TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

# SPES 2020 Deliverable D1.2.A

Initiales System an Abstraktionsebenen

Software Plattform Embedded Systems 2020

| | |
|---|---|
| Author: | Judith Thyssen, TUM-SSE |
| Version: | 1.0 |
| Date: | 31.12.2009 |
| Status: | Released |

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

## Inhaltsverzeichnis

# 1 Einführung

Ziel des Deliverable D1.2.A ist die Bereitstellung einer initialen Systematik an Abstraktionsebenen für das Projekt SPES durch die TUM-SSE. Diese Systematik an Abstraktionsebenen liefert ein Framework für den durchgängigen Einsatz von Modellen über den gesamten Software Entwicklungsprozess. Auf Basis dieser initialen Version sind im weiteren Projektverlauf Änderungswünsche zu erfassen, die Abstraktionsebenen weiter zu detaillieren und domänenspezifische Instanzen zu erstellen. Das Deliverable D1.2.A umfasst zwei Teildokumente:

1. D1.2.A-1 „Abstraction Layers – Motivation and Introduction of a System of Abstraction Layers for Embedded Systems“ (siehe Anhang A)
   Dieses Dokument enthält einen initialen Vorschlag für ein System von Abstraktionsebnen bestehend aus den drei Ebenen Funktionale Ebene, Logische Architekturebenen und Technische Ebene.

2. D1.2.A-2 „A System of Abstraction Layers for the Seamless Development of Embedded Software Systems“ (siehe Anhang B)
   Die in diesem Dokument beschriebene Systematik an Abstraktionsebnen ist eine Erweiterung des im ersten Dokument beschriebenen Systems, in die das Feedback und die Ergebnisse zahlreicher Diskussionen innerhalb von SPES eingeflossen sind.

## A Deliverable D1.2.A-1

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

# SPES 2020 Deliverable D1.2.A-1

## Abstraction Layers

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

Software Plattform Embedded Systems 2020

| | |
|---|---|
| Author: | Martin Feilkas |
| | Alexander Harhurin |
| | Judith Hartmann |
| | Daniel Ratiu |
| | Wolfgang Schwitzer |

| | |
|---|---|
| Version: | 1.0 |
| Date: | 31.07.2009 |
| Status: | Released |

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

## 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

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.

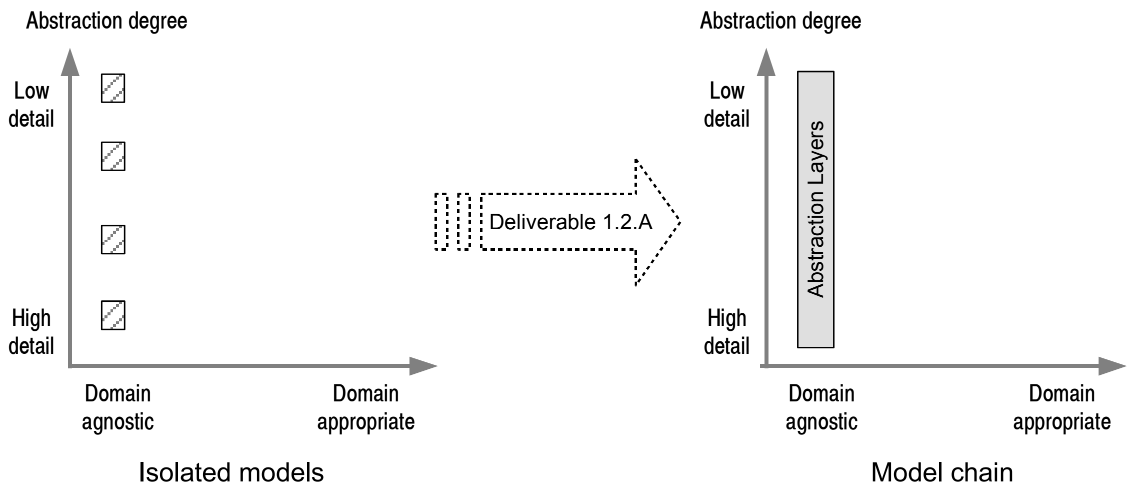

**Figure 1: The scope of this deliverable**

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 seamless 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.

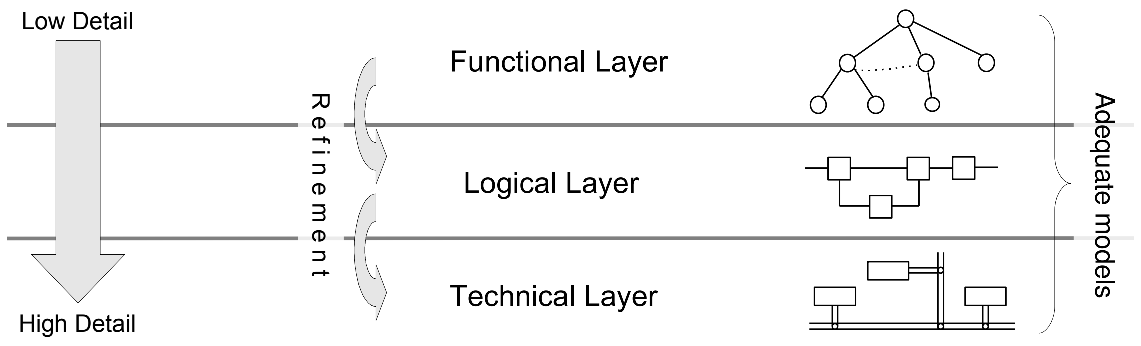

**Figure 2: Abstraction layers intuition**

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.

