1. Introduction

Evolving Computer Systems have become common over the past years. It is an accepted procedure that updates for desktop operating systems or office applications are regularly released and must be installed in order to remove flaws of the originally installed software. Similar developments can be observed for most software intensive systems. One has grown to expect that firmware updates should be made available for all kinds of electronic equipment including TV sets, routers, your favorite gaming console or your mobile phone.

By installing updates or additional software on these devices, they evolve over time. Functionality is added or removed, bugs in the original software are fixed. In non-critical environments, such as a desktop computer or a mobile phone, these updates may already be applied automatically without the necessity of interaction with the user as soon as they are released by the manufacturer.

Using a similar process, some non-critical systems already evolve over time to adapt to changing consumer or environmental demands. Currently, this process is driven by the developers or manufactures of these systems. In the future, it is to be expected that this process will be increasingly driven by customer demand, as devices are expected to adapt to the specific user. In order to satisfy this demand, systems will be extended with logic enabling them to autonomously decide on software updates and additions to adapt to customer demands. On a more global scale, devices could also be allowed to autonomously cooperate with each other to provide novel services to the user. For example, a mobile phone could automatically (as opposed to manually, by e.g. Bluetooth^{TM}(IEEE 802.15.1)) integrate with e.g. an automotive infotainment system to provide telephony or media services as long as it is in the car.

This trend does not only apply to general purpose computing or multimedia applications, but extends also into safety-critical domains such as automotive systems. It is already commonplace that upper-class vehicles get software-updates or replacement control units (also containing updated software) in the shop if a software issue arises at the customer. Allowing such a process, while still guaranteeing safe operation of the vehicle poses a tremendous challenge on the design and verification of such updates as will be further motivated in the next section.

Today, these updates are released by professional developers after extensive development and testing in the lab and installed by trained engineers in the shop. This process allows for stringent control of updates in the field by the system manufacturer. The complexity of this process however, limits its applicability to cases where it is absolutely necessary. This means, that only major or critical bugs in existing software will be fixed, additional functionality is only be released together with a new product.

Driven by customer experience from non-critical electronic systems, a demand for software evolvability, i.e. the ability to also change system functionality, will arise also for critical, software intensive systems, such as automobiles. This trend is supported by design-opportunities arising from the increasingly structured design approach of such systems. This shift towards added flexibility is already adapted by Audi [103], who plan to deploy an "App-Store" for future cars, enabling the owner to install additional software, such as navigation solutions, on demand. In the future, such a scheme could also extend into safety-critical domains, such as driver-assistance-systems. Ensuring safe operation under such circumstances is an open challenge in the design of future evolving safety-critical systems.

As mentioned before, the design of embedded systems software is becoming increasingly structured. This includes a trend towards layered software architectures and standard interfaces, enabling independent development of hardware units, operating system and applications as is well-known from general purpose computing. In the automotive domain, the AUTOSAR standardization effort [1] aims at developing an industry standard for interfaces in automotive software.

Enabling independent development of applications also enables their development by third parties. This is already common practice today. Many applications integrated into an automotive systems are supplied by so-called tier-one suppliers to be integrated by the OEM. A Bosch CEO [54] has recently noted that an increasing amount of third party software will run on future automobile systems. He continues to claim that these may not adhere to the quality demands of today's automotive systems.

Currently, the OEM is responsible for ensuring a sufficient quality of software bought from suppliers. In a more open system, where e.g. the user may install third-party software, this is no longer the case. The user cannot, however, take over the part of the OEM, controlling and ensuring integrity and correct functionality of the compound system of original software, updates, newly installed applications supplied by the OEM and third party software. If not solved, this fact may become a show-stopper in the development of more flexible, also safety-critical systems in the future, which will be demanded by the customers.

To overcome this issue, this thesis proposes to establish a trusted update and installation process, that automatically ensures correct functionality of the compound system. Such a process could diminish the need for central validation of each configuration at the OEM. If to be employed in a practical setting, such a process needs to address all relevant properties of embedded software, ranging from functional correctness, over security aspects to issues arising from applications sharing a common platform ("integration issues").

Clearly not all of these issues can be addressed in one thesis. This thesis focuses on integration issues, which are a key challenge when implementing flexible embedded systems, as will be further motivated in the next section. This thesis proposes a general framework and process that ensures correct functionality of all system config-

urations that a flexible embedded system may experience during its life time. The key component of this framework is a formal analysis of the multiple effects arising from integration. Here, this thesis addresses the analysis of timing issues potentially arising from integration as an important example of the large set of potential integration issues. For completeness, this thesis also presents a runtime system that can provide the flexibility needed by future embedded systems, while pertaining the performance that we know today.

