

Tackling the Complexity of Timing-Relevant Deployment Decisions in Multicore-Based Embedded Automotive Software Systems

Wolfgang Schwitzer
Technische Universität München

Rolf Schneider
AUDI AG

Dominik Reinhardt
BMW AG

Georg Hofstetter
Elektronische Fahrwerksysteme GmbH

ABSTRACT

Multicore-based ECUs are increasingly used in embedded automotive software systems to allow more demanding automotive applications at moderate cost and energy consumption. Using a high number of parallel processors together with a high number of executed software components results in a practically unmanageable number of deployment alternatives to choose from. However correct deployment is one important step for reaching timing goals and acceptable latency, both also a must to reach safety goals of safety-relevant automotive applications. In this paper we focus at reducing the complexity of deployment decisions during the phases of allocation and scheduling. We tackle this complexity of deployment decisions by a mixed constructive and analytic approach. On the constructive side, we model a multicore-based embedded automotive software system as a tuple $S = (L, R)$ with a given architecture L of the system's software component network and a given architecture R of the system's platform component network. On the analytic side, we derive models of the system's deployment as a tuple $D = (M, Z)$, where the relation $M: L \rightarrow R$ describes the allocation mapping from software components to platform components and the relation $Z: L \rightarrow \mathbb{N}_0$ describes the scheduled start times of software components. The architectures L and R allow to describe hierarchically nested software- and platform-component networks. Hence, we are able to represent coarse-grain structures like domain functions and domain networks as well as fine-grain structures like individual driver-calls, algorithms, ECUs and CPU-cores as well. With all this architectural information contained in precise models, deployment alternatives (allocations and schedules) can be automatically derived and analyzed for conformance with timing requirements, given by a set of timing constraints C . To demonstrate the usability of the approach, we implemented an architecture description language in the academic open source tool CADMOS together with a prototypical allocation and scheduling support. We are evaluating the practicability of the approach within a case study in the German national joint project ARAMiS using a close-to-production multicore-based system in cooperation with Technische Universität München, AUDI AG and BMW AG.

CITATION: Schwitzer, W., Schneider, R., Reinhardt, D. and Hofstetter, G., "Tackling the Complexity of Timing-Relevant Deployment Decisions in Multicore-Based Embedded Automotive Software Systems," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 6(2):2013, doi:10.4271/2013-01-1224.

INTRODUCTION

Multicore-based platforms are increasingly used in embedded automotive software systems. Emerging parallel

platforms such as multicore- and manycore-CPU's promise to realize very demanding future applications such as

autonomous driving and car-to-x communication at moderate cost of units and with acceptable energy consumption [1].

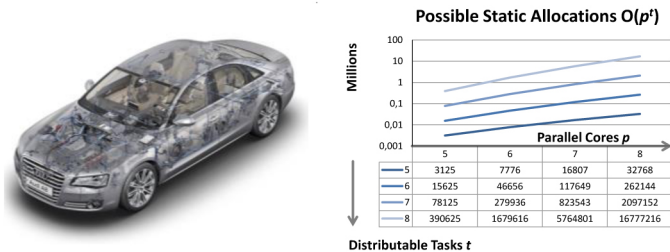


Figure 1. Growing complexity of deployment decisions for modern automotive software systems.

In order to utilize these parallel platforms efficiently, vehicle software architectures must provide an increasing number of distributable and parallelizable software components. This simultaneous growth of a system's parallel platform resources and software components poses a significant challenge for development and deployment. As illustrated in Figure 1, the number of possible static allocations to choose from grows proportionally to $O(p^t)$, where p is the number of parallel processing cores and t is the number of distributable software tasks. While for 5 cores and 5 tasks only 3,125 possibilities exist, an impressive number of more than 16.7 Million possibilities exist for 8 cores and 8 tasks. In real world systems, hundreds of cores and tasks lead to an almost unmanageable number of allocations to choose from. Nevertheless, engineers have to select one of these possibilities for a legitimate reason.

