Chapter 1

INTRODUCTION

An embedded system is a micro computer system that is embedded into another technical device that itself does not appear as a computer. Embedded systems can be found everywhere. Examples are telecommunication devices, consumer electronics, automotive systems, building technology, etc. Embedded systems have a very high influence on the system industry. Nowadays, modern products cannot be realized in a competitive manner without embedded micro computer systems.

Embedded systems have to fulfill a large variety of requirements to be fully functional and accepted. One important requirement that is imposed on embedded systems is dependability. Dependability is important since embedded systems are often safety-critical. Examples are aircrafts, cars, and trains. The notion of dependability covers several important aspects reaching from reliability and availability to safety and security.

Another important requirement is efficiency. There are many different metrics for efficiency that are applied to embedded systems. One important metric, that is often also related to dependability, is run-time efficiency. For instance, the environment may impose certain timing constraints on the system, meaning that the system must not only produce correct results, but in addition must deliver these timely. In this case we speak about an *embedded real-time system*. In this thesis we focus on complex embedded systems that are subject to such real-time constraints. However, there are a large variety of additional efficiency metrics that are important, including energy consumption, code-size, weight, cost, etc.

Today, many embedded system applications are implemented using distributed architectures, consisting of several hardware nodes interconnected in a network. Thereby, each hardware node consists of a processor, memory, interfaces to I/O and to the network. The networks are arbitrated by specialized communication protocols that depend on the application area. For example, in the automotive electronics area communication protocols such as CAN [15, 1], Flex Ray [33], and TTP [120, 60] are common.

1.1 Design flow today

In this section we give a short overview of modern design flows for embedded systems. This overview is not meant to be exhaustive. It rather introduces the basic concepts and imposed requirements, which are then discussed using two popular example design flows: the Y-model known from hardware-software co-design, and the V-model utilized in the automotive industry. Based on these two design flows we will later motivate the techniques and contributions presented in this thesis.

1.1.1 Basic concepts

A coarse-grain overview of the embedded design flow is shown in Figure 1.1 (compare e.g. [26]).

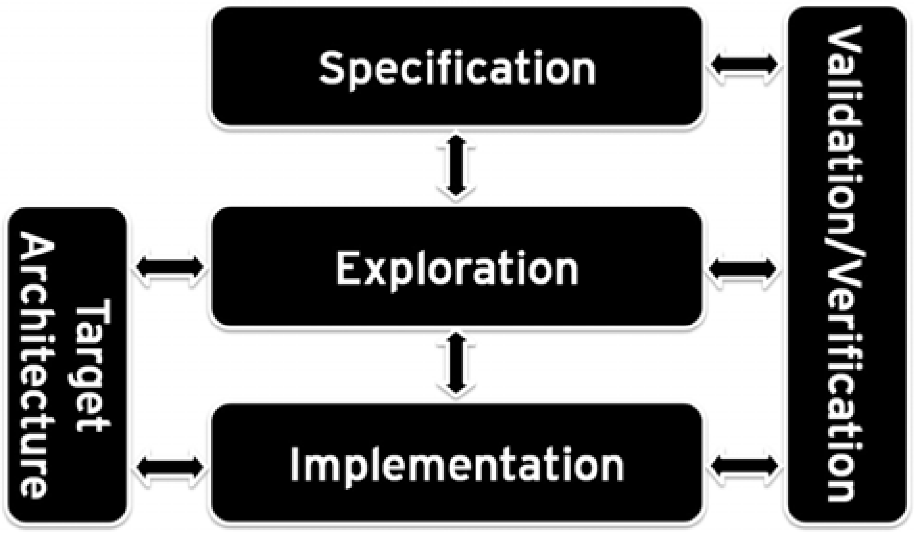

Fig. 1.1: Embedded Systems Design Flow - Coarse-grain overview

First of all the functionality that shall be realized by an embedded system must be specified. This is generally done using different specification languages with specialized models of computation. Examples are state charts [44], the ESTEREL programming language [9], data flow process networks [66], etc. Thereby, the choice of the utilized specification language very much depends on the type of application that shall be realized. State charts, for instance, are well suited for designing reactive applications (e.g. safety in the car: ABS, airbag, etc.), whereas data flow process networks best fit transformative applications (e.g. video processing, digital signal processing, etc.). However, usually not all needed components are designed from scratch. Some components may be re-used from previous designs (legacy code), or are purchased as intellectual property (IP) from external suppliers.

