TUM

INSTITUT FÜR INFORMATIK

Motivation and Introduction of a System of Abstraction Layers for Embedded Systems

Martin Feilkas, Alexander Harhurin, Judith Hartmann, Daniel Ratiu and Wolfgang Schwitzer

TUM-I0925
September 09

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-09-I0925-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2009

Druck: Institut für Informatik der
Technischen Universität München

About the Document

In this document we introduce a system of abstraction layers as a backbone for a systematic development process of embedded software systems. We motivate the use of abstraction layers, describe the role of each layer and which aspects of a system should be modeled at it. However, this document does not describe which models should be used to represent the layer in detail.

The aim of the document is to give a basic introduction into our vision of a systematic development process along different abstraction layers. Although there exist ideas how the concrete models of different abstraction layers could look like [BFG+08, WFH+06, Pen08, BBR+05], we intentionally left out the details here. The consolidated definition of a concrete modeling framework and its domain-specific instances is one of the major goals of work package ZP-AP 1.2. In this sense the document should serve as basis for discussion within the SPES project.

Contents

1 Motivation 3

2 Abstraction Layers at a Glance 4

3 The Functional Layer 5

4 The Logical Layer 7

5 The Technical Layer 8

6 Crossing the layers 9

7 Related Work 9

A Guiding Questions for the Feedback 11

References 11

1 Motivation

Today, innovative functions realized by software are the key to competitive advantage in various application domains. Due to the increasing size and complexity of the system functionality and the interaction and dependencies between different features, the implementation of complex interdependent functions in software needs to be done in a mature, managed and predictable way.

Traditionally, the development of complex embedded systems involves the integration of different (relatively independent) electronic components, each containing its own standalone software. However, due to the demands for tight integration of functionality, software takes the primary role in more and more situations. In todays embedded systems, software plays the central role in assuring the functionality and quality of the product and at the same time generates the biggest costs and benefits. This situation demands a paradigm shift towards a technical system development where the development of software plays the primary role. In the software-centric development, the decomposition of systems along independent hardware units (containing their software) needs to be replaced by the decomposition along (logical) functionalities.

As illustrated in Figure 1-left, today's model-based software development involves the use of different models at different stages in the process and at different abstraction levels. Unfortunately, the current approaches do not make clear which kinds of models should be used in which process steps or how the transition between models should be done. Instead of a disciplined use of models, the choice of a particular modeling technique to be used is done in an ad-hoc manner and mostly based on the experience of the engineers or on the modeling capabilities of the tools at hand. This subsequently leads to gaps between models and thereby to lack of automation, to difficulties to trace the origins of a certain modeling decision along the process, or to perform global analyses that transceed the boundaries of a single model.

