the ECU, such as caches and pipelines. However, the WCET does not include delays due to task-external factors, such as interrupts.

Existing approaches to computing the WCET predominantly use integer linear programming and abstract interpretation. These approaches analyze the machine (assembly) code of each task for finding control flow and timing properties. However, they often need manual inputs, e.g., specification of loop bounds, and may not obtain a tight WCET bound. To overcome these two limitations, we are working on a new approach to obtain a tight WCET for each task, without requiring manual inputs.

The central idea in our approach is to perform the WCET analysis as close as possible to the source code, since data dependencies and flow analysis are not obfuscated by architectural details, and yet we retain the temporal behavior from the machine level, such as the instruction timing. Specifically, we chose to perform the analysis using model checking on an augmented version of the C source code.

For each task, we perform the following steps:

1) Cross-compilation of the C code for the target ECU to which the task is mapped,
2) flow analysis on the binary to obtain *basic blocks*,
3) lifting of the binary control flow into the C code,
4) back-annotation of basic block timing to the C code,
5) slicing, acceleration, abstraction and (optional) overapproximation of the C code for WCET computation, and,
6) perform model checking for automatic deduction of logical constraints and computation of WCET.

Whereas steps 1) – 2) are standard within WCET analysis techniques [4], at step 3) we deviate from known paths. Here, we first establish a mapping from the basic blocks as obtained in the previous step, to the control flow of the C source. However, *control flow differences* have to be expected. Even in the simplest case, the compiler may have generated function calls, jumps or loops in the binary that are not present in the source code.

Such flow differences are translated back from the binary to the source code, producing a new, *augmented source code*, which captures the control flow of the task on the specific target processor. In step 4), we then use the established mapping and annotate the instruction timing from the basic blocks in the source code, where a model checker can be used to compute the WCET, similar to the approach in [5].

However, model checking would not scale, if all details in the augmented C source were retained. Since we are only interested in the timing, it turns out that a lot of irrelevant details can be removed. For example, all variables not influencing the control flow can be removed. Also, the specific value of variables might not be important, but only their sign, or coarse range. Based on this strategy, we perform a slicing w.r.t. the timing annotations and perform loop abstractions similar to [6]. Finally, we run the model checker on the sliced and abstracted C code of each task, to find its WCET. In comparison to [5], our approach for WCET estimation is more precise due to the consideration of flow differences, and meanwhile more scalable, due to the slicing and abstraction. Both of these aspects become important in later stages of our toolchain, where a WCET is needed, and in particular a tight one, i.e., with little pessimism. The details of the above ideas is the subject matter of research paper we are currently working on and are out of scope for this paper.

Once the WCET for each task is available, preemptions due to other tasks mapped to the same ECU, the OS, as well as communication delays (inter-process communication, bus messages) between the tasks have to be considered. This is presented below.

System-Level Timing Analysis: Based on the execution times of the tasks, their interconnections, the stimulation (activation pattern) of the tasks, their mapping onto resources, the scheduling strategies and the communication protocols, the tool chronVAL from INCHRON, allows computing the timing behavior, like upper and lower bound of the end-to-end latencies, of the distributed system. This takes into account preemptions by other tasks, the real transfer times and scheduling on bus systems, but also effects like missed protocol slots and execution windows.

The tool chronVAL is based on real-time calculus [7], a method for schedulability analysis for distributed systems. The simplified method is to obtain for each task an *incoming upper/lower arrival curve*, describing the density of the activations for the task, and the *upper/lower service curves* describing the available capacity in terms of computation time. Together with the internal behavior of the task itself, like the WCET, it is possible to compute the remaining capacity and the density of the outgoing activations (outgoing arrival curve) for each task. The outgoing activations are then the incoming activations for the successor task, and the remaining capacity is the available capacity for the task with the next lower priority. Finally, the incoming capacity for the highest-priority task is usually the ideal capacity of the resource. For more complex systems with other scheduling strategies, such as bus protocols with blocking times, the schedulability analysis is more complex, but the principle remains the same.

The mentioned curves are functions on time intervals, extracting the cumulated worst-case/best-case behavior over all interval lengths. For example, an upper arrival curve returning five events for an interval of length 10ms, means that it is not possible to find any interval with a length of 10ms that has more than five events. All these curves are described by a set of triples $\{(\Delta t, n, s)\}$, where Δt is an interval length, n is the number of events or the amount of computation time and s is a slope. To limit the number of elements required to describe the curves, chronVAL uses an approximation when obtaining the curves as described in [8] and [9]. That simple description that captures all the possible behaviors including the worst-case and best-case, allows an efficient schedulability analysis and the computation of the required (end-to-end) response times.