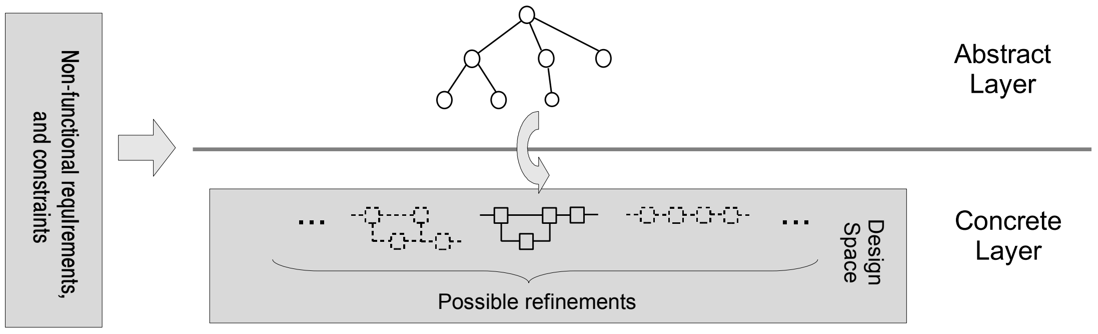

**Figure 3: Crossing the layers intuition**

## 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.

## B Deliverable D1.2.A-2

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

# SPES 2020 Deliverable D1.2.A-2

# A System of Abstraction Layers for the Seamless Development of Embedded Software Systems

Software Plattform Embedded Systems 2020

|  |  |
|---|---|
| Author: | Daniel Ratiu |
|  | Wolfgang Schwitzer |
|  | Judith Thyssen |
| Version: | 1.0 |
| Date: | 15.10.2009 |
| Status: | Released |

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

| | |
|---|---|
| Projektbezeichnung | SPES 2020 |
| Deliverable | D1.2.A |
| Autoren | Daniel Ratiu (TUM)<br>Wolfgang Schwitzer (TUM)<br>Judith Thyssen (TUM) |
| Diskussionsbeiträge | Alexander Metzner (OFFIS)<br>Eike Thaden (OFFIS)<br>Carsten Strobel (EADS-IW)<br>Ernst Sikora (UniDuE)<br>Kim Lauenroth (UniDuE)<br>Jörg Holtmann (UPB) |
| Qualitätssicherung | Kim Lauenroth (UniDuE) |
| Artefaktzustand | abgeschlossen |
| Freigabestatus | öffentlich |

## About the Document

Model-based development aims at reducing the complexity of software development by the pervasive use of adequate models throughout the whole development process starting from early phases up to implementation. Models are used by different stakeholders (from system integrators to suppliers) as the single development artifact. They describe the system both at different levels of granularity and from different points of view. Models can describe the whole system in a coarse grained manner as well as small parts of the system in high detail – specification models developed by system integrators are refined by suppliers, that can in turn be integrators of lower level functionality modeled a level deeper in the development hierarchy. Models describe the system from different points of view, each view concentrating on a particular kind of information that is relevant for describing interesting aspects of the system. For example, models of the system functionality are implemented by using models of the architecture that are subsequently deployed on models that describe the hardware.

In this document we present a conceptual framework that holistically comprises models that are used for development of the software product and that are at different levels of abstraction. We do this by using adequate abstractions for different development stages while ignoring the information that is not relevant at a particular development step or for a certain stakeholder. The abstraction is achieved in terms of the granularity level of the system under study (e. g., system, sub-system, sub-sub-system) and in terms of the information that the models contain (e. g., specification of functionality, description of architecture, deployment on specific hardware).

We also present the relation between different models (that describe different views of the system or are at different granularity levels) in order to offer guidelines about their usage in a sound and disciplined manner. However, in this document we do not address the process to be followed for building these models.

## Contents

## 1 Motivation

Today, in various application domains (e. g., automotive, avionics), software plays a dominant role. Central challenges that are faced during the development of today's embedded systems are twofold: firstly, the rapid increase in the amount and importance of different functions realized by software and their extensive interaction that leads to a combinatorial increase in complexity, and secondly, the distribution of the development of complex systems over the boundaries of single companies and its organization as deep chain of integrators and suppliers that have to get synchronized in order to realize the end-product. Model based development of software promises to provide effective solutions to these problems by the pervasive use of adequate models in all development phases as main development artifacts.

As illustrated in Figure 1-left, today's model-based software development involves different models at different stages in the process and at different levels of abstraction. 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. This subsequently leads to gaps between models and thereby to a lack of automation, to difficulties in tracing the origins of modeling decisions along the process, and to difficulties in performing global analyses or optimizations that transcend the boundaries of a single model.