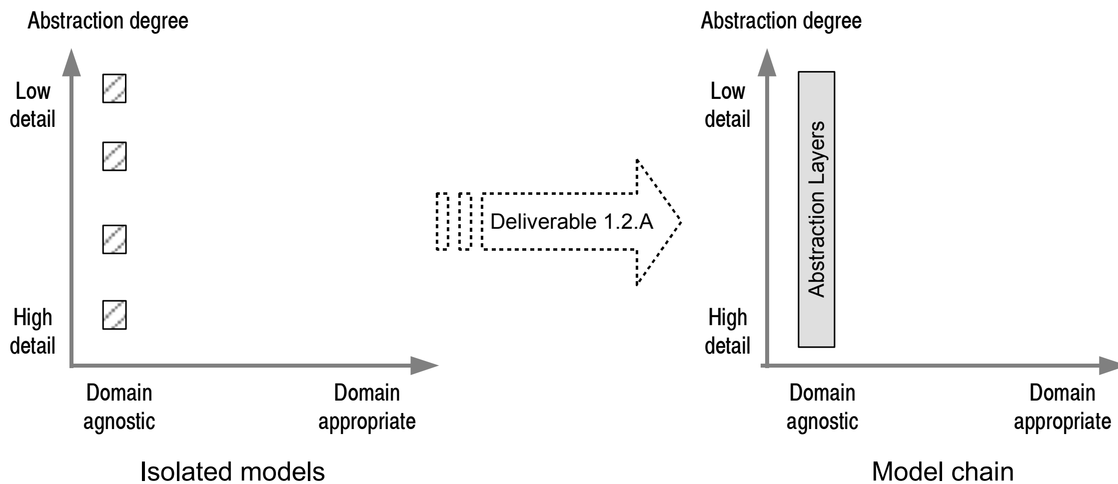

Figure 1: The scope of this deliverable

To master the complexity of todays embedded systems and to assure the seamless integration of the models of different development steps, we present a system of abstraction layers. These

layers and well-defined relations between them provide the basis for a systematic and comprehensive modeling framework, as illustrated in Figure 1-right containing models that cover the different abstraction layers from requirements, to design, and to deployment. The present report aims at realizing only the first step of the vision of a seameless and domain appropriate modeling theory presented in [CFF+09] – namely, to present a set of abstraction layers that seamlessly enable the use of models at different stages in the development process. To make these layers also domain appropriate is the scope of a future report.

The system of abstraction layers supports the consecutive refinement of model information from abstract layers down to concrete layers. Thereby, early models can be used to capture incomplete requirements and are afterwards step-by-step enriched with design and implementation information. Thus, the gap between (informal) requirements and the implementation is bridged and a higher degree of automation and a seamless development is enabled. Besides, since the higher layers abstract from technical details, an extensive reuse of models is supported. By this, our approach supports the construction of embedded software of high quality, shortening the development life cycle and decreasing the development costs.

2 Abstraction Layers at a Glance

We propose to describe systems along a set of abstraction layers which are built upon each other. Hereby, each abstraction layer provides self-contained concepts for the representation of the information, which is specific for each development phase. *At each layer it should be possible to model all relevant information explicitly. Furthermore, the used models should be based on a uniform modeling theory in order to assure that the models on different abstraction layers can be integrated. Therefore, the modeling theory must be rich enough to cover all relevant aspects of the system under construction.*

The abstraction layers are ordered hierarchically starting with (very abstract) high layers and leading to (very concrete) low layers. During the transition from a higher layer to a more concrete layer, the model information is enriched; i. e., the completeness of the models – with regard to the implementation of the system – is increased top down. Thereby, the transition has to be correct: in spite of the additional information the specification of models at a higher abstraction layer must be obeyed and completely realized in the lower layer.

> The system of abstraction layers is the backbone for the systematic and seameless model-based development of software for embedded systems.

We propose the description of the system by using the following layers

- functional layer,
- logical layer,
- technical layer.

In Figure 2 we intuitively present the three abstraction layers. As we will describe in the rest of this paper, the different layers are defined such that the specific challenges in developing software for embedded systems can be addressed. The level of abstraction decreases from the top to the bottom ranging from the (partial) description of the system based on its requirements to the description of its deployment on a technical platform. Each layer should make use of suitable models which allow the description of the relevant development aspects in an adequate manner. Furthermore, the models should be carefully chosen and seamlessly integrated in order to support the transition between the layers without loss of information.