Often the developed specifications represent executable models of the desired system functionality. These so-called *executable specifications* enable the designer to perform early system optimization and design space exploration. For instance, data flow process networks are well suited to optimize buffer requirements and throughput of filtering applications. Generally, design space explorations based on such executable specifications help the designer to choose between functional alternatives, perform hardware-software partitioning, take scheduling decisions, etc.

The specification phase is usually performed without explicit consideration of the target architecture. This is attractive, since it allows the designer to focus on functional correctness ignoring the verification of the concrete implementation. However, once the executable specifications satisfy functional requirements, the focus shifts to target architecture design and implementation. Often parts of the architecture are fixed. Reasons include the need to utilize standard components (processor, memory, bus, RTOS, etc.), and maximum cost and size constraints per unit. During the implementation phase the main challenge is to ensure functional correctness, while successfully integrating all components onto the target architecture under the constraints imposed by the limited service capacity of the available resources.

At the current state-of-the-art, none of the above mentioned design steps can be guaranteed to be correct. Therefore, usually a test-bench is developed in parallel to validate the correctness of the intermediate or final design representations and implementations. Typically, various properties need to be validated to ensure the correctness of the developed system, including performance, dependability, energy consumption, etc.

In case the designers encounter difficulties during implementation, i.e. for instance non-compliance of the resulting system with required performance properties, they need to get back to the exploration or specification phase to find better alternative implementations. In the worst-case parts of the functionalities need to be re-designed, or the target architecture must be modified.

1.1.2 The Y-model

As a popular representative of an iterative design flow with successive refinements we now shortly discuss the Y-model known from hardware-software co-design [30]. A simplified version of the Y-model is shown in Figure 1.2.

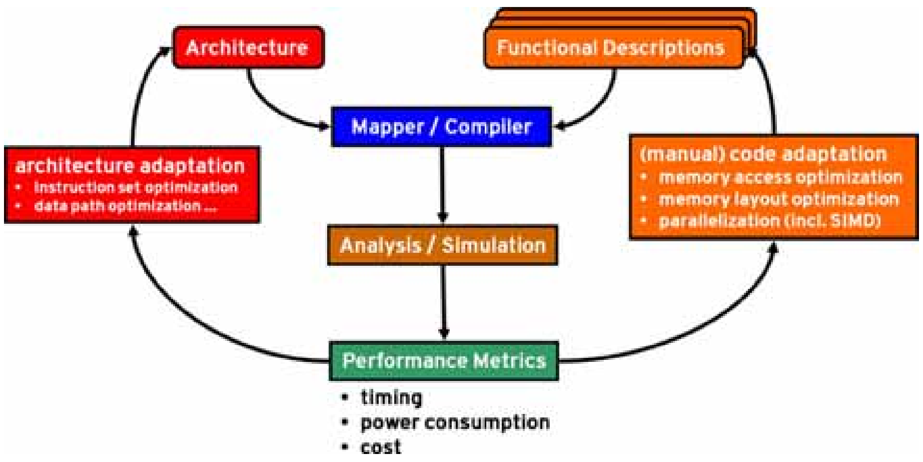

Fig. 1.2: Y-model known from HW/SW co-design

As in the generic embedded system design flow discussed above, the first step in the Y-model design flow consists in specifying platform-independent models for the intended functionality. Based on these models, object code is compiled and mapped on the target architecture. The resulting intermediate implementation is then tested and evaluated with respect to timing, power consumption, cost, etc., using simulation and analysis. Based on these metrics the designer decides about architecture and/or code adaptations. This process is iteratively repeated until a satisfactory design is found. Obviously, to evaluate a large number of different architectures, and thus to potentially obtain a better final implementation, it is desirable to achieve short turn-around times for one iteration.

The risk that is linked to the design flow according to the Y-model is relatively small, since the designer can react in each iteration to performance problems and solve them.