1.1. Development Process and Related Issues

After setting the stage for the general challenge to be addressed, take a closer look at the development of embedded software for complex devices such as automobiles to learn more about the specific challenges to be addressed.

Today, complex embedded systems are developed in a distributed process. Again, the automotive domain serves as an example. In a modern car, hundreds of different functions, ranging from anti-lock brakes over driver assistance systems to infotainment solutions are implemented to a large extend in software [26]. The OEM only develops a fraction of these functions in-house. Most of the development and implementation is outsourced to multiple tier-1 suppliers. These in turn may sell their functions to multiple OEMs.

The development processes of the functions at the different tier-1 suppliers are only loosely coupled via requirements specified by the OEM. Note that no specific effort is undertaken by different tier-1 suppliers to ensure interoperability of their solutions, if not specified as a requirement. In this development process, ensuring interoperability is the task of the OEM. In some cases, tier-1 suppliers might delegate development of parts of a functionality (e.g. base operating system) to tier-2 suppliers further complicating this process [25].

Note that in order to save costs, these functions are integrated onto a common platform for a given automobile. Today, this is especially true for communication. In order to save weight and cost of the wiring harness, only few shared bus systems are physically implemented. Clearly a goal is to minimize the amount of cabling. In the future, this may also apply for computational resources, i.e. an increasing amount of functions may be integrated onto single electronic control units (ECUs).

Now consider the task of ensuring or even formally proving correctness of the complete system, i.e. the car, as is required by existing certification standards such as IEC 61508 or the upcoming standard ISO 26262. One aspect that clearly has to be considered is functional correctness, i.e. do all functions adhere to their requirements. This is the domain of traditional software engineering and can be covered by the design teams located at the tier-1 and tier-2 suppliers. Considering this aspect, the functions to be developed are mostly independent of each other, easing verification.

There are, however also properties that require to consider the complete system in order to be verified. An example is system timing. Many control functions have

timing constraints in order to work properly. It is for example mandatory that breaks engage instantaneously after pressing the brake pedal. From a technical point of view instantaneously translates to a couple of milliseconds, a delay not noticed by the driver. Adherence to such constraints can only be shown in conjunction with all other functions sharing infrastructure (i.e. bus systems) with the specific function in question (i.e. the brake system). This is due to the fact that complex interdependencies between functions sharing e.g. communication infrastructure may arise due to the employed arbitration scheme. It is to be observed that verification of adherence to this class of constraints can in most cases not be performed during development of the specific function, as not only the behavior of a single function has to be considered, but all potential combinations of function behavior to be integrated into the final system.

With this in mind, return to the challenges outlined beforehand. How does an increasing demand for software flexibility affect this process? Irrespective of whether a change in the system originates in the update of an existing or the addition of a new function, the following can be observed. Clearly, all aspects as discussed above have to be considered also for the system resulting from the update.

Current practice in non-critical systems shows that retesting functions in isolation is feasible and yields good results also with short product life-cycles. Also, in some cases the functions developed for the next generation of cars could run on the computational platform already deployed in current cars. Selling these functions as extensions to existing automotive systems may constitute a valuable business case.

Integration testing, however, cannot follow these time-lines due to its complexity. Here, it must be ensured that an update is compatible with the complete configuration landscape in the field, which may not be completely known. This makes this verification process extremely costly. As such, currently updates in automotive systems are only deployed if inevitable, e.g. in case bugs in the original software need to be fixed. New functionality, that has not been available with the release of the specific version of the vehicle is deployed with the following generation, which gives a greater revenue. This topic has also been discussed in [68]. Here the focus lies on security and privacy issues.

Thus, the challenge to be addressed if a more flexible approach to software configurations of complex safety-critical systems is to be realized, is to simplify the task of *integration*. Hence, a practical solution to show adherence to all non-functional constraints of a given system in case an update is applied must be found. In this thesis, the aspect *system timing*, i.e. adherence to all timing constraints is considered as one non-functional aspect of complex safety-critical systems.

The next section outlines the current practice in timing verification of (automotive) real-time systems to give further insight on the state of the art and specific challenges to be addressed.

1.2. Current Practice in Timing Verification

As discussed above, the ultimate goal of timing verification of hard real-time systems e.g. in the automotive domain, is to prove (or at least to convince the certification authorities), that the system complies with all timing constraints irrespective of the operational state of the vehicle.

