A System for Seamless Abstraction Layers for Model-based Development of Embedded Software*

Judith Thyssen, Daniel Ratiu, Wolfgang Schwitzer, Alexander Harhurin, Martin Feilkas
Technische Universität München
Institut für Informatik, Lehrstuhl für Software und Systems Engineering
{thyssen,ratiu,schwitze,harhurin,feilkas}@in.tum.de
Eike Thaden
OFFIS - Institute for Information Technology
eike.thaden@offis.de

Abstract: 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. In this paper we present a conceptual framework to holistically classify developed models along different levels of abstraction. We do this by defining 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 models that describe different perspectives of the system or are at different granularity levels. However, we do not address the process to be followed for building these models.

1 Motivation

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 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. Model based development promises to provide effective solutions by the pervasive use of adequate models in all development phases as main development artifacts.

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

*This work was mainly funded by the German Federal Ministry of Education and Research (BMBF), grant SPES2020, 01IS08045A.

models and thereby to a lack of automation, to difficulties in tracing the origins of modeling decisions made in different stages, and to difficulties in performing global analyses or optimizations that transcend the boundaries of a single model.

Our aim is to enable a seamless integration of models into model chains. A first step to achieve this is to understand which kinds of models are built, who are their main stakeholders, and the relation between them. To this end, we introduce a conceptual framework to holistically classify models that describe the system.

Outline. In Section 2 we introduce our modeling framework. In the following sections we detail the framework along two dimensions: on the vertical dimensions model elements specified on different granularity levels are mapped on each other. The specifications of elements on the lower granularity level form a refinement of the specification of their counter parts on the higher granularity level. A special kind of mapping is the mapping of one element on the higher level to a set of elements on the lower level which is similar to the classical decomposition (Section 3). On the horizontal dimension different development perspectives are related to each other (Section 4). In Section 6 we tackle the issue of crossing the different abstraction layers. Section 7 presents related work and Section 8 concludes the paper giving an outlook on future work.

2 Achieving abstraction

Abstraction means the reduction of complexity by reducing details. There are two typical possibilities to reduce details:

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, less complex parts. These parts can be regarded as full-fledged systems themselves at a lower level of granularity. 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 as atomic.

Distinct development perspectives. 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 perspective. 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 1: one given by the *level of granularity* at which the system is regarded and the second one is given by different *software development perspectives* 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 3). 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

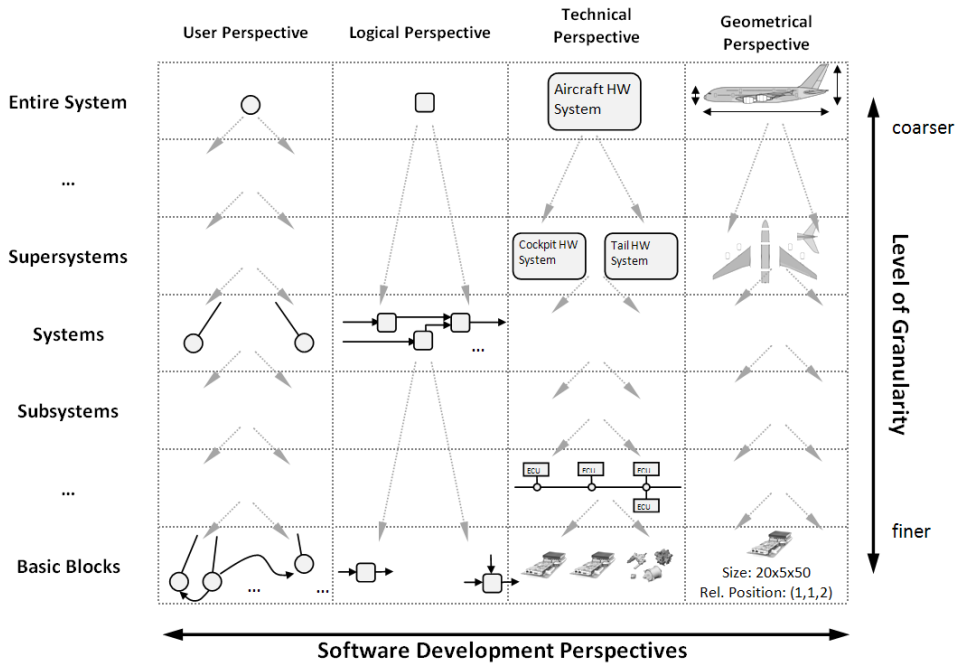

Figure 1: Two approaches to achieve abstraction: a) different granularity levels (vertical), and b) different software development perspectives (horizontal)

and subsequently the composition of the sub-systems into a whole are of central importance. As a consequence, we explicitly include the different granularity levels of systems in our framework (Figure 1 - vertical).