C. Automata Generation and Formal Verification

The automata generation framework provides a mechanism for analyzing the platform-level timing results as derived by the tool flow, using established formal verification engines, so that system-level properties w.r.t. control performance can be derived, while taking into account the timing characteristics of the implementation platform. The system-level timing analysis can only give a schedulability guarantee. Our approach integrates the accurate timing data obtained from a low-level platform model with the high-level platform model, and through this enables performing functional verification with full timing data. Thus, our approach can formally verify end-to-end functional properties which depend on the platform timing behavior.

The timing analysis phase, as discussed earlier, helps in abstracting out the low-level timing details of the platform (latency, scheduling policies, etc.) resulting in RTC curves which capture task-level arrival and service patterns. The analysis also provides task-level worst-case response time (WCRT) information. For checking whether the timing constraints are consistent with the planned end-to-end properties that serve as indicators of control performance, a representative timed transition system of the implementation is synthesized for model checking using UPPAAL [10]. The generation of such a template model requires the synthesis of the following set of parallel timed automata:

1) A pair of synchronizing timed automata to generate the events within the RTC arrival bounds for every task. For generating events which satisfy periodic RTC arrival curves, we use a modified version of the method discussed in [11]. Our method can handle conjunctions of RTC constraints by generating the arrival curves on-the-fly for such complex constraints, to include any arbitrary set of arrival curves, both periodic and aperiodic.
2) A pair of timed automata which keeps track of the minimum and maximum service available for a task in a resource as per the RTC service curves. The minimum and maximum service available at any instant are stored as a global shared variable, which can be accessed by other automata.
3) A task automaton which keeps track of the task generation, execution and completion. It also ensures the conformity of the task with the timing constraints.
4) A scheduler automaton which releases the generated task from the task automaton in a resource according to a scheduling policy.
5) A resource automaton which simulates the resource usage. Resource refers to the processing and communication nodes in the system. It can be a processor, an ECU or a bus. The resource automaton synchronizes with the task automaton and the scheduler automaton.
6) An observer automaton which quantifies the parameters in the functional property to be verified.

The basic idea for this kind of framework to simulate the system is similar to the work reported in [12]. The set of automata we need to construct depends solely on the type of property we want to verify. For example, in order to verify some end-to-end functional property, we consider only the tasks associated with the property to be verified. The system-level timing analysis gives us the RTC bounds for input arrival and output completion of the task set under question. The process of RTC generation abstracts out the other tasks by considering their effects when generating the bounds. Using the RTC bounds as derived, we simulate the event generation automata, service automata and the task automata for the candidate tasks. This method of abstraction helps us to effectively tackle state space explosion, without compromising on the accuracy of the model.

The level of detail needed to represent the tasks and resources can be varied as per the property to be verified. The simplest case will be to consider the resource as a node, without considering the internal architecture of the processing unit. But if there are properties that directly refer to the internal architecture of a processing node, then we have to model the internal architecture of that processor as a set of timed automata. The worst case in our method occurs when we have to verify a property for which we will have to simulate the entire architecture as it is. In this case, the general problem of state space explosion still occurs and cannot be avoided.

A crucial outcome of our method is that we are able to give formal guarantees for functional correctness, even with deadline violations. In general, formal verification tool flows are often conceived to work under the assumption that all tasks meet their deadlines. In our method, we are generating RTC bounds for task arrival and completion. Some of the patterns within these bounds cause deadline violations for the tasks. The system-level timing analysis is not affected by these deadline violations. Hence, when we perform formal verification for properties over these RTC bounds, the guarantees are obtained for the entire set of task execution patterns, including the ones for which there exists some amount of deadline violations.