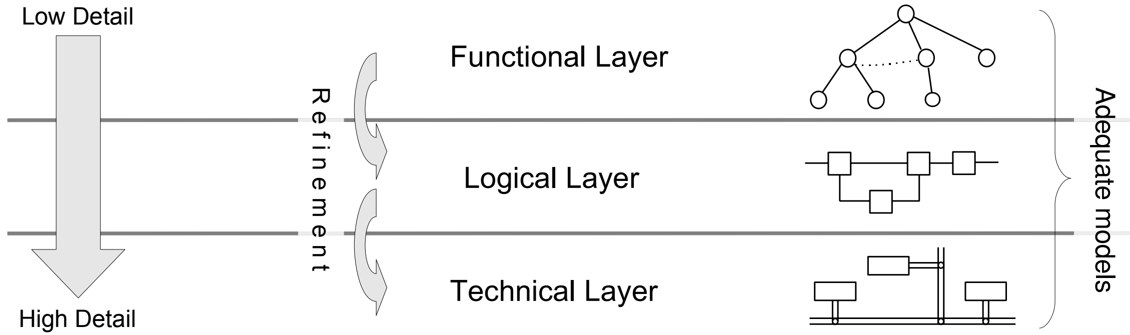

Figure 2: Abstraction layers intuition

The *functional layer* is responsible for a formalization of functional requirements, representing them hierarchically and additionally illustrating their dependencies. By using formally founded models, this layer provides the basis to detect undesired interactions between functions at an early stage of the development process. Due to the high level of abstraction, this layer is a step towards closing the gap to the informal requirements. Thus, it provides the starting point of a formally founded model-based development process.

The *logical layer* addresses the logical component architecture. Here, the functional hierarchy is decomposed into a network of interacting components that realize the observable behaviour described at the functional layer. Due to this layer's independence from an implementation, the complexity of the model is reduced and a high potential for reuse is created.

The *technical layer* describes the hardware platform in terms of electronic control units (ECUs) that are connected to busses. A deployment mapping is specified that maps the logical components (defined in the logical layer) onto these ECUs. Thus, a physically distributed realization of the system is defined and the complete middleware which implements the logical communication can be generated. Additionally, the interaction between the logical system and the physical environment via sensors and actuators is modeled.

3 The Functional Layer

The starting point for the functional layer is a set of requirements for the behavior of the system. These requirements can have different forms, for example the form of a textual documentation of a set of individual requirements (e. g., text documents in Telelogic DOORS) or of a collection of Use Cases. Assuming that all requirements to the system have already been collected in an

informal way, the functional layer represents the first step in capturing them as models and thereby is the entry point in the model-based development process.

Functional requirements are captured as function hierarchies consisting of functions and dependencies in-between. Each function realizes a piece of black-box functionality and is defined by its syntactic interface and its behavioral specification. The syntactic interface comprises the ports via which the function is connected to the environment, and the behavioral specification defines the message exchange on these ports.

Aims The central aims of the functional layer are:

- Definition of the boundary between the system under consideration and its environment;
- Definition of the syntactical interface: abstract information flow between the system and its environment;
- Consolidation of the functional requirements by formally specifying the requirements on the system behavior from the black-box perspective;
- Mastering of feature interaction: detection and resolution of inconsistencies within the functional requirements;
- Reduction of complexity by hierarchically structuring the functionality from the user's point of view;
- Understanding of the functional interrelationships by collecting and analyzing the interactions between different (sub-)functionalities.

The functional layer provides a hierarchically structured specification of the system behavior as it is perceived by the user at the system boundary (also known as *usage behavior*). Hereby, a user may be a person but also another system. Thus, the *system boundary* of the entire system is determined at the functional layer. The functional layer comprises the definition of the system interface with surrounding systems and users. The behavior of the entire system will then be specified from the *black-box* perspective by describing the exchange of messages between the system and its environment. Hereby, the abstract data flow is specified, namely the intentional meaning of the exchanged data (as opposed to the concrete message types). By formally describing the requirements we lay the basis to measure the completeness and detect inconsistencies of the requirements.

The overall system functionality can be obtained as the combination of sub-functions (with respect to the dependencies between them). Hereby, the decomposition/structuring is not guided by architectural or technical aspects but only done along the functional aspects required by the users.

Thereby, an informal requirement can be realized by one or several functions and a function is able to realize one or more informal requirements. The formalization of requirements on the functional layer makes it possible to analyze existing requirements and thus to detect and solve inconsistencies (e. g., *feature interaction*) and missing requirements.