Surely there is no need to discuss that finding a valid deployment out of all this possibilities can barely be done efficiently without mathematical methods implemented on some kind of tool running on a computer, even less if the target is to find a fairly optimized configuration. In addition, there are deployment decisions, which will certainly have to be reconsidered during system development, while more and more code is implemented. This is resulting in growing functions combined with changing requirements added after the point of time when the initial decision was made. So the effort is tremendously multiplied and these complexity considerations are not only existing in theory, but have a huge impact on practical development as well.

One major problem regarding this is the fact that it is hard to estimate the *impact* on the overall system *timing*, even for a minor change of software components in a complex system and software architecture. The result of a change may still be a well performing system, an ineffective waste of resources with bad response time or, in the worst case, even a totally invalid system configuration.

To decide for an appropriate deployment, consisting of an allocation of software components and their schedule, it is required to take into account a lot of factors like timing constraints, resource requirements, distribution of given resources, speed of communication connections, and so on.

Today, the lack of appropriate tools is making this a hard job even with much less complex systems, based on single core microcontrollers and currently available solutions could more or less appear like a try-and-error approach. Common tools available for productive use, typically only allow the verification of a system timing, a deployment and/or a resource consumption, given as "candidate" input by the responsible engineers. The output of these tools is often a simple answer like "OK" or "NOT OK". At best, a vague idea of efficiency of the given candidate solution is returned. In order to improve insufficient results, iterations of this process are unavoidable, consuming precious development time.

Considering the challenges mentioned at the beginning of this section, it is high time for an approach delivering valid (and in the future maybe optimized) suggestions for possible deployments that fulfill a set of given timing constraints instead of only returning a simple OK/NOT OK judgment.

Outline. This paper presents a mixed constructive and analytic approach to tackle the complexity of timing-relevant deployment decisions for automotive software systems. In this approach, we use formal concepts such as models, logical constraints and data-flow analysis and show how these can be supported by tools. First, we introduce the description of a system by separate models for software and platform. Second, we formalize timing-relevant constraints. In the following, the relation between our abstract system models and the relevant industry standard AUTOSAR is outlined. In the next section, an implementation of the concepts within the academic open source tool CADMOS is shown. Finally, we outline some ongoing and possible future work and conclude with some remarks.

MODELING AND SYSTEM ABSTRACTION

In the context of this paper, we use dedicated models of the software- and the platform-component network of a system. These models are comparatively abstract descriptions of the software and of the runtime platform of an embedded automotive software system. We use these models to catch relevant details for automated analysis of timing relevant aspects of the embedded software system.

In general, a model serves a *purpose*, has a relation to an *original* and *abstracts* certain properties of this original (see [2]). The so-called *separation of concerns* is a common approach in engineering disciplines to divide a complex engineering task into a number of smaller, more specialized and more comprehensible subtasks. According to [3], separation of concerns typically focuses on a smaller number of relevant details inside a given concern while aspects of the respective other concerns are deliberately ignored. Among the advantages of separation of concerns are

- *reduced complexity* of individual engineering subtasks by focusing on certain details only;

- *better work division* during the development process, so different teams of experts can work simultaneously on distinct aspects of a system;
- *increased reusability* of (parts of) models by lower coupling between concerns and higher cohesion inside concerns (compare to [4]);
- *more flexible re-deployment* in response to software or platform changes supported by extensive tool automation, e.g., test-case generation, partial code and configuration file generation (see [5]) and allocation and scheduling support, as presented in this paper.

Additionally, we separate *timing-relevant deployment decisions* from the models of the software and the platform, hence, these decisions can be explicitly documented. Finally, this permits *traceable* and *precise* documentation of timing-relevant deployment decisions along the development process.

Figure 2 sketches the idea of separation of concerns by dedicated models, each focusing on certain aspects of a system only. In this paper, we separate the software components from the platform components and describe their relation explicitly by allocation mappings. These allocation mappings can be derived automatically if desired, and the impact of re-allocation on the schedule can be analyzed.