D. Closing the Loop

The overall tool flow initiates a design process with a high-level specification, followed by code generation, platform mapping and subsequent timing analysis which augments the model with timing bounds. As discussed, a timing-annotated system description is translated to a network of timed automata and model-checked for property verification. The properties are derived from some desired control performance criteria as mandated by the designers. The satisfaction of the properties acts as a formal guarantee provided by the tool flow. Otherwise, the model checker reports a counterexample in the form of a candidate scheduling trace which violates the property. In such cases, the platform is simulated with the counterexample trace. This helps to deduce whether the counterexample is actually a feasible one. Since the RTC data used as input excitations are approximations of actual execution patterns, they do admit such spurious counterexamples in the real system. In case such spuriousness can be inferred about a counterexample, respective RTCs may be modified to eliminate it. Subsequently, the entire tool flow is re-executed like a Counterexample Guided Abstraction Refinement (CEGAR) loop [13]. As of now, in our tool flow, the process of elimination of spurious counterexamples is not automated.

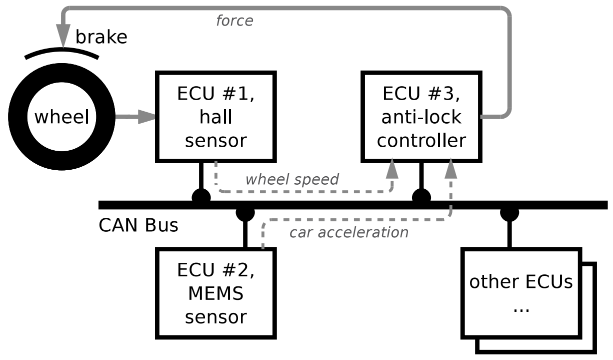


Fig. 2: Anti-lock braking system with distributed sensors and controller.

If the iterative method as discussed above fails to satisfy a target property, relaxed versions of the original target are derived in a methodical manner, and the verification flow is repeated. The amount of relaxation required by a target property in order to ensure a satisfiable verification run actually provides an important feedback to the designers, in the form of how near the present design is w.r.t. some target control performance.

III. Tool Integration

In this section, we explain the implementation of our method using an illustrative example. We consider a version of an anti-lock braking system. The system has two sensors for measuring wheel speed and linear acceleration of the car. The sensing is done on individual sensor ECUs and the sensor data is communicated over a common bus to a controller ECU, see Fig. 2. The controller ECU evaluates the sensor data, and computes an optimal setting for the brake pressure. Towards that, the controller has to estimate the current amount of tire slip based on the wheel speed and the car deceleration, and then regulate the brake pressure to keep the slip at an optimal value (usually 20%).

One of the performance requirements of this system is, that the car shall stop within a certain distance when the brakes are fully applied. The critical variable in this system is the estimated tire slip. Naturally, when the sensor data is poorly synchronized, then the estimate is less accurate, leading to suboptimal control actions. The observable result is a longer stopping distance. Therefore, in order to provide a formal guarantee that the requirement can be fulfilled, the timing behavior of the implementation has to be fully known. In the following sections we explain how our approach enables to derive the timing behavior of the distributed system, and how we formulated and verified a formal property representing the mentioned performance requirement.

A. From High-Level Specification to Task Mapping

Whilst our approach in principle works with any high-level model for which some tool can generate source code, we demonstrate it with the example of a control algorithm that – as oftentimes the case – was being modeled in MATLAB/Simulink.

The anti-lock braking system was modeled with four blocks (Fig. 3a, from top to bottom): (1) acceleration sensor, (2) controller, (3) wheel speed sensor and (4) car simulation. From those blocks, the car model is only meant for performance analysis of the controller, and not supposed to be implemented in the real system. Additionally, we did not fully model the sensor blocks, since reading sensors is usually a manually written code, which at this stage of the design had not been done, yet. Instead, the sensor blocks simply pass through the respective variables from the car simulation.

A control performance analysis in MATLAB/Simulink could be used to compute the effectiveness and stability of the (ideal) control loop. However, in this high-level model, we are yet lacking information about the timing of the implementation, and it is well-known, that jitter and latency can deteriorate the control performance. Though, if the control designer had a guarantee on the end-to-end timing in the system, then he could derive the control performance of the implementation.