1.1.3 The V-model

The Y-model defines a very efficient design flow for hardware-software co-design. However, to be fully applicable an important prerequisite is that one design team controls most of the design parameters. Therefore, it only partly fits design tasks with shared responsibilities requiring sub-system integration. This is, for instance, an important issue in the automotive industry. Different sub-systems are independently developed and delivered by multiple external suppliers, and the OEM[1] has to integrate these into the car under a huge amount of constraints, including performance, safety, reliability, and consumer demands. In order to overcome this huge integration problem the so-called V-model [75] is used in the automotive industry. Figure 1.3 shows a basic version.

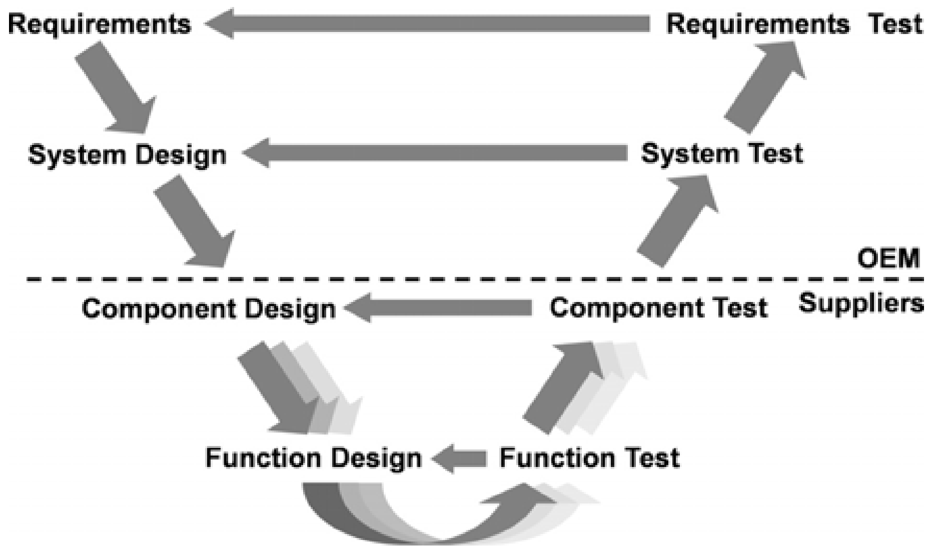

Fig. 1.3: V-model utilized in the automotive industry [75]

The V-model is based on the traditional top-down system engineering approach. First, the requirements imposed on the overall system are specified. Based on these requirements the OEM performs system design. This consists in the definition of the overall structure of the system functionalities and their interactions. Afterwards, the system is partitioned into several components, which are independently designed and tested by external suppliers according to given specifications. Once component design is finished the OEM's task is to integrate the delivered components into the final system, which mainly consists of network integration and a large amount of testing.

However, besides this idealized design flow an automotive system is usually not developed from scratch. In most cases an existing board net is taken as baseline for development. In other words, by reusing existing components time, work, and money can be saved. The V-model is, therefore, often supplemented by bottom-up methods. This does not compromise the V-model, since design has still to go through all the stages.

The basic V-model shown in Figure 1.3 also contains iterative refinements. However, these iteration are performed relatively late on proto-

[1]Original Equipment Manufacturer

types or real implementations. Consequently, iterative refinements are far more time intensive compared to the Y-model, and thus very expensive. Complex OEM-supplier dependencies additionally complicate design iterations.

In order to circumvent this problem the V-model was extended by the concept of *virtual design* (compare e.g. [34] and [104]). Figure 1.4 shows the extended V-model.