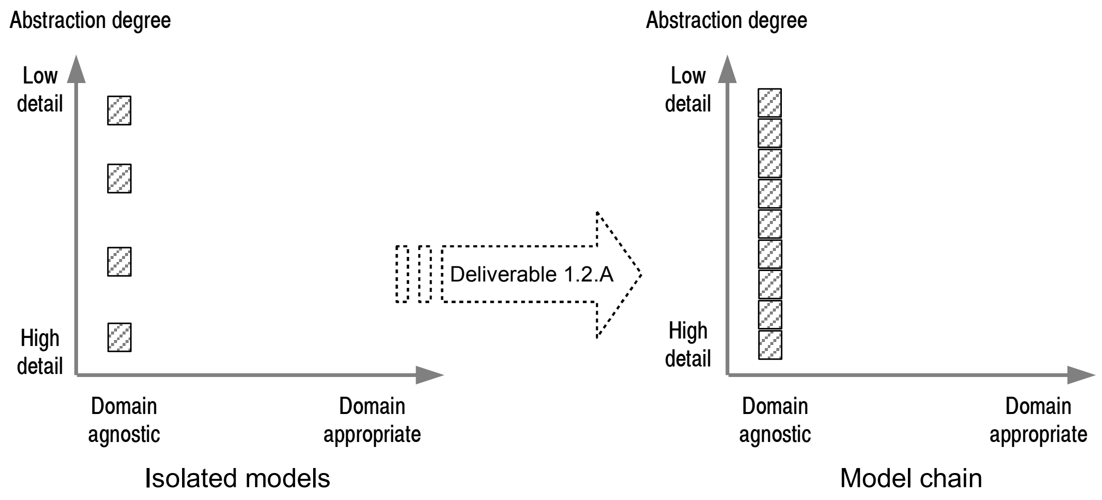

**Figure 1: The scope of this deliverable**

Our aim is a seamless integration of models, as illustrated in Figure 1-right, into model chains. A chain of models contains models that cover the relevant development views over the system from user functionality to logical or technical architectures, and that describes the system at different granularity levels such as system, sub-system or basic components. The models in a chain represent different perspectives over the system that capture different system aspects while leaving apart unnecessary details. When they are integrated, these perspectives (many times referred to as 'views') offer enough information to describe all interesting aspects of the system and their interaction. This deliverable, that defines a system of abstraction layers, represents a first step to achieve the aims of the SPES project ([CFF+09]), namely, to define domain appropriate model chains (compare Figure 1 with Figure 3 from [CFF+09]).

**Achieving abstraction.** When talking about "abstraction" we mean the reduction of complexity by reducing details. There are two typical possibilities to reduce details: Firstly, we can describe the system separately at different levels of granularity and without caring about the details of its sub-systems or of the super-system into which it will be integrated. Secondly, we can describe the system from different points of view focusing on particular aspects of the system (and leaving other aspects unaddressed). Thus, our approach to manage complexity is based on two kinds of decomposition:

- **Whole-part decomposition.** One manner to deal with complexity is to apply the "divide and conquer" principle and to decompose the whole system into smaller and less complex parts. These parts can in turn be regarded as full-fledged systems themselves at a lower level of granularity. Thereby, we structurally decompose the system into its sub-systems, and sub-systems into sub-sub-systems until we reach basic blocks that can be regarded in an atomic manner.
- **Distinct development views.** The second manner to deal with complexity is to focus only on certain aspects of the system to be developed while leaving away other aspects that are not interesting for the aims related to a given development view. The essential complexity of a system is given by the (usage) functionality that it has to implement. By changing the perspective from usage functionality towards realization, the complexity is increased by considering additional implementation details (e. g., design decisions that enable the reuse of existing components).

Consequently, our modeling framework comprises two different dimensions as illustrated in Figure 2: one given by the *level of granularity* at which the system is regarded and the second one is given by different *software development views* on the system.

- **Levels of granularity.** A system is composed of sub-systems which are at a lower granularity level and which can themselves be regarded as systems (Section 2). Often, the sub-systems are developed separately by different suppliers and must be integrated afterwards. Especially for system integrators, the decomposition of a system into sub-systems and subsequently the composition of the sub-systems into a whole are of central importance. As a consequence, we explicitly include the different decomposition layers of systems in our framework. The different granularity levels constitute the vertical dimension of our framework (Figure 2 - vertical).
- **Software development views.** A system can be regarded from different perspectives, each perspective representing different kinds of information about the system. We call each of these perspectives a "software development view" (Section 3). Our framework contains the following development views: the functional view, the logical (structural) view, and the technical view. The views aim to reduce the complexity of the software development by supporting a stepwise refinement of information from usage functionality to the realization on hardware. Thereby, early models are used to capture the usage functionality and are afterwards step-by-step enriched with design and implementation information (Figure 2 - horizontal).

