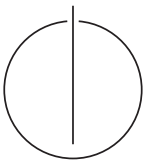
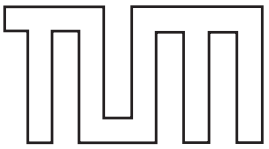


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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) Wolfgang Schwitzer (TUM) Judith Thyssen (TUM)
Diskussionsbeiträge	Alexander Metzner (OFFIS) Eike Thaden (OFFIS) Carsten Strobel (EADS-IW) Ernst Sikora (UniDuE) Kim Lauenroth (UniDuE) 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.

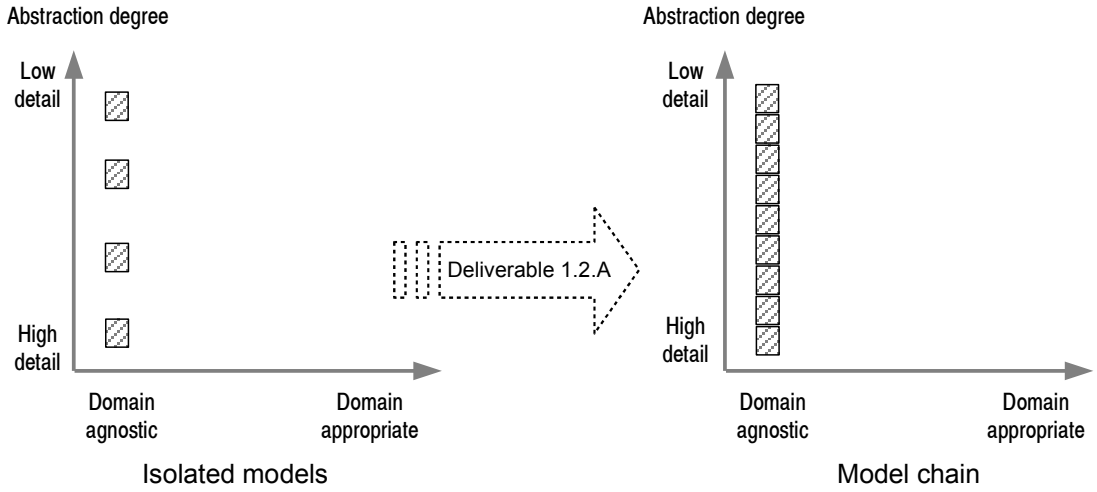


Figure 1: The scope of this deliverable

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