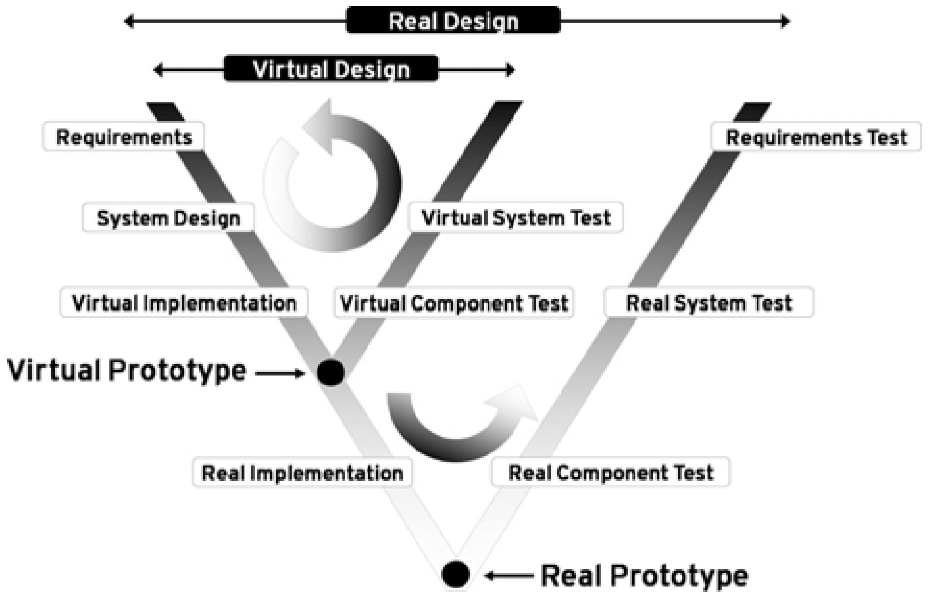

Fig. 1.4: V-model extended with the concept of virtual design [34, 104]

The extended V-model differs from the basic V-model in that it contains a second (smaller) V that is used to iteratively refine a virtual system model. Design iterations based on such abstract virtual models are usually far less time consuming and allow, therefore, to efficiently explore the design space. This is of great help for the system architect to choose an optimal system architecture (i.e. topology, number of nodes, number of buses, etc.) as well as an efficient function mapping.

1.2 Increasing design efficiency through reuse and modularity

In the development process of complex distributed embedded systems, reuse is recognized as key factor to meet growing productivity demands and cost pressure. In the ideal case whole product families and variants are based on the same set of reusable components allowing the designers to concentrate on basic differences between the products.

One important trend to achieve a high level of reusability is the so-called *platform-based design*. Nowadays, embedded system architectures are usually not designed from scratch. Instead so-called platforms are used. Platforms are programmable MpSoCs (multi processor system-on-chip) consisting of (multiple) cores, co-processors, specialized hardware components, buses, bridges, interfaces, etc. Platforms are often tailored for specific application domains. Examples are the Nexperia platform for multimedia and mobile digital audio applications from NXP [78], or the Tricore TC1796 platform from Infineon [110] used in the automotive domain (Figure 1.5).

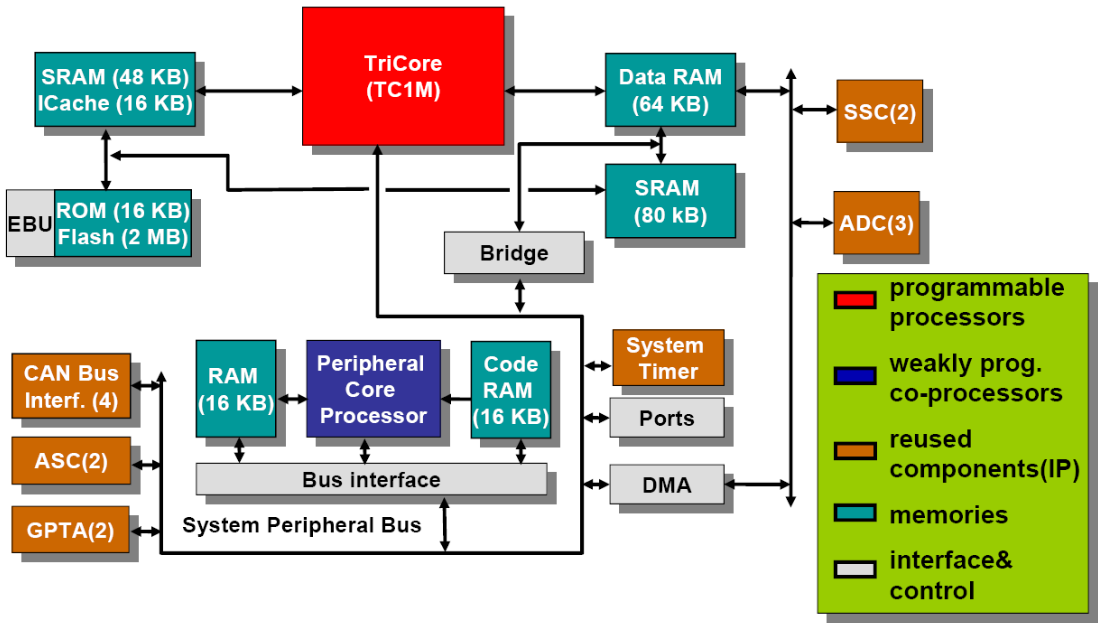

Fig. 1.5: Block diagram of the Infineon Tricore TC1796 micro-controller for automotive applications [110]