At the most abstract level (upper-left entry of the matrix from Figure 2) are the functional (usage) requirements to the system, the user functionalities. The most detailed level (bottom-right

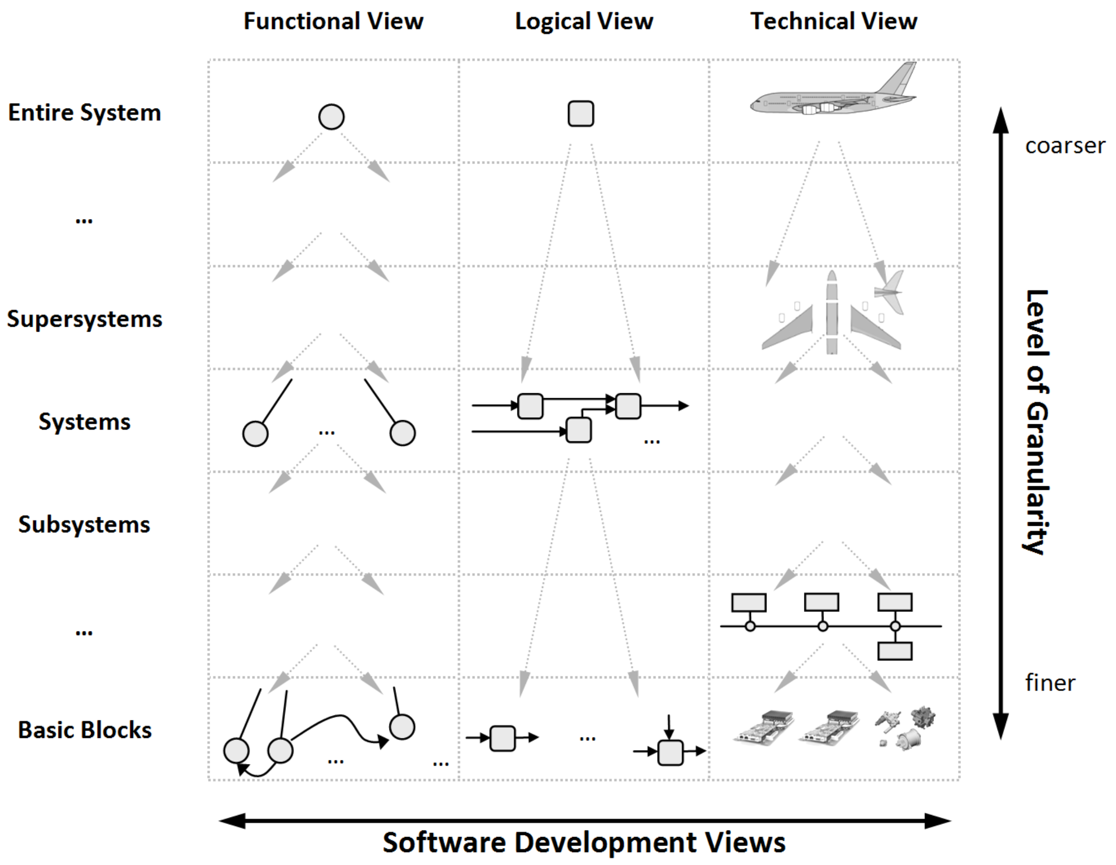

**Figure 2: Two approaches to achieve abstraction: a) different levels of granularity (vertical), and b) different software development views (horizontal)**

entry of the matrix of Figure 2) contains the complete functionality including all realization details of basic components.

The relation between two subsequent granularity layers is defined in such a way that the subsystems at a finer granularity level correspond to the components defined by the architecture at the upper level: the logical sub-components (or technical parts) at the system level, represent the sub-systems at a lower level of granularity. Inside a single level of granularity, the relation between different development views is done through an explicit allocation: the functionality is allocated to logical components which are in turn allocated to hardware parts.

**Outline** As already introduced, Figure 2 provides an overview of our approach for achieving abstraction. In the following sections the two dimensions of the modeling framework – the structural decomposition of a system into its sub-systems and components (Section 2) and the different development views (Section 3) – are explained in detail. In Section 4 we tackle the issue of crossing the different abstraction levels both on the horizontal between software development views and on the vertical between different granularity levels.

## 2 Granularity Levels

In order to cope with the complexity of today's systems, we decompose them into sub-systems. Each sub-system can be considered as a system itself and can be further decomposed until we reach basic building blocks. As a result we obtain a set of granularity levels upon which we regard the system, e. g., system level – sub-system level – sub-sub-system level – ... – basic block level. This system decomposition enables us to seamlessly regard the systems at increasingly finer levels of granularity. Since the system is structured into finer parts which can be modeled independently and aggregated to the overall system afterwards, the system decomposition allows us to reduce the overall complexity following the "divide and conquer" principle.

*Note:* In order to enable the decomposition of the system into its components and the subsequent construction of the system out of components, the models used to describe the system must be based on theories that are compositional. The use of compositional theories is a central requirement for a seamless comprehensive modeling theory as presented in deliverable D1.1.A [HHR09].

**From systems to sub-systems.** Each system can be decomposed into sub-systems according to different criteria that are many times competing. For example, we might choose to decompose the system into sub-systems that map at best either the functional hierarchy, or the logical architecture, or the technical architecture (these software development views are presented in Section 3 in detail). Depending on which of these software development views is prioritized, the resulting system decomposition into sub-systems looks different: the more emphasis is put on decomposing the system based on one of the software development views, the more the modularization concerning the other software development views is neglected and other aspects of modularity are lost (this phenomenon is known as "tyranny of dominant decomposition" – illustrated in Figure 3).