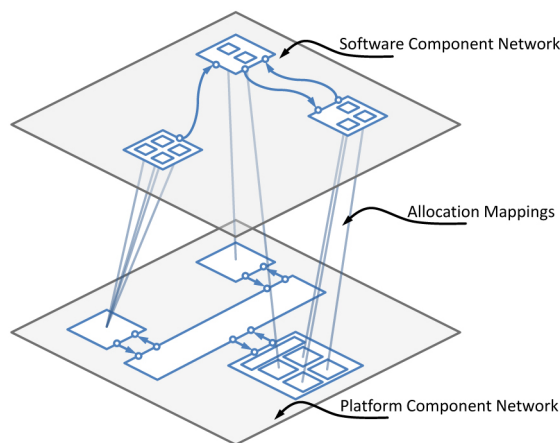


Figure 2. Separating an automotive software system into models of a software- and a platform-component network related by allocation mappings.

Basic System Model

Before we introduce the modeling of dedicated software- and platform-component architectures of automotive software systems, we define a *basic system model* that involves the three basic model elements *components*, *ports* and *channels*. Accompanied by a set of composition rules, this basic system model is then applied to describe the software components as well as the platform components, using the same basic concepts. In this paper we substantially adopt the ideas for formal system architectures as presented in [6].

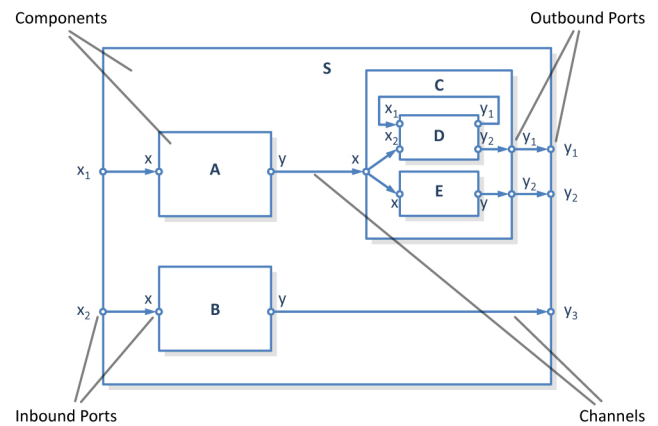


Figure 3. Modeling the architecture of a system *S* with the basic model elements: components, ports and channels.

Components. A component represents a subsystem with a defined syntactic *interface* and has an associated *behavior* with respect to that interface. A component can be a *parent* component, composed of several *children* components, where the behavior of the parent is implied by the composition of its children. For example, component *S* in Figure 3 has the children components *A*, *B* and *C* and component *C* again has children *D* and *E*. In the following, a component that is parent of a network of children components is called a *composite* component. This recursive parent-children definition of a component allows to describe nested hierarchies of component networks. Hence, a complete system *S* can be modeled as the *root* component of a *tree* of hierarchically nested components, as shown in Figure 4, for example. Interpreting a network of components as a component again is also called the *Kahn-Principle* [7, 8]. In the following document, we use the terms *system*, *subsystem* and *component* synonymously.

Ports. A port represents an interface point of a component and a port always belongs to *one* component. If port *p* belongs to component *C*, we write *C.p*. The set of all ports belonging to a component is called the *syntactic interface* of that component. A port can either be *inbound* or *outbound*. Inbound ports route messages to the inside of a component while outbound ports route messages to the outside. For example, in Figure 3 the ports *S.x1* and *A.x* are inbound, while the ports *S.y1* and *A.y* are outbound. A port can optionally be *typed*, which restricts such a port to accept messages of the desired type only.

Channels. A channel describes *directed communication* between two components. A channel connects from a source port to a destination port and is a model for the transmission of *messages* from source to destination. For example, the system in Figure 3 has 11 channels including the channels (*A.y*, *C.x*) and (*B.y*, *S.y3*). A history of messages transmitted by a channel over time is also called a *stream*. More formally, a component can be seen as a stream-processing function (see

[9] for an overview) that maps messages from incoming to outgoing channels.