The platform-based design style is a so-called meet-in-the-middle approach. It combines the power of top-down methods with the efficiency of bottom-up styles [101]. Platform-based design can drastically reduce time to market while decreasing development and production costs [19]. ST Microelectronics estimates that each platform can lead to four or five products per year and, frequently, ten or more products over the lifetime of the platform [19].

In parallel there exist also efforts to standardize important system functionalities to ensure modularity, maintainability, reusability, scalability, and transferability on the software level. The Artist roadmap [14] identifies software as one key factor to successfully integrate components and sub-systems into complete systems. Classical middleware approaches such as CORBA [21] and COM/DCOM [20] are good examples for successful approaches to software modularization. Another prominent example known from the automotive industry is the AUTOSAR initiative [5]. The core idea of AUTOSAR is the definition of a run-time environment (RTE) executing so-called software components that communicate over a Virtual Function Bus (VFB). According to this concept, two functionalities can exchange data without knowing the exact communication path by using abstract communication ports of the RTE. Consequently, software can be developed independently of the real ECU topology in the car. The communication paths are defined relatively late in the design process. Obviously, such middleware concepts facilitate software portability and reusability.

1.3 Motivation

The platform-based design style and the standardization efforts on the software level help to increase design efficiency at the functional level. Example scenarios include the integration of several functionalities from different suppliers on the same node, or the distribution of (safety-critical) functionalities over several nodes. However, these concepts do not solve another key embedded systems challenge: the control of performance and other non-functional constraints, such as timing, power consumption, or dependability during the design process and over the service life of the product.

For instance, several components are often dependent on each other, and their integration can lead to complex and hard to predict performance degradation effects, which increase the design risk since they are often discovered late during the integration phase. In automotive systems, for example, the active front steering (AFS) interacts with other functionalities like the active roll stabilization (ARS). Also, not every functionality has its own sensors, data like individual wheel speeds are broadcasted over the bus and shared by many functionalities.

It is desirable to control the impact of sub-system integration on non-functional system properties, both during the design flow and during the lifetime of the product. The benefits are clear. The control of non-functional properties helps the system architect, on the one hand, to take the right design decisions before proceeding to implementation, and thus to decrease the design risk. On the other hand, it also assists the designer in conceiving systems with performance head-room for reuse, future extensions, updates, and bug-fixes. However, in modern design flows conformance to non-functional requirements is difficult to ensure, which is mainly due to the increasing size and complexity of modern systems, and the concurrent design [65] between OEM and suppliers involving dozens of parallel activities that need to be coordinated. As a consequence performance verification is still a major issue during design.

One possible approach to simplify sub-system integration is the so-called *conservative design.* Its principle consists in eliminating all coupling effects by strictly separating functionally independent sub-systems spatially and timely. The separation is achieved by assigning fixed memory spaces as well as static shares of communication and computational resources to each sub-system. Obviously, this strategy eliminates all complex timing effects and solves the integration problem. However, the resulting systems are not very efficient in terms of resource utilization and, thus, system efficiency and cost, since the statically assigned resources are not released for other functionalities in case of disuse. While this might be acceptable for highly safety critical systems, like for instance in avionics, such over-design is not an alternative in most other industrial sectors, including consumer electronics or automotive systems.

A promising starting point to overcome integration challenges while ensuring system efficiency are state-of-the-art performance analysis methodologies that have been proposed in the last decade [79, 77, 115, 84, 43, 113, 50, 46]. The different approaches operate at different levels of abstraction and allow a step-wise refinement of the utilized application and execution platform models. Figure 1.6 shows how performance verification can be applied along the V-model.