Software development perspectives. A system can be regarded from different perspectives, each perspective representing different kinds of information about the system. (Section 4). Our framework contains the following development perspectives: the user perspective, the logical (structural) perspective, the technical perspective, and the geometrical perspective. The perspectives aim to reduce the complexity of the software development by supporting a stepwise refinement of information from usage functionality to the realization on hardware: early models are used to capture the usage functionality and are step-by-step enriched with design and implementation information (Figure 1 - horizontal).

3 Granularity Levels

In order to cope with the complexity of today's systems, we decompose them into subsystems. 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. These granularity levels enable 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 granularity levels allow us to reduce the overall complexity following the "divide and conquer" principle.

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 user, or the logical, or the technical architecture (these software development perspectives are presented in Section 4 in detail). Depending on which of these software development perspectives is prioritized, the resulting system decomposition looks different: the more emphasis is put on decomposing based on one of the software development perspectives, the more the modularization concerning the other software development perspectives is neglected and other aspects of modularity are lost (phenomenon known as "tyranny of dominant decomposition").

On the one hand, for models situated at coarser granularity levels, the decomposition is many times driven by established industry-wide domain architectures. These 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). On the other hand, for the models situated at finer granular layers, the decomposition 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 user and technical perspective (see Section 4), 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 Figure 2): the end users are interested only in the top-level functionality of the entire system, the system integrators 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.

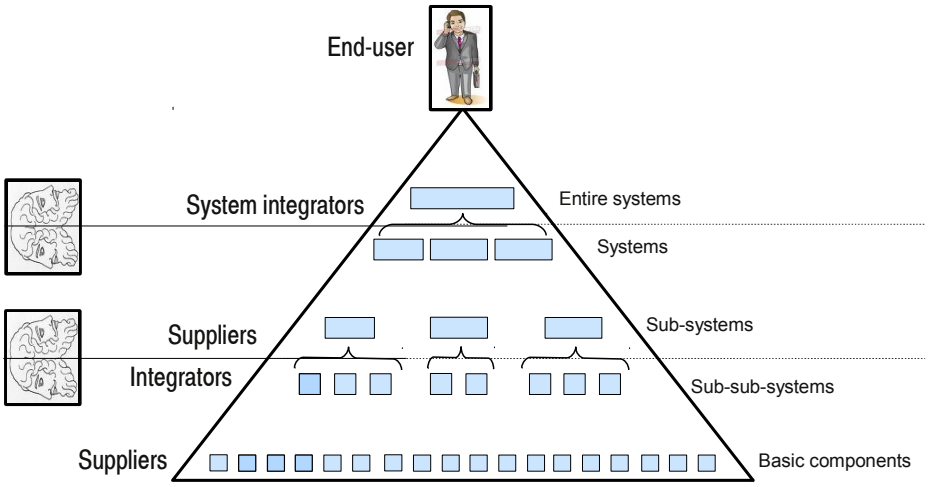

Figure 2: Suppliers vs. Integrators

4 Software Development Perspectives

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. 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 perspectives that allows to incrementally add more information to the system models. This refinement of models is inspired by the goal-oriented approach introduced by [Lev00], in which the information at one level acts as the goals with respect to the model at the next level. Our software development perspectives are: the *user perspective* that represents the decomposition of the system according to the behavior needed by its users, the *logical perspective* that represents the logical decomposition of the system and the realization of the software architecture, and the the *technical perspective* that represents the technical implementation of the system. Additionally, the *geometrical perspective* represents the geometrical (physical) layout of the system, e. g. information about the shape of an hardware element or its concrete position in an airplane. From a systems engineering point of view, the geometrical perspective is essential. In this paper, however, we concentrate on the development of the software of the future systems. Thus, the geometrical perspective is not further detailed in the following.