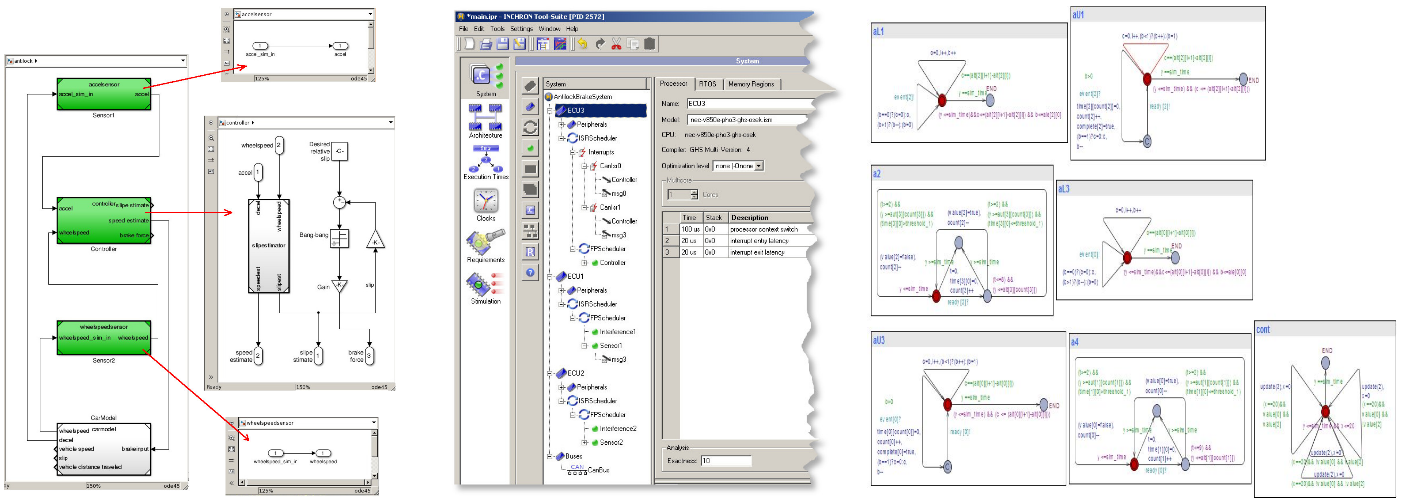
One way around this, is to let the control engineer specify timing requirements for the implementation, and make sure that the implementation follows them. Usually, this would entail specifying a relative deadline for each task, i.e., the maximum time that the task may take to complete computation. However, there are two problems: First, we are dealing with a *network* of tasks, where platform timing, such as communication delays, also have to be considered, and second, in a traditional control system, the deadline must never be violated to guarantee performance, which often results in overdimensioned hardware.

Our workflow addresses both of these problems by allowing to formulate timing requirements that are referring to the *distributed* system, which moreover permit specifying complex properties beyond simple deadline violations.

In the case of the anti-lock brake, we formulate a property that allows for occasionally "poor" sensor synchronization and still can guarantee the desired control performance, resulting in a more resource-efficient implementation than traditional designs. As mentioned before, the critical, performance-driving variable is the estimate of the tire slip, whose quality depends on the merit of sensor data synchronization. Therefore, we can translate the performance requirement into conditions for the sensor data. The data itself is timestamped, hence, depending on when the sensor data arrives at the controller, it can be classified as either *fresh* or *stale*, depending on some pre-specified tolerance threshold.

The control engineer can then quantify the required freshness in terms of an (m, k)-firm deadline [14] for the data. That is, he can require that the data is fresh at least m out of k successive runs, whereas the relation from [15] is used to translate the control performance requirement into the values of m and k.

Next, the control engineer uses Simulink's *Embedded Coder* to generate a C source code for the controller task, whilst in parallel the formal properties of the design are exported, such as the required (m, k)-firm property [14], as well as the periods of the controller and sensor tasks.



(a) Functional high-level model in Simulink. (b) Platform modeling and task mapping in INCHRON Tool Suite. (c) Timed automata in UPPAAL during the verification.

Fig. 3: The anti-lock braking system at various stages in the workflow.

Finally, the specification of the platform and the mapping of the tasks onto the platform can be done either in *Enterprise Architect*, or in *INCHRON Tool Suite*. For both cases, we developed a pre-defined set of commonly used building blocks for ECUs, buses and so on, which can be easily instantiated and tuned, similar to the blocks in Simulink. Towards this, *Enterprise Architect* was extended by a *profile*, which also allows exporting the Enterprise project to INCHRON Tool Suite. The platform definition and mapping using *INCHRON Tool Suite* are shown in Fig. 3b.

B. Timing Analysis

Task-Level Timing Analysis: Each generated source code contains a `step`-function, which has to be invoked periodically with the respective task period. Consequently, the WCET of the individual tasks are the longest possible execution times of the respective step functions.