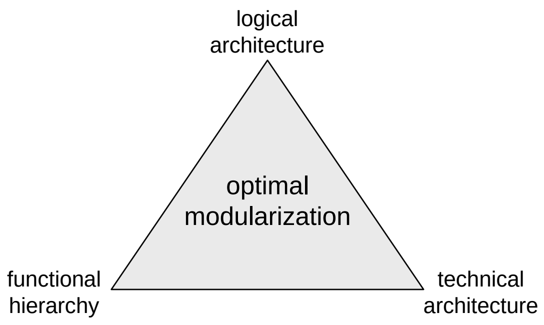

**Figure 3: Contradictory factors of influences for the system decomposition.**

Today the whole-part decomposition is primary driven by technical concerns. As a consequence, many systems are often not optimally structured with respect to the functionality they implement. In turn, this leads to functionalities that are weakly modularized (if at all) and scattered over different sub-systems (phenomenon similar to cross-cutting concerns). This subsequently leads to difficulties in the integration of sub-systems in order to realize the desired functionalities.

On the one hand, for models situated at coarser granularity levels, many times the established industry-wide domain architectures determine the decomposition of the system into sub-systems. The domain architectures are primary determined by the physical/technical layout of a system, that is subsequently influenced by the vertical organization of the industry and division of labor between suppliers and integrators (most suppliers deliver a piece of hardware that contains the corresponding control software – e. g., an ABS system contains a controller and its corresponding software). On the other hand, for the models situated at finer granular layers, their decomposition into sub-systems is influenced by other factors, e. g., an optimal modularization of the logical component architecture in order to enable reuse of existing components.

*However, since the logical architecture acts as mediator between the functional and technical view (see Section 3.2), we strongly believe that the logical modularization of the system decomposition should be reflected mainly in the part-whole decomposition of the system.*

**Integrators vs. suppliers.** According to the level of granularity at which we regard the system, we can distinguish between different roles among stakeholders (illustrated in the pyramid from Figure 4): the end users are interested only in the top-level functionality of the system (the entire system), the system integrators (system engineers) are responsible for integrating the high-granular components in a whole, while the suppliers are responsible for implementing the lower level components. An engineer can act as a system integrator with respect to the engineers working at finer granularity levels, and as supplier with respect to the engineers working at a higher granularity level. This top-down division of work has different depths depending on the complexity of the end product – for example, in the case of developing an airplane the granularity hierarchy has a high depth, meanwhile for developing a conveyor the hierarchy is less deep.

## 3 Software Development Views

The basic goal of a software system is to offer the required user functionality. There can be other goals that need to be considered such as efficiency, reliability, reuse of other existent systems or integration with legacy systems. However, in our opinion the functional goals are primordial since it is meaningless to build a highly efficient or reliable system that does not perform the desired functionality. However, we acknowledge that in some systems these secondary goals are as important as the implementation of the functionality – e. g., for safety-critical systems such as a flight controller the reliability and error tolerance requirements are central as well. Regarding the system purely from the point of view of the user functionality that it implements offers the highest level of abstraction since implementation, technical details and the other concerns required by the non-functional requirements are ignored (abstracted away).

By changing the perspective from usage functionality towards implementation, we add more (implementation) details that are irrelevant for the usage and thus, the complexity of the system description is higher. We therefore propose a system of software development views that allows to incrementally add more information to the system models. The consecutive refinement of models is inspired by the goal-oriented approach introduced by [Lev00], in which