Current research partitions this challenge into two problems. First, the individual maximum computational demand of all software components must be derived, together with their communication needs. Secondly, the system timing must be considered. Here the focus lies on bounding arbitration effects, i.e deriving a bound on the time that other (e.g. higher priority) tasks might interfere with the application under consideration. In the industrial case, both challenges are to a large degree addressed by extensive simulation. This requires a lot of effort to be put into the generation of test cases and in general lacks coverage (i.e. test cases cannot cover all operational states).

Over the last years, research has provided a broad body of work addressing both challenges with formal methods. Such methods aim at deriving bounds on computational demand and interference by overestimating actual behavior. The next section give a short introduction to this work.

1.2.1. Bounding Computational Demand

First consider formal determination of the computational demand of a piece of software. This is a tedious task due to multiple reasons. First, the control flow of the software components may be non-trivial (e.g. contain potentially unbounded loops), making it hard to decide on the worst case control flow. Secondly, advanced processor architectures are hard to predict, i.e. depending on the state of the processor (e.g. cache, pipeline), the same command may take a different amount of time to complete, again complicating the task of finding the worst-case computational demand of a software component. This is an active area of research. A good overview on the current state of the art can be found in [137], where different approaches are outlined.

Where formal methods cannot be applied due to their complexity, or are not required by existing safety standards, computational demand is oftentimes derived by simulation and benchmarking. It is the opinion of the author, that since the developers usually have good knowledge of the application, such benchmarking schemes can reach high accuracy. Clearly, benchmarking can not replace formal analysis, where proven behavior is required. However, in many other cases, it can serve as a sufficiently accurate estimation.

1.2.2. Deriving System Behavior

The second challenge is to derive properties of system timing. Again, first consider formal approaches. Here, the computational demand of software components serves as input data. A system model is built from application descriptions consisting of

task graphs describing functional dependencies of software components, which are annotated with computational and communication needs. System timing properties are derived by considering the employed arbitration policies, where software components or communication shares processors or communication media. A survey and comparison of different approaches to this problem can be found in [95]. For single-core processors and buses, these approaches provide accurate results at reasonable computational cost. Extensions for multi-core processors exist, but added interdependencies due to shared resources such as memory complicates analysis [111].

Again, simulation and benchmarking pose an alternative approach to show feasibility of system timing. This is, however, by far more complex than for software components. This is for several reasons. First, the state space of the compound system grows exponentially with the number of applications, making it extremely hard to find suitable test patterns. This is complicated by the fact that the integrator in many cases does not have in-depth knowledge of the single applications internal behavior. As such, it is the author's opinion that statements on system timing based on benchmarking are not as trustworthy as statements on component timing, due to the size of the state space and lack of knowledge.

As a further consequence, it is to be expected that formal methods for system analysis will take an increasing role in the verification process, as the complex interdependencies arising from hundreds of applications sharing an infrastructure cannot be overlooked by an engineer. This is in contrast to the analysis of computational demand of components. Here, formal methods face tremendous challenges, where on the other hand the designers of the software are available for benchmarking.

1.2.3. System Timing Verification Process

To summarize, it is to be observed that in case an (in parts) formal approach to the verification of system timing properties is to be taken, this process will consist of two steps. First, descriptions (or *models*) of the single software components (or applications) in isolation must be derived. In a second step, these are aggregated for verification of system properties.

In the automotive industry, the second step is further complicated by the amount of variants sold of a single product line. Feil [41] pointed out that in modern upper-class vehicles of the year 2005 at least 25 options to the car configuration exist, that influence the Electric/Electronic (E/E)-infrastructure of the car. This means that $2^{25} = 33,554,432$ potential variants of this specific model exist. It is to be expected that this number has grown over time.

Each of these configurations must be tested for compliance with the requirements. This is a challenging and costly task required for each change in software. It is the author's impression that this cost is one of the main reasons prohibiting updates in today's automotive systems on a large scale. The cost of this process is only accepted where bugs in the original software need to be fixed.