For the sensor tasks, which are not yet fully implemented, we allocated an execution time budget by specifying some WCET values. Later during the design, their actual WCET can be obtained similar to what is shown in the following for the controller task. The WCET for our controller task was obtained as (cmp. §II-B):

1) Cross-compiling the C code with the target's *gcc*,
2) using *newtool* to analyze the resulting binary for basic blocks and determine their execution times,
3) using *newtool* to find the flow differences between binary and source code and translating the differences back to obtain an *augmented source code*,
4) using *newtool* to back-annotate the execution times into the augmented source code,
5) using *AutoGen* [3] to perform slicing and abstractions w.r.t. timing and
6) using *cbmc* [16] to compute the WCET value.

Where *newtool* is currently under development.

To include other effects, such as the communication delay between the sensor tasks and the controller task, we need to find the worst-case latency of the bus between them, as well the scheduling effects on the ECU. That *system-level timing* is obtained using INCHRON's *chronVAL* tool.

System-Level Timing Analysis: *chronVAL* uses real-time calculus (RTC), as explained before, to compute the bounds for both the execution times of all tasks *including* preemptions and the bus communication. The result of the analysis are *arrival curves* for all tasks and *service curves* for all processing and communication resources in the system. Simple timing properties, such as the worst-case reaction time (WCRT) of a task, or an upper bound for end-to-end timing of the controller tasks, can be read off the curves more or less directly. However, more complex properties, such as the (m, k)-firm property in our system that ensures a sufficient control performance of the anti-lock brake, require further analysis. This is described in the next section.

C. Automata Generation and Formal Verification

For our anti-lock braking system, the property we need to verify is the (m, k)-firm bound on the freshness of the input sensor values at the controller ECU. The freshness or staleness is calculated as the difference between the time at which the sensor data reaches (i.e., is processed at) the controller ECU and the time at which it is actually captured by the sensor. For this simple representative example, we discuss the basic working principle of our automata generation framework, including techniques for abstracting out intermediate subsystems to avoid a state space explosion problem.

Using the method as discussed in section II-C, the results of system-level timing analysis are translated into a set of timed automata models for UPPAAL [10]. Note that the property under examination is not concerned with any architectural event or any internal communication event. It does not take into account any event concerning values of sensed variables, but only refers to the timing of sensor data as processed by the controller. Hence, for the verification run, we do not have to generate an elaborate network of automata which captures the details of the sensor ECUs, the bus and the controller. Instead, our model only needs to capture the input sensor data and the input to the controller, and the rest of the subsystems can be abstracted with their timing effects being accounted for by ChronVAL. Thus, our UPPAAL model consists only of:

1) Two pairs of synchronizing timed automata to generate the two input sensor events as per RTC constraints given by the arrival curves. This generates all possible combinations of sensor arrival patterns satisfying the RTC bound.
2) Two task automata which keep track of when the two input tasks have been generated by the synchronizing automata. The automata also capture when the sensor task has arrived at the controller based on the dependencies, WCRT, BCRT and the RTC constraints given by chronVAL. We also add labels of freshness or staleness to every instance of task in both automata.
3) The controller automaton, which simulates the periodicity of controller execution without going into the details of the architecture nor functionality. The controller is modelled as a node and due to the simplicity of this model, the observer is integrated into this controller automaton. An observer is needed to simplify the queries that needs to be asked to verify for the (m, k)-firm property.

The above network of automata is modelled in UPPAAL and is shown in Fig. 3c. Our task automaton labels the task instances that are generated as either fresh or stale. The controller automaton executes periodically and it knows the freshness or staleness of both the tasks at that executing instance. To verify the (m, k)-firm property [14], the controller automaton keeps track of a sliding window of length k, and updates the value of m at every controller period. Here, m is the number of tasks that are fresh in a window of k. This is how the observer may be integrated into the controller automaton for simple (m, k)-firm properties. During verification, we query for reachability of m. For example, if $(4, 5)$ is the (m, k)-firm bound we need to verify, we query for the reachability of $!(m < 4)$ && $(m >= 4)$. If this is reachable, then the tool infers that the (m, k)-firm property is satisfied.