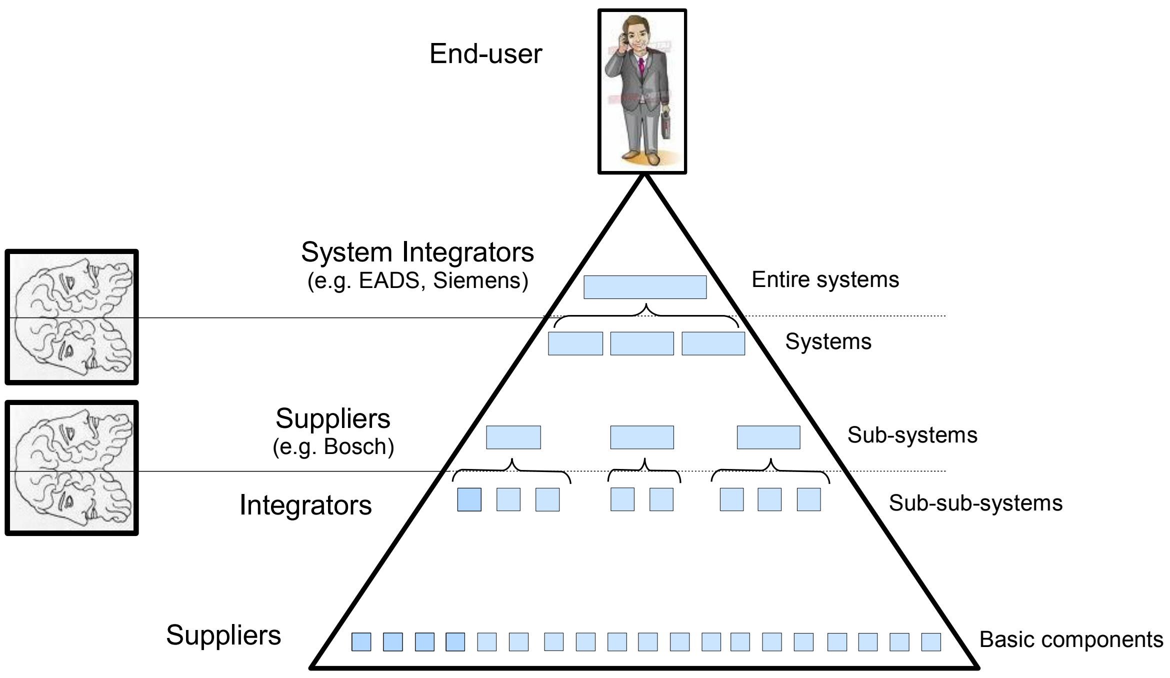

**Figure 4: Suppliers vs. Integrators**

the information at one level acts as the goals with respect to the model at the next lower level. Our software development views, presented in the next subsections, are

- the *functional view* (Section 3.1) that represents the decomposition of the system according to the behavior needed by its users (end users or system integrators),
- the *logical view* (Section 3.2) that represents the logical decomposition of the system and realization of the software architecture, and
- the *technical view* (Section 3.3) that represents the technical implementation of the system.

In order to enable the extensibility of models with additional information relevant for the realization of non-functional requirements (e. g., failure models) or even other development disciplines (e. g., mechanical information), we enable the use of decorators (Section 3.4).

## 3.1 Functional View

The functional view describes the usage functionality that a system offers its environment/users. Thereby, a user can be both, the end user of the system (at the highest level of granularity) or a system integrator (at a lower level of granularity). The functionality that is desired from the system represents the most abstract (less detailed) – but nevertheless, most important – information about the system. During design and implementation, we add realization details that are irrelevant for the usage of the system (but nevertheless essential for its realization).

**Aims** The central aims of the functional view are:

- Hierarchical structuring of the functionality from the point of view of the system's users;
- Definition of the boundary between the system functionality and its environment: definition of the syntactical interface and 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;
- Understanding of the functional interrelationships and mastering of feature interaction.

*Note:* The functionality of a system can be realized either in software or hardware. For example, a lot of functionality might be realized by mechanical or electrical systems that do not include software at all. In SPES-ZP1, however, we do not consider functionalities that are not implemented with software. Every system functionality that is outside of software is considered to belong to the environment of the system. For example, in SPES we do not address the development of airplanes whose functionality is implemented without software (e. g., hydraulic, electric), nor do we address the development of such (e. g., mechanic, hydraulic, electric) parts in the case when the airplanes functionality is also partly realized in software. The functionality outside of software belongs to the environment of the software system.

## 3.2 Logical View

The logical view describes how the functionality is realized by a network of interacting logical components that determines the logical architecture of the system. The design of the logical component architecture is driven by various considerations such as: achieving maximum reuse of already existent components, fulfilling different non-functional properties of the system, etc. The logical architecture bridges the gap between functional requirements and the technical implementation means. It acts as a pivot that represents a flexibility point in the implementation.

**Aims** The main aims of the logical view are:

- Describing the architecture of the system by partitioning the system into communicating logical components;
- Supporting the reuse of already existent components and designing the components such that to facilitate their reuse in the future;
- Definition of the total behavior of the system (as opposed to the partial behavior specifications in the functional view) and enabling the complete simulation of all desired functionalities;
- Mediation between the structure of the functional hierarchies and that of the already existing technical platform on which the functions should run.

Since the functions of the functional view are defined by given user requirements and the prerequisites of the technical layer are primarily given a priori, from a software development point of view, the main engineering activities are concentrated on the logical component architecture. Thereby, the logical architecture should be designed in order to capture the central domain abstractions and to support reuse. As a consequence, the logical architecture should be as insensitive as possible to changes of the desired user functionality or technical platform. It should be the artifact in the development process with the highest stability and with the highest potential of reuse.

### 3.3 Technical View

The technical view comprises the hardware topology on which the logical model is to be deployed as well as the resulting software running on the hardware. For us hardware means entities on which the software runs (ECUs), or that directly interact with the software (sensors/actors). On higher granularity levels hardware entities can also be abstractions/aggregations of such entities.

In the technical view engineers need to consider hardware related issues such as throughput of communication, bandwidth, timing properties, the location of different hardware parts, or the exact interaction of the software system with the environment.

**Aims** The main aims of the technical view are

- Describing the hardware topology on which the system will run including important characteristics of the hardware;
- Describing the actuators, sensors, and the HMI (human-machine interaction) that are used to interact with the environment;
- Implementation and verification of real-time properties;
- Ensuring that the behavior of the deployed system (i. e., the hardware and the software running on it) conforms to the specifications of the logical layer.

Note: One of the main advantages of the clear distinction between the logical and technical architecture is that it enables a flexible (re-)deployment of the logical components to a distributed network of ECUs. If the hardware platform changes, the logical components only need to be (re-)deployed, but the logical architecture does not need to be redesigned.