User perspective. The user perspective 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 – 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).

The central aims of the user perspective 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.

Logical perspective. The logical perspective 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.

The main aims of the logical perspective 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; Optionally: Definition of the total behavior of the system (as opposed to the partial specifications in the user perspective) and enabling the complete simulation of all desired functionalities; Mediation between the structure of the function hierarchy and that of the already existing technical platform on which the system should run.

Since the functions of the user perspective 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.

Technical perspective. The technical perspective comprises the hardware topology on which the logical model is to be deployed. 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 perspective 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.

The main aims of the technical perspective 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 in combination with a deployment of logical components; 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.

5 Core-models and their Decorations (Viewpoints)

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 (also called Viewpoints).

Each of the three software development perspectives is usually represented by a dominating $core\text{-}model$ (e. g. functions hierarchies in the user perspective, networks of components in the logical perspective) and may provide a number of additional $decorator\text{-}models$. Decorator-models are specialized for the description of distinct classes of functional and non-functional requirements that are relevant to their respective software development perspectives. 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 safety 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 perspective are information concerning physical, mechanical or electrical properties of the technical system under design.

6 Relating the Models

In the previous sections (Sections 3 and 4) we detailed two different manners to achieve abstraction: by providing mappings between elements on different granularity levels and by using different software development perspectives. Thereby we obtain models (contained in each cell of the table in Figure 1) that describe the system either at different granularity levels 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).

Horizontal allocation (mapping models at the same granularity level). In general, there is a many-to-many (n:m) relation between functions (user perspective) and logical components (logical perspective) 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. If necessary, the user perspective should be further decomposed in finer granular parts until an allocation of each function on individual components is possible. In a similar manner, the logical components should be fine granular enough to allow a many-to-one (n:1) deployment on hardware units.

Allocation represents decision points since each time a transition between an abstract to a more concrete perspective 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 perspectives (e. g. that the main sub-functions of a complex product are realized by dedicated logical components that are run on dedicated hardware).

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 2).

In Figure 3 we illustrate the transition between two subsequent granularity levels generically named “system” and “sub-systems”. The sub-systems of a system can be determined by the structural decomposition in the logical or hardware perspectives (see Section 3). More complex mappings between elements on different granularity levels are possible, too, but are not further considered in this paper. 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 3 (top-right) the structural decomposition at the system level contains three components. Subsequently, at the next level of granularity we have three sub-systems that correspond to the components.