4 The Logical Layer

The logical layer represents a realization of the functionality (including their dependencies) defined within the functional layer by a network of communicating logical components. Components declare a logical interface in terms of ports that can be connected via channels. The behavior of a component is either defined directly (for example using an automaton) or in a composite way by a network of sub-components. Thus, an entire system is specified by a tree of hierarchical components.

Aims The main aims of the logical layer are:

- Provide an architectural view of the system by partitioning the system into logical communicating components;
- Definition of the total behavior of the system (as opposed to the partial behavior specifications described at the functional layer);
- Simulation of the system based on the internal data flow between components;
- Reuse of already existent components;

The functional and logical layers are two orthogonal structures of the system functionality. A brief comparison of both layers is sketched in Table 1.

Functional Layer	Logical Layer
Problem domain	Solution domain
Black-box view of the system	White-box view of the system
Structured by user's functions	Structured by architectural entities
Used primarily to specify what the system should do	Used primarily to design the system
Functional specification may overlap and must be checked for inconsistencies (horizontal decomposition)	Network of communicating components (vertical decomposition)
Captures the functionality of the system	Works as a first cut at design
(Possibly) partial behavioral specification	Total behavioral specification

Table 1: Brief Comparison of the Functional and Logical Layers

In contrast to the functional layer, in the logical layer, emphasis is no longer put on the formalization of the functionality that can be observed at the system boundary but rather on the structuring and partitioning of the system into logical communicating components. The entire behavior of these components realizes the behavior determined by the functional layer. In the broadest sense a logical component represents a unit which provides one or more functions of the functional layer. Generally, there is a $n : m$ relationship between functions from the functional layer and components of the logical layer.

At the logical layer, structuring is done by means of the most diverse criteria, such as, for example, a partitioning according to the hierarchy of the functional layer, according to the organizational structure within the company, or according to non-functional requirements.

However, it is important to note that the logical layer abstracts from implementation details. Therefore, some (non-functional) requirements should better be addressed in the technical layer.

The logical layer provides a complete description of the system functionality, however, without anticipating technical decisions with regard to implementation (e. g., the platform on which the components will be deployed).

5 The Technical Layer

The technical layer serves as a "target model" for the model-based development of software for embedded systems. It represents the layer with the lowest level of abstraction. This layer provides models of the hardware on which the application logic has to run. A deployment mapping is specified that assigns an ECU to every logical component. The technical layer represents an abstraction from the details of the hardware that is employed. It focuses on the aspects relevant for running the software in a distributed environment, e. g., the utilization of ECUs, communication busses and peripheral devices.

Aims The main aims of the technical layer are

- Describing the hardware topology on which the system will run including important characteristics of the hardware;
- Describing the actuators, sensors, and the MMI that are used to interact with the environment;
- Enabling a flexible (re-)deployment of the logical components to a distributed network of ECUs;
- Ensuring that the behavior of the deployed system conforms to the specifications of the logical layer (e. g., time constraints).

The technical layer describes the topology of the hardware which is employed to execute the system. A hardware topology consists of one or several ECUs that are connected by bus systems. Additionally, actuators and sensors are connected to the ECUs. Given a model of the logical components and a model of a hardware topology a deployment model can be specified. Such a deployment model defines a mapping for each logical component to exactly one ECU.

After specifying a deployment for the logical components a port mapping model is needed. The port mapping model defines a mapping of the input and output ports of the logical components that are deployed onto a specific ECU to the sensors and actuators that are connected to this ECU. Thus, the port mapping model defines the interaction of the logical behavior with the physical environment.

The logical components are connected via logical channels to enable communication between them. Depending on the deployment model these communication connections have to be implemented as local communication on a single ECU or as remote communication via one or several bus systems. Based on the specifications of the topology and the deployment, the complete middleware can be generated.

There are several constraints that have to be fulfilled to ensure that the deployed system behaves as specified in the logical layers: The ECUs must be 'fast enough' and provide enough memory to be able to execute the logical components in time without violating assumptions made in the logical layer. Also the bus systems must be capable to transfer all the signals in time. These constraints must be proven correct by a set of static anlysis techniques (such as schedulability analysis).