We have observed interesting results through this analysis. The property we are considering is that for a sliding window of k sensor tasks, at least m out of those k tasks are fresh. The freshness is decided by the time threshold defined by the designer. The sensor tasks arriving before this time threshold are fresh. The initial property derived by the control engineer is that, for a time threshold of 7 time units, at least 4 out of 5 sensor tasks should be fresh, i.e., $(4, 5)$-firm should be satisfied. Our analysis shows that $(4, 5)$-firm is not satisfied. Then we search for any other relaxed combination of m and k, where $m <= k$, which is satisfied for the same system architecture. In our case, none of the m and k combinations were satisfied. We found that, if we alter the requirement such that freshness is defined with a relaxed time threshold of 8 time units, the target property of $(4, 5)$-firm is satisfied.

To summarize, in case of verification failure with a valid counterexample, we first check whether a slightly relaxed property holds. In that case, the designer can go for slight changes in system architecture, e.g., change in sampling rate. In the present scenario no *nearby and relaxed* property could be satisfied. As it happens, in spite of trying different possible sampling rates, neither the target property, nor its relaxed versions could be satisfied. In this case, it requires a more fundamental change in the definition of freshness in the control law for satisfying the target property. The scenario as discussed actually exemplifies the usefulness of the verification feedback for the control designer.

IV. Related Work

Overall approach: High-level design tools like Simulink nowadays have an associated "Design Verifier", which can check high-level properties using back-end theorem provers, as well as formal analysis engines like Polyspace [17]. However, the properties being checked are not platform-aware, i.e., only the high-level model is considered, without any underlying architecture or executable task. Hence, the validation capabilities of such commercially available control toolboxes are limited. They can check for simple errors in the model, like possible integer overflows, division by zero, unreachable logic etc. Also, using the Simulink Design Verifier, violated assertions can be identified by simulation only.

Existing end-to-end approaches, such as the CESAR toolchain [18], either do not include a complete temporal model, or only allow for simulation as means of verification [19]. Again other toolchains, such as the combination of ASCET and AbsInt Tools [20], focus on temporal properties, but lack an integration with functional verification, e.g., cannot provide performance guarantees of the high-level model. Most existing tools, however, such as [21, 22], enforce a one-way workflow, where the high-level model is successively refined up to an implementation, but then there is no feedback to the original high-level model. Most importantly, the temporal behavior of such an implementation is just an end product, with no handle for the designer. Consequently, our approach stands out for its integrated coverage of both functional and temporal verification goals, providing an industrial toolchain that supports the entire development workflow for embedded real-time control systems.

A novel approach proposed in [23, 24] integrates timing models into functional analysis using hybrid automata, a superset of timed automata, for formal verification. However, the WCRT considered is far smaller than the actual WCRT and it is assumed that when a task violates this smaller WCRT, its execution is discarded. Also, in this approach, the entire system needs to be modelled as a hybrid automaton for verification, and this leads to a state space explosion for complex models. There is also no feedback to the original

high-level model. Moreover, their approach does not take into consideration bus protocols and associated latencies for timing analysis. Our approach does not have any restrictions on the WCRT values and can give formal guarantees over all possible execution traces. Also, our method allows for abstraction of the system architecture during verification. This inherently addresses the problem of state space explosion, even for complex models. Model refinement is shown in [24], using a simulation approach. Being a simulation-based approach, it cannot give any formal guarantees. Our approach both allows for model refinement and can also provide formal guarantees.
WCET analysis: An estimation of WCET at source level using cbmc has been proposed in [5]. However, the authors did not model the inevitable flow differences between binary and source, which is why their approach has to over-approximate the WCET. In contrast, our approach is not only more precise, but also more scalable, due to the slicing and abstraction w.r.t. timing.
Real-Time Calculus: The concept of RTC was firstly proposed by Thiele [7], and was later extended by approximative methods, as described in [8] and [9]. Those methods were implemented and further developed in the chronVAL tool. The problem of generating events from RTC curves has been studied by Lampka [11] before. We refined that approach in order to include any arbitrary set of arrival curves, both periodic and aperiodic.

V. Conclusion and Future Work

The presented work introduces a tool flow for design and verification of embedded control solutions, taking into account the control-theoretic, software-centric, as well as platform-level design factors in a sound and integrated manner. In general, the problem of co-design for real-time control requires analysis and optimization skills from multiple disciplines, like control design, real-time analysis, as well as formal verification. The tool flows available in each of these realms are typically lacking interface points between each other, and thus are inhibiting a comprehensible and effective feedback between the disciplines. The result is a non-holistic system design that often only attains a suboptimal level of performance and efficiency.