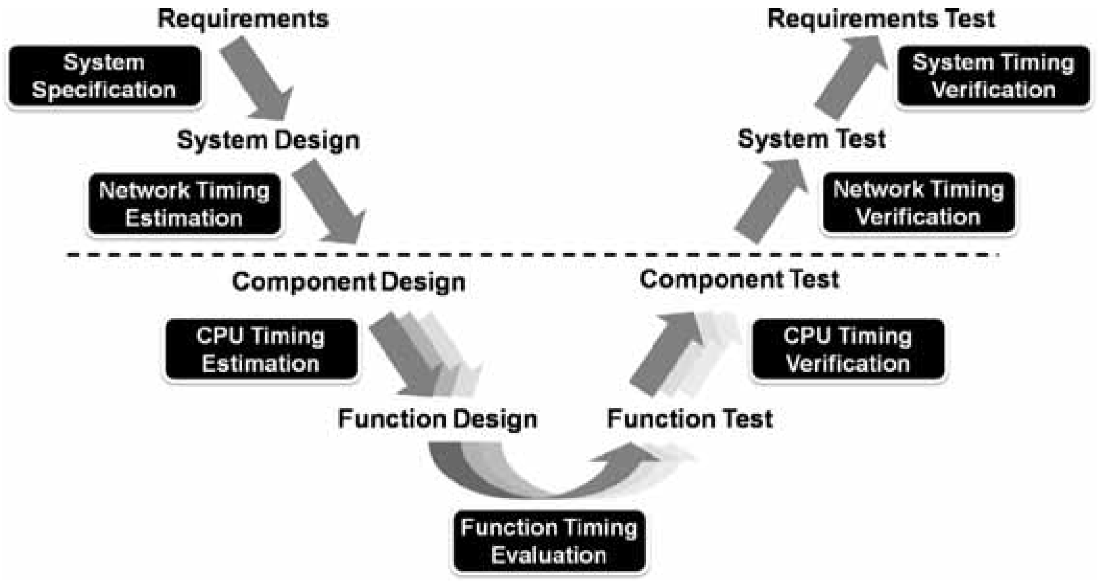

Fig. 1.6: Performance verification along the V-model

During the specification phase system performance is characterized based on data estimates. Even though this information might be coarse-grain and partly incomplete at the beginning, it can be utilized to derive first performance approximations helping the system architect to take architectural or mapping decisions (e.g. number of ECUs, bus bandwidth, etc.). Later, during component design and integration, these estimations can be refined step-by-step to obtain more accurate performance data.

Since the mentioned methods are based on rather abstract performance models and are able to analyze even large systems in a short time, they are perfectly suited for design space exploration. Design space exploration on top of these methods represents a valuable tool for the system architect to take good design decisions and systematically control system performance throughout the whole design process.

However, even though formal performance analysis methods are capable of deriving accurate performance data and are perfectly suited for design space exploration, their integration into real-world design flows is rather difficult. As a consequence, formal techniques have been mainly (successfully) applied to characterize and solve isolated performance issues like, for instance, timing analysis of single ECUs [97], bus bottleneck detection [98], or bus configuration [16]. A systematic and continuous performance verification flow spanning the whole design process allowing to control and optimize system performance is not trivial to accomplish. There are several reasons for this.

The first reason is that system performance is not composable in the general case. In other words, the system integrator (OEM) cannot automatically conclude that the integrated system satisfies its performance constraints from the fact that all supplied components are compliant to their specifications. The reason are complex performance dependencies that can often only be discovered during integration. One possibility to overcome this problem is to continuously verify and control performance during the design process across all involved design teams. However, in real-world design flows with OEM-supplier dependencies the required information exchange is problematic due to IP (intellectual property) protection issues. In fact, each of the involved design teams controls different parts of the system and is reluctant to share realization details.

There exist first approaches to solve this problem. The authors of [98], for instance, propose that each involved design team individually performs component-level analysis and communicates relevant results to functionally dependent system parts along the supply chain. By iteratively repeating such local analysis steps, OEM-supplier spanning timing analysis can be realized. Note that the performance data that needs to be exchanged mainly describes the dynamic communication behavior of the involved components (e.g. message jitters and frame offsets), and represents uncritical information with respect to IP protection. However, to fully exploit the benefits of formal methods in the context of concurrent design flows, such practical solution approaches for performance verification must be complemented with a flexible design space exploration framework of similar structure that supports iterative partial exploration steps at component level. The introduction of such an exploration framework is one of the aims of this thesis.