### 3.4 Core-models and their Decorations

Each of the three software development views is usually represented by a dominating *core-model* (e. g., functions hierarchies in the functional view, networks of components in the logical view) and may provide a number of additional *decorator-models*. Decorator-models are specialized for the description of distinct classes of functional and non-functional requirements that are relevant to their respective *software development views*. Decorator-models enrich the core-models with additional information that is necessary for later steps in the software development process or for the integration with other disciplines. For example this could be security analyses, scheduling generation or deployment optimizations. The complexity of decorator-models is arbitrary and their impact on the overall system functionality may be significant. Failure-models are an example of usually quite complex decorator-models to the models of the logical architecture. The impact of failure-models to the overall functionality is usually relatively critical, too. Other examples for existing decorations concerning the technical view are information concerning *physical*, *geometrical*, *mechanical* and *electrical* properties of the technical system under design.

*Note:* Inside their original engineering domains (e.g. mechanical engineering, electrical engineering) those decorations do represent the dominating core-models themselves. Here, the software development specific models may supply additional decorating information.

## 4 Relating the models

In the previous sections (Sections 2 and 3) we detailed two different manners to achieve abstraction: by using the whole-parts decomposition and by using different software development views. Thereby we obtain models (contained in each cell of the table in Figure 2) that describe the system either at different levels of granularity or from different software development points of view. In this section we discuss the relation between models in adjacent cells (horizontally from left to right, and vertically between two consecutive rows).

### 4.1 Horizontal Allocation (Mapping models at the same granularity level)

In general, there is a many-to-many (n:m) relation between functions and logical components that implement them, respectively between logical components and hardware on which they run. However, in order to keep the relations between models relatively simple, we require the allocation to be done in a many-to-one manner. Especially, we do not allow a function to be scattered over multiple logical components or a logical component to run on multiple hardware entities, respectively.

These links between views can be followed in either direction, reflecting either the means by which a function or logical component can be accomplished (a link to the more concrete view) or the goals a lower level component fulfills (a link to the more abstract view). So, the software development views can be traversed either from left to right or from right to left.

**Enabling a n:1 allocation.** In order to be allocatable, we require the functionalities to be fine granular enough to allow a many-to-one (n:1) mapping on logical components. In Figure 5a) we present the situation when the decomposition is insufficient since the relation between functions and logical components is many-to-many (there is a function that should be realized by more logical components). In order to enable a many-to-one allocation, the functional view (left) should be further decomposed in finer granular parts. Figure 5b) presents the situation when the decomposition of functionality advances until the moment when we can allocate each function on individual components. In a similar manner, the logical components should be fine granular enough to allow a many-to-one (n:1) deployment of the logical components on hardware units.

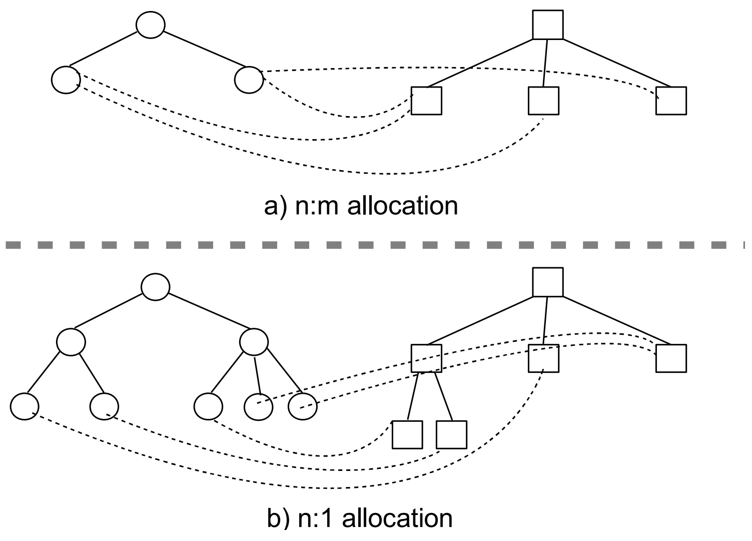

**Figure 5: Decomposition stages: a) insufficient; b) allocation/deployment possible**

Allocation represents decision points since each time a transition between an abstract to a more concrete view is done engineers have to decide for one of several possible allocations/deployments.

Note. It can happen (especially at coarse levels of granularity) that there is an isomorphism between the decompositions realized in different views (e. g., that the main sub-functions of a complex product are realized by dedicated logical components that are run on dedicated hardware). This is the ideal situation since the modularity is maximal. In these cases the allocation is trivial and realized by an isomorphic mapping.

### 4.2 Vertical Allocation (Transition from Systems to Sub-systems)

Vertical allocation means the top down transition from systems to sub-systems that happens typically at the border between suppliers and integrators – sub-systems built by suppliers are integrated into larger systems by integrators (see Figure 4).

In Figure 6 we illustrate the transition between two subsequent granularity levels generically named "system" and "sub-systems". The sub-systems of a system are determined by the structural decomposition in the logical or hardware views (see Section 2). The structure of the system defines its set of components and how they are composed. Each leaf component of the system structure determines a new system at the next granularity level. For example, in Figure 6 (top-right) the structural decomposition at the system level contains three components. Subsequently, at the next level of granularity we have three sub-systems as illustrated in Figure 6 and that correspond to the components – i. e., $Component_1 \mapsto Subsystem_1$, $Component_2 \mapsto Subsystem_2$, and $Component_3 \mapsto Subsystem_3$.