6 Crossing the layers

Once the layers are defined, it is important to define a clear and systematic method to bridge the layers. Ideally, the transition between a more abstract layer and a more concrete one should be done exclusively by adding more details and without loosing the abstract information. In this case the concept of *refinement* can be employed to prove that the models at a more concrete layer fulfill the requirements of the models at a more abstract layer.

The functional layer models only functional requirements. However, in practice there are a multitude of other non-functional requirements and constraints that need to be considered. These influence the transition between the models at different abstraction layers. As illustrated in Figure 3, the refinement of an abstract model into a more concrete layer can be done in a multitude of ways – i. e., there are several concrete models that are refinements of the abstract model and that form the design space. Choosing one or another of these refinements is a matter of engineering and design trade-offs and is influenced by non-functional requirements such as reliability, or constraints such as the need to reuse a particular platform.

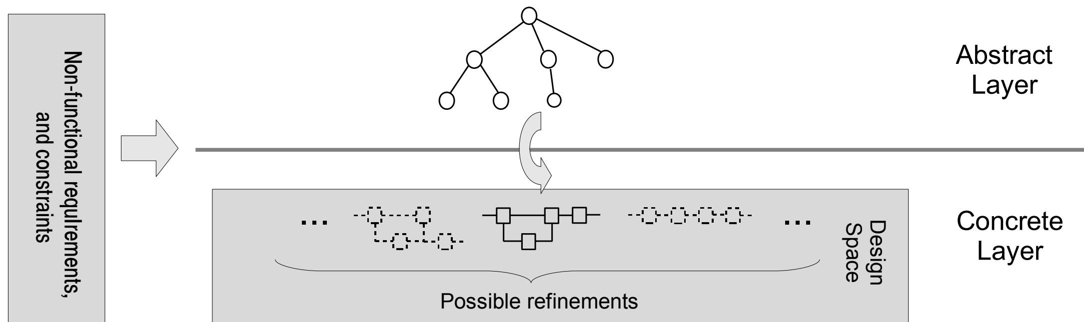

Figure 3: Crossing the layers intuition

7 Related Work

The presented approach to reduce complexity by a systematic software development for embedded systems along domain-specific architectural layers is not new. In this section, we shortly sketch approaches that have influenced the presented system of abstraction layers.

EAST ADL The EAST ADL (Electronics Architecture and Software Technology – Architecture Definition Language) was developed in 2004 in the scope of the ITEA1 project EAST EEA [ITE08], consisting of automotive manufacturers, suppliers, software manufacturers and universities. The EAST ADL had been designed for the automotive domain and describes software-intensive electric/electronic systems in vehicles on five different abstraction layers starting from high-level requirements and features which are visible to the user to details close to implementation, such as constructs of operating systems (e. g., tasks) and electronic hardware.

The architectural layers of the EAST ADL served as a basis for the presented system of abstraction layers. With regard to contents and aims the Vehicle Feature Model and the Functional Analysis Architecture of the EAST ADL can be seen as a counterpart of the functional layer. The Functional Design Architecture vaguely corresponds to the logical layer and the abstraction levels of the Function Instance Model, the Platform Model and the Allocation Model vaguely correspond to the abstraction level, which can be found on the technical layer.

The focus of the EAST ADL is hereby placed on describing the structural aspects and not on describing the behavior. The description of the behavior mainly results from external tools. Moreover, a formal basis for the models is missing within the East ADL. In contrast, our aim is to describe the structure and the behavior of the system in an integrated way based on a uniform formal basis. Thus, we want to go one step further and intend to create a basis for an integrated and systematic development process based on a formal fundament.

Model Driven Architecture (MDA) Analogously to our approach, the MDA approach [MM03] aims at mastering the complexity of todays systems by describing the system on differently abstraction levels: It starts with an informal description of the system by the Computation Independent Model (CIM). Based on the CIM, the Platform Independent Model (PIM) defines the pure system functionality independently from its technical realization and at last the PIM is translated to one or more Platform Specific Model (PSM) that computers can run.