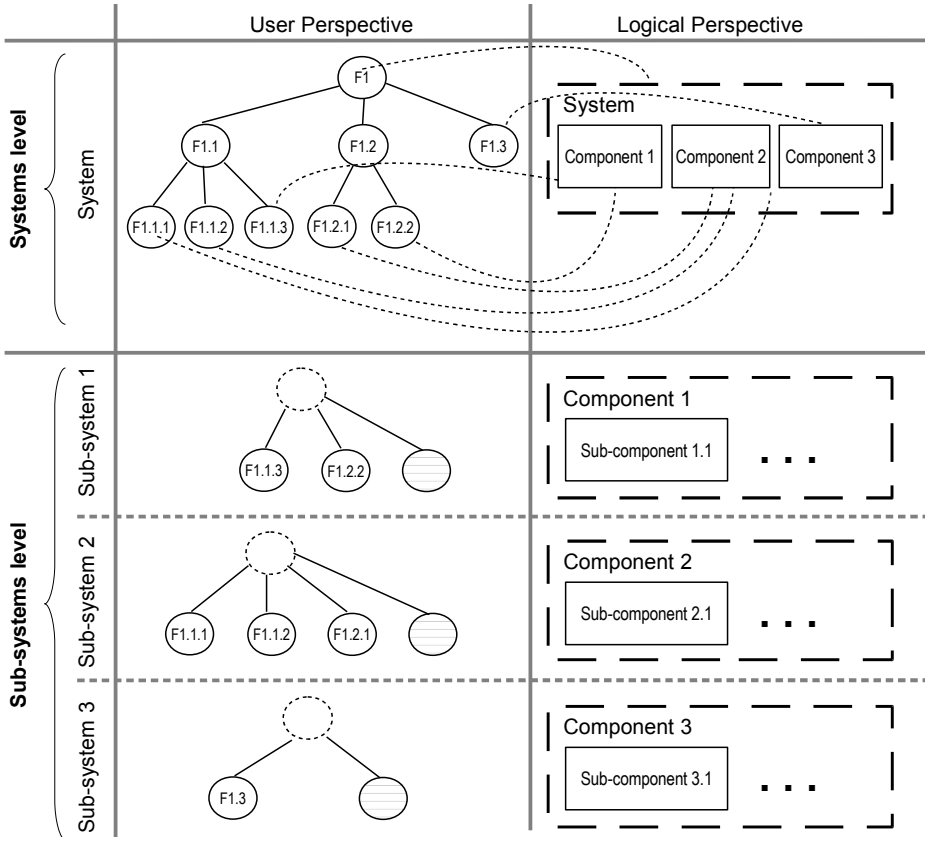

Figure 3: Structural decomposition of the system is one way to define the sub-systems situated at the subsequent granularity level

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 functionality allocated to its corresponding component at a higher granularity level. For example, in Figure 3 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 3 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.

7 Related Work

Our approach to reduce complexity by a systematic software development along different abstraction layers is part of a more general area of research about the pervasive and disciplined use of models along the development process.

MDA [MM03] represents a well known approach to master the complexity of todays systems by describing the system on different levels of abstraction starting with an an informal description of the system (known as "Computation Independent Model"). Based on this, 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 can run on a specific platform. While we are aiming at a modeling framework for the development of embedded systems or even specific domains, the MDA is a general purpose approach. The different development perspectives of our modeling framework can be seen as instantiation of the MDA-layers. Furthermore, the MDA layers do not address issues related to the vertical decomposition of systems into sub-systems down to basic components.

The EAST ADL (Electronics Architecture and Software Technology – Architecture Definition Language, [ITE08]) has 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. In contrast to our approach, the EAST ADL does not clearly distinguish the different dimension of abstraction, namely the decomposition in sub-systems and the different perspectives on a system. However, the EAST ADL can be seen as instantiation of our modeling framework. 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 perspective, the Functional Design Architecture vaguely corresponds to the logical perspective 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 perspective.

The idea to describe a system from different perspectives is not new. For example, the 4+1 View Model by Kruchten [Kru95] provides four different views of a system: the logical, process, development and physical views. The views are not fully orthogonal or independent – elements of one view are connected to elements in other views. The elements in the four views work together seamlessly by the use of a set of important scenarios. This fifth view is in some sense an abstraction of the most important requirements. However, since this approach is not based on a proper theory and sufficient formalization, a deeper benefit is not achieved. As a result, possibilities to do analysis with these views are weak, and the models are applied only at particular stages of the development process without a formal connection between them. In contrast, the introduced approach aims at "theoretical foundations of a strictly model based development in terms of an integrated, homogeneous, but yet modular construction kit for models" [BR07].

In [BFG+08], a first step has been made to integrate the existing research results into an

integrated architectural model for the development of embedded software systems. The architectural model comprises three subsequent layers, namely the service layer, the logical layer, and the technical layer. This architectural model served as basis for the modeling framework presented here. The different layers are reflected by the different software development perspectives of our modeling framework. In the current work we extended the architectural model by introducing granularity levels as a second dimensions.

8 Future Work

In this paper 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 are subject to current and future work: 1) *methodology* – the steps that should be performed to instantiate the layers is of capital importance for the realization of the software product; 2) *allocation* – based on which criteria is the allocation of functionalities on logical components and of logical components on technical platform made; 3) *models* – define which modeling techniques would fit at best to describe particular aspects of the system (e. g. functionality) at particular granularity level (e. g. entire system).

Acknowledgements: Early ideas of this paper originate from a discussion with Carsten Strobel, Alex Metzner, and Ernst Sikora.

References

[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.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007.

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

[Kru95] Philippe Kruchten. Architecture Blueprints - the "4+1" View Model of Software Architecture. In *TRI-Ada Tutorials*, pages 540–555, 1995.

[Lev00] Nancy G. Leveson. Intent Specifications: An Approach to Building Human-Centered Specifications. *IEEE Trans. Software Eng.*, 26(1):15–35, 2000.

[MM03] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.