Generally, the functionality allocated to one of the components of the system defines the functional requirements for its corresponding sub-system. Each sub-system carries the user

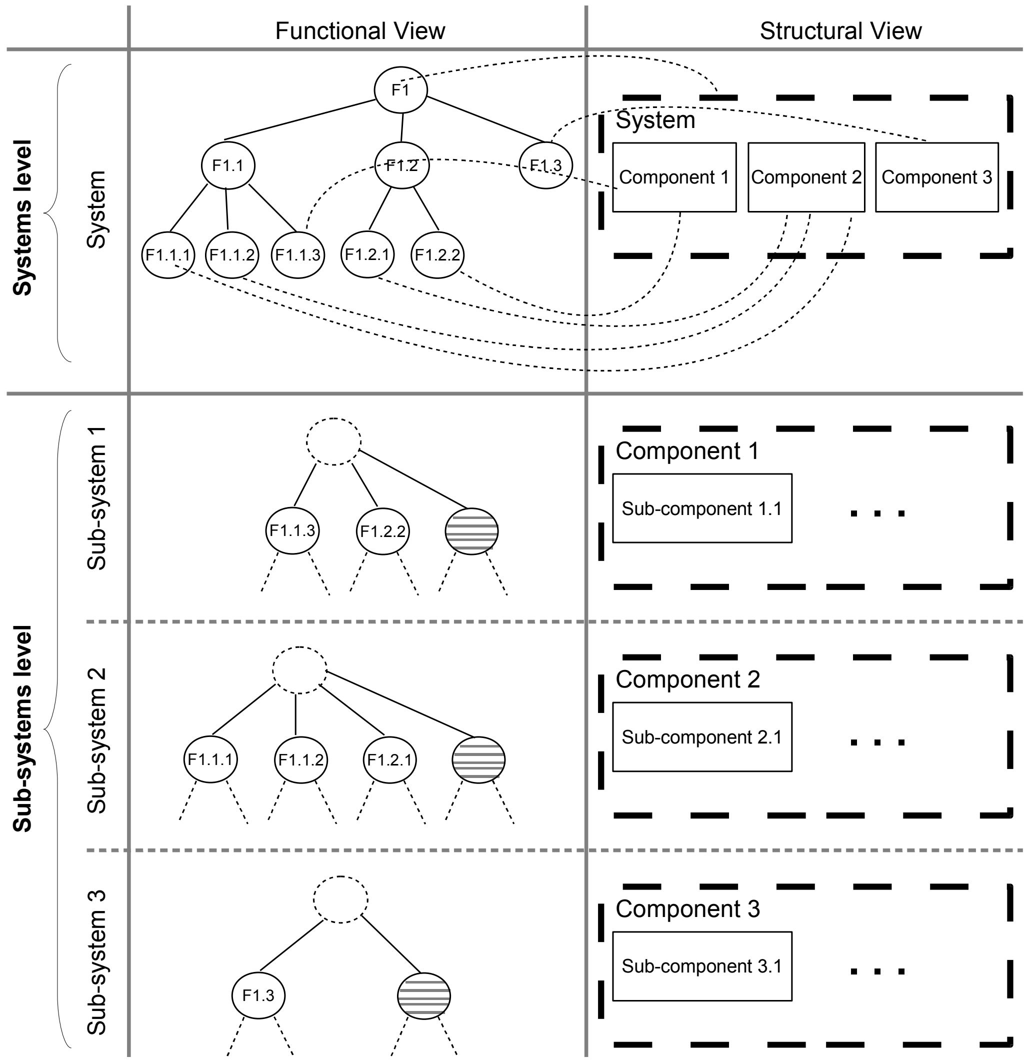

**Figure 6: Structural decomposition of the system defines the sub-systems situated at the subsequent level of granularity**

functionality allocated to its corresponding component at a higher granularity level. For example, in Figure 6 to the "Component 1" component was allocated the F1.1.3 and F1.2.2 functions. These functions should be implemented by the "Sub-system 1" that corresponds to "Component 1" on the next level of granularity. In addition to the original user functions allocated to components at the system level, at the sub-system level new functionality is required due to design (and implementation) decisions taken at the system level. This functionality is needed in order to allow the integration of the sub-systems into the systems and can be seen as a "glue" functionality. In Figure 6 we pictured the glue functionality by hashed circles. The root representing the entire functionality of a sub-component (not-existent at the system level) is pictured through a dotted circle.

## 5 Future Work

In this document we presented different abstraction layers at which the models used to realize a software product should be categorized. There are however many open issues that need to be investigated in the next deliverables:

1. **Methodology.** The steps that should be performed to instantiate the layers is of capital importance for the realization of the software product. Possible solutions can be top-down, bottom-up or even mixed.
2. **Allocation.** Based on which criteria is the allocation of functionalities on logical components and of logical components on technical platform made represent other open issues.
3. **Models.** Which modeling techniques would fit at best to describe particular aspects of the system (e. g., functionality) at particular granularity levels (e. g., entire system).

## References

[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. SPES 2020 Deliverable D1-1: Model-based Development - Motivation and Mission Statement of Work Package ZP-AP 1. Technical report, 2009.

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. SPES 2020 Deliverable D1.1.A-1: Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical report, Technische Universität München, 2009.

[Lev00] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Software Eng.*, 26(1):15–35, 2000.