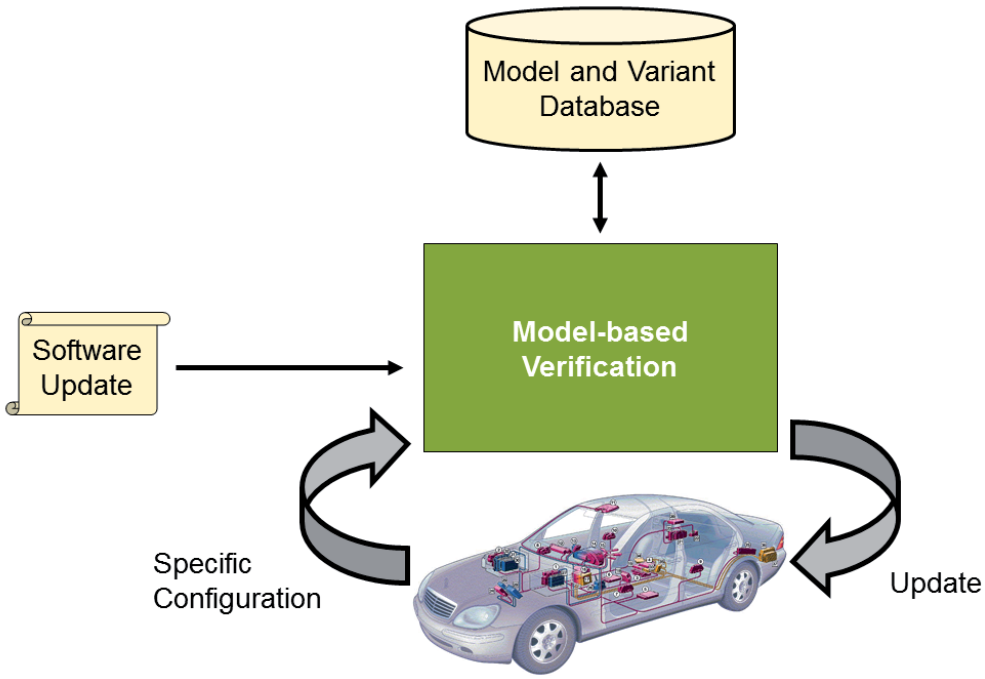

Figure 1.1.: Novel Integration Concept

1.3. Research Objective

This thesis discusses methods to enable software flexibility also in critical systems. Here, it specifically addresses the challenges arising due to integration testing and verification. Its objective is to propose a method reducing the overall cost of integration testing, thus enabling software evolution as known from other embedded devices, such as TV sets or routers, also in safety critical systems. The main use-case addressed will be addition or removal of applications by means of a software update. Clearly, this subsumes changing applications. As motivated before, this thesis will focus on system timing as one of many integration aspects.

The approach presented in this thesis is motivated by two observations on the current state of the art in the design and verification of complex embedded real-time systems. First, a general trend towards adoption of formal methods for the verification of system properties also by the industry can be observed (e.g. [63, 136, 64]). This is driven by the increasing complexity of the systems to be verified, rendering excessive benchmarking useless due to poor coverage.

The second observation goes beyond industrial practice and considers properties of these formal methods. As discussed before, verification of system timing properties is a two-stage process. In compositional or modular approaches to system analysis, construction of the system model follows stringent rules once descriptions of applications in isolation are available. Thus, this step lends itself for automation

Increasing acceptance of formal methods for system verification in industry paired with the observation that system models can be automatically aggregated from local models of applications motivates design of the update flow presented in this thesis.

The general idea is depicted in figure 1.1. For aspects, where formal verification methodologies exist (e.g. timing), integration testing in the lab is reduced to a minimum. Instead, partial models describing the additional application to be installed are packaged with the update and used to perform a system analysis at installation time.

Here, the specific configuration of the product to be updated is considered by extracting a model of its current configuration. This model arises from automated aggregation of the partial models of all previously installed applications. The model of the update is then integrated into this model to perform an analysis of the system that would result from the update.

The update is installed only if analysis of the resulting compound model proves correctness of the resulting system. Although not explicitly necessary in this flow, a Model and Variant Database may still be used to steer analysis and gather results for e.g. stochastic analysis. Irrespective of the trend in the automotive industry, establishment of such a process has repeatedly been identified as a key research challenge to be addressed in the context of future evolving or self-adaptive systems [31, 37].

Such an admission control process has numerous advantages. First, as admission control is performed based on data associated with the specific product to be updated, this approach scales seamlessly with a large number of variants. Secondly, if automated, this process can be used to allow for a user-driven scheme of installing software in a given car, such as an “app-store”. Once the computational resources are fully allocated to existing software, no further applications can be installed without the removal of others. Overall, the process enables a more flexible way of managing the software configurations of also critical systems, which could provide multiple interesting business opportunities.