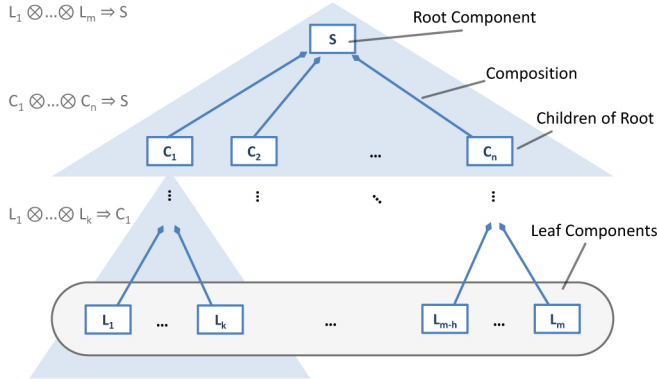


Figure 4. A system S interpreted as the root of a tree of nested components with leaf components $L_1, \dots, L_m$.

Parallel Composition. In this paper, we are dealing with parallel platforms such as networks of multicore processors and we are dealing with parallelizable software intended to be distributed on these platforms. Complex systems typically comprise numerous components connected by channels in *sequence*, in *parallel* and *recursively*. Hence, we define an operator for general *parallel composition* of components, denoted by the symbol $\otimes$ (see [10, 6] for details). This operator allows to connect components in sequence, in parallel and recursively by channels. For example, in Figure 3 the components A and B are composed in parallel, A and C are composed in sequence, and D is (also) composed recursively with itself by the channel (C, x_1, C, y_1) . The behavior of a single parent component P is implied by the composed behavior of its children components $C_1, \dots, C_n$, formally expressed by

$$C_1 \otimes \dots \otimes C_n \Rightarrow P.$$

We extend this idea from a single component and its children to a whole tree of nested components, utilizing the transitivity of logical implication. In a tree of nested components, a *leaf component* does not contain further children components. Consequently, the behavior of a system S with leaf components $L_1, \dots, L_m$ is ultimately implied by the behavior of its leaf components, formally expressed by

$$L_1 \otimes \dots \otimes L_m \Rightarrow S.$$

An interesting practical consequence of this formal conclusion is that many kinds of parallelism- and timing-relevant analyses can be performed directly on the “flattened” *leaf component network* of a system and issues with different levels of nesting can be avoided. Figure 4 shows how a system S can be interpreted as the root component of a tree of recursively nested components. This tree of components has a set of leaf components $L_1, \dots, L_m$. The software data-flow graph explained in the next section (see subsection *Data-*

Flow Analysis on page 6) is one possible presentation of a leaf component network.

Modeling an Automotive Software System with Dedicated Architectures

In the previous subsection we introduced a *basic system model*. We reuse the concepts of this basic system model to describe the software- and platform-component architecture as separate concerns of a system. More formally, an embedded automotive software system is a tuple $S = (L, R)$, where L describes the application *logic* realized by software, and R describes the *resources* provided by the platform. Both L and R inherited the concepts of the basic system model. Hence, L is the root component of the software component architecture and R is the root component of the platform component architecture. The following two subsections explain the meaning of model elements in the two architectures L and R .

Software component architecture

The software component architecture L of a system S describes the software-realized application *logic* of S . Used in the context of L , the basic model elements and concepts have the following meaning.

Software Components. A leaf software component is a model for a stateless or stateful function implemented e.g. by a C/C++-function, by a state-diagram of MATLAB/Simulink Stateflow or UML/SysML, or by a call to a well documented third-party library or driver API. It is assumed that each leaf software component can be executed without race-conditions and other non-deterministic side-effects on all other software components. By applying composition recursively, composite software components form more and more complex structures, e.g., single device functions, vehicle domain functions and finally a complete vehicle software component architecture. During deployment, a software component is allocated to be executed on a platform component, e.g., a processor core.

Software Ports. A software port is a typed interface point of a software component. Software ports define the expected input data and produced output data of the software-realized behavior of a component, e.g., the parameters of a C-function.

Software Channels. A software channel is a model for the transmission of messages from a source port to a destination port. If both ports are typed, the destination type must be assignable from the source type and the software channel transmits messages of the source type. During deployment, a software channel is allocated to be transmitted on a platform component, e.g., a field bus.

Platform component architecture

The platform component architecture R of a system S describes the set of *resources* provided by the platform of S . Again, we reuse the basic system model and in the context of