With the presented workflow, it becomes possible for a control designer to specify the control law, for an embedded systems engineer to come up with an implementation mapping, for a verification engineer to check for some target control-theoretic property, without any of them requiring to work outside their own area of expertise. With our effort, we automated the transformation of the design problem across the respective domain boundaries, enabling an integrated approach to a system design with optimized performance and efficiency.

In the present status of the verification engine, the generation of spurious counterexamples is not prevented in subsequent runs by modifying the RTC constraints automatically. Such refinements in the CEGAR loop at the moment require manual intervention. Automating this segment of the tool flow is a part of future work.

References

[1] Mathworks, "Simulink," 2015, http://mathworks.com/products/simulink/, Online.

[2] INCHRON, "Tool-Suite," 2015, http://www.inchron.com/tool-suite/tool-suite.html, Online.

[3] P. Bokil, P. Darke, U. Shrotri, and R. Venkatesh, "Automatic test data generation for c programs," in *Proceedings of the Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009, pp. 359–368.

[4] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[5] S. Kim, H. Patel, and S. Edwards, "Using a Model Checker to Determine Worst-Case Execution Time," *Computer Science Technical Report CUCS-038-09, Columbia University*, 2009.

[6] P. Darke, B. Chimdyalwar, R. Venkatesh, U. Shrotri, and R. Metta, "Over-approximating loops to prove properties using bounded model checking," in *DATE*, 2015.

[7] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," *IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century.*, vol. 4, pp. 101–104, 2000.

[8] K. Albers and F. Slomka, "An event stream driven approximation for the analysis of real-time systems," in *Proceedings of the EUROMICRO Conference on Real-Time Systems*, 2004, pp. 187–195.

[9] K. Albers, "Approximative real-time analysis," Ph.D. dissertation, University of Ulm, 2011.

[10] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *Third IEEE International Conference on Quantitative Evaluation of Systems*, 2006.

[11] K. Lampka, S. Perathoner, and L. Thiele, "Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems," in *ACM international conference on Embedded software*, 2009, pp. 107–116.

[12] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikučionis, U. Nyman, and A. Skou, "Hierarchical scheduling framework based on compositional analysis using uppaal," in *Proceedings of Formal Aspects of Component Software*. Lecture Notes in Computer Science. Springer, 2014, pp. 61–78.

[13] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald, "Abstraction and counterexample-guided refinement in model checking of hybrid systems," *in International Journal of Foundations of Computer Science*, vol. 14, no. 04, pp. 583–604, 2003.

[14] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1443–1451, 1995.

[15] M. Kauer, S. Steinhorst, D. Goswami, R. Schneider, M. Lukasiewycz, and S. Chakraborty, "Formal verification of distributed controllers using time-stamped event count automata," *Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 411–416, 2013.

[16] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 2988, pp. 168–176.

[17] P. Munier, "Polyspace®," *Industrial Use of Formal Methods: Formal Verification*, pp. 123–153, 2012.

[18] E. Armengaud, M. Biehl, Q. Bourrouilh, M. Breunig, S. Farfeleder, C. Hein, M. Oertel, A. Wallner, and M. Zoier, "Integrated tool–chain for improving traceability during the development of automotive systems," in *Proceedings of the Embedded Real Time Software and Systems Conference*, 2012.

[19] M. Karner, M. Krammer, and A. Fuchs, "System level modeling, simulation and verification workflow for safety-critical automotive embedded systems," in *Proceedings of the SAE World Congress & Exhibition*, April 2014.

[20] K. Richter, C. Ferdinand, and R. Heckmann, "Timing correctness and model-based software development for safety-critical automotive applications – an integrated, tool-supported workflow," *Automotive–Safety & Security*, 2008.

[21] R. Mader, G. Griessnig, E. Armengaud, A. Leitner, C. Kreiner, Q. Bourrouilh, C. Steger, and R. Weiss, "A bridge from system to software development for safety-critical automotive embedded systems," in *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Sept 2012.

[22] W. Herzer, R. Schlick, M. Schlager, B. Leiner, B. Huber, A. Balogh, G. Csertan, A. LeGuennec, T. LeSergent, N. Suri, and S. Islam, "Model-based development of distributed embedded real-time systems with the decos tool-chain," SAE Technical Paper, Tech. Rep., 2007.

[23] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, "Formal analysis of timing effects on closed-loop properties of control software," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[24] D. Ziegenbein and A. Hamann, "Timing-aware control software design for automotive systems," in *DAC*, 2015.