On the other hand, the establishment of such a methodology bears many research challenges. In order to be widely applicable and thus useful, it must address a broad range of aspects arising during integration so that lab-based testing is reduced to a tolerable amount or – in the best case – becomes unnecessary. It is unclear, for which aspects an automated admission control scheme as outlined above can be realized and what the associated cost is.

As a second aspect, a flexible runtime environment capable of managing such a scheme must be developed. It must be able to accommodate analysis routines required for the admission control scheme and provide a flexible execution platform for the applications. This extends beyond current practice in runtime environments for embedded systems (such as AUTOSAR), which are mostly statically configured at compile time. In order to find acceptance, the mechanisms implemented in such a runtime environment must meet the highest standards (of e.g. availability, reliability, security), so that they can be trusted to always yield correct results.

Isolation between applications is another, equally important aspect. Proving that the system works given that all applications behave as is described in their associated models is one aspect. Ensuring that this is also the case if one application does not adhere to its description is a completely different challenge. Here, suitable isolation facilities must be put into place. These again must match all aspects covered by the visioned admission control scheme and adhere to their highest standards in order to be trustworthy.

Clearly, in the scope of this thesis, not all of these aspects can be considered. The next section gives an overview of the research challenges addressed herein.

1.4. Contributions

As has been stated throughout the introduction, this thesis is concerned with system timing as one integration aspect. A broad base of theoretical work on formal system timing analysis exists and its adoption in industrial practice has already started, making it a worthwhile topic. The timing aspect provides a research vehicle to touch all aspects of the update process outlined above. The contributions to the state of the art in this area are threefold.

First, a conceptual view on the problem is taken, yielding a suitable high-level protocol and software architecture supporting safe updates of real-time systems. The software architecture and overall approach is designed to also apply to other integration aspects, where formal analysis mechanisms exist. A discussion of implementation options available together with a description of the approach taken within this thesis defines the requirements for the algorithms further presented in this thesis.

One key contribution of this thesis concerns the analysis algorithms to be employed for automated analysis. Here, three aspects are touched. First a novel formalism of compositional performance analysis is presented enabling reasoning about equivalence and optimality of algorithms solving the analysis problem. Secondly, a distributed analysis algorithm is proposed and its equivalence to existing algorithm is proved using the aforementioned formalism.

As a second aspect, the processing demand of the analysis algorithms themselves is considered. Two aspects need to be considered in order to derive a bound on the runtime. First, a bound on the number of steps needed by the global iterative algorithm is presented. In order to derive a runtime bound from the number of steps, the computational demand of each step must be bounded. This constitutes the second aspect. Each computational step of the global iterative algorithm comprises of the computation of worst-case response times. Similar to the global algorithm, these are also computed in an iterative manner. There exist, however, bounded time approximations. An empirical study on the trade-off between computational demand of existing algorithms and their accuracy completes the discussion of the computational demand of the analysis itself.

As a last aspect, this thesis touches design aspects of an execution environment, which is sufficiently flexible to handle the demands of evolving real-time systems,

while still allowing a low-overhead implementation. An implementation of this environment yielded a system setup demonstrating the practical applicability of the proposed approach to a generic control-theoretic example.

1.5. Outline

This thesis is structured as follows. The following chapter is concerned with the presentation of the overall approach to the implementation of an update mechanism as outlined before. This discussion yields a high-level software architecture together with an update protocol to follow.

Chapter 3 describes the analysis algorithms developed in the course of this thesis. It first discusses potential base technologies to motivate the use of Compositional Performance Analysis [106, 62, 57] as a base technology. In a first step, it defines a novel formalism describing the analysis approach in order to be able to reason about specific algorithms implementing it. In a second step, it presents a distributed algorithm which integrates into the overall concept described in chapter 2, to then prove its optimality using the introduced formalism.

Chapter 4 is dedicated to the derivation of a runtime bound of the distributed algorithm. This entails a formal consideration using the formalism introduced in the previous chapter, enabling reasoning about a class of algorithms rather than only the distributed one presented herein. As another aspect, this chapter also contains the empirical study of existing bounded-time schedulability analysis algorithm, which lay the foundation of the algorithms used for system analysis in Compositional Performance Analysis.

After considering these theoretical aspects, chapter 5 highlights key design choices and properties of the runtime environment implemented to show the volatility of the approach. Here, special attention is given to dynamic task management and runtime communication synthesis, as in current systems, this is usually static.

Chapter 6 closes this thesis by summarizing the basic findings and major lessons learned.