While we are aiming at a system of abstraction layers which is specific for the development of embedded systems or even specific domains, the MDA is a general purpose approach. Thus, the presented system of abstraction layers can be seen as instantiation of the MDA-layers. Besides, as mentioned in comparison to the EAST-ADL, our aim is to provide a uniform formal basis for the models used at the different layers. MDA, however, is missing such a formal basis.

Further systems of abstraction layers At the chair of Software and Systems Engineering of Prof. Broy, there already exists prelimary work on abstraction layers. As a result of the research cooperation "software engineering for the automobile of the future - mobilSoft" between automotive manufacturers, suppliers and research institutions, a set of automotive-specific abstraction layers has been designed and described in [WFH+06]. Also in other projects, e. g., AutoMode [BBR+05], VEIA [GHH07] REMSES [Pen08] systems of abstraction layers have been used to some extend. In [WFH+06], a first step has been made to integrate the existing research results into an integrated architectural model for engineering embedded software-intensive systems. We have carefully examined and taken into consideration all existing approaches while developing the presented system of abstraction layers.

A Guiding Questions for the Feedback

In order to get feedback about the adequacy of the abstraction layers for the industry we propose the following guidelines. Please think of a project that is relevant for the state of the practice today in your company / application domain. If you think of the models that you are using please answer the following questions:

- How early in the process do you start modeling (since requirements, or design, ...)?
- How are you using the models (e. g., only for documentation and communication, for complex analysis, for code generation, ...)? Which models are you using in each of these cases (e. g., statecharts, component diagrams, data-flow diagrams, ...)?
- Which tools (or modeling dialects) do you use (e. g., UML-based tools, Statemate, ...)?
- Which of the models can you map with which abstraction layer?
  - How do you specify the functionality?
  - How do you specify the architecture?
  - How do you specify the technical architecture on which the software is to be deployed? How do you describe the deployment?
- Are there models that cover several abstraction layers?
- At which layer don't you have any model?
- Can you map all your models using the defined abstraction layers? If not, which information are you missing?
- If you are using models at different abstraction levels, how do you realize the transition between the models?
- How do you document the transition between models (e. g., through traceability links)?
- Do you find the layers adequate / sensible?
- Which layers are superfluous and which are missing?

If you have more feedback to give, please feel free to do so. The above questions represent only a minimal feedback about the state of the practice in your industry domain with respect to the abstraction levels. **Thank you!**

References

[BBR+05] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Núria Mata, Robert Sandner, and Dirk Ziegenbein. AutoMoDe — notations, methods, and tools for model-based development of automotive software. In *Proceedings of the SAE 2005 World Congress*, volume 1921 of *SAE Special Publications*, Detroit, MI, April 2005. Society of Automotive Engineers.

[BFG+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes architekturmodell für das engineering eingebetteter software-intensiver systeme. Technical Report TUM-I0816, Technische Universität München, june 2008.

[CFF+09] Alarico Campetelli, Martin Feilkas, Martin Fritzsche, Alexander Harhurin, Judith Hartmann, Markus Hermannsdörfer, Florian Hölzl, Stefano Merenda, Daniel Ratiu, Bernhard Schätz, and Wolfgang Schwitzer. Model-based development – motivation and mission statement of workpackage zp-ap 1. Technical report, Technische Universität München, 2009.

[GHH07] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, volume 0, pages 349 – 358, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[ITE08] ITEA. EAST-EEA Website. http://www.east-eea.net, January 2008.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[Pen08] Birgit Penzenstadler. Tackling automotive challenges with an integrated re & design artifact model. In *Intl. Workshop on System/Software Architecture*, 2008.

[WFH+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl, and Sabine Rittmann. An architecture-centric approach towards the construction of dependable automotive software. In *Proceedings of the SAE 2006 World Congress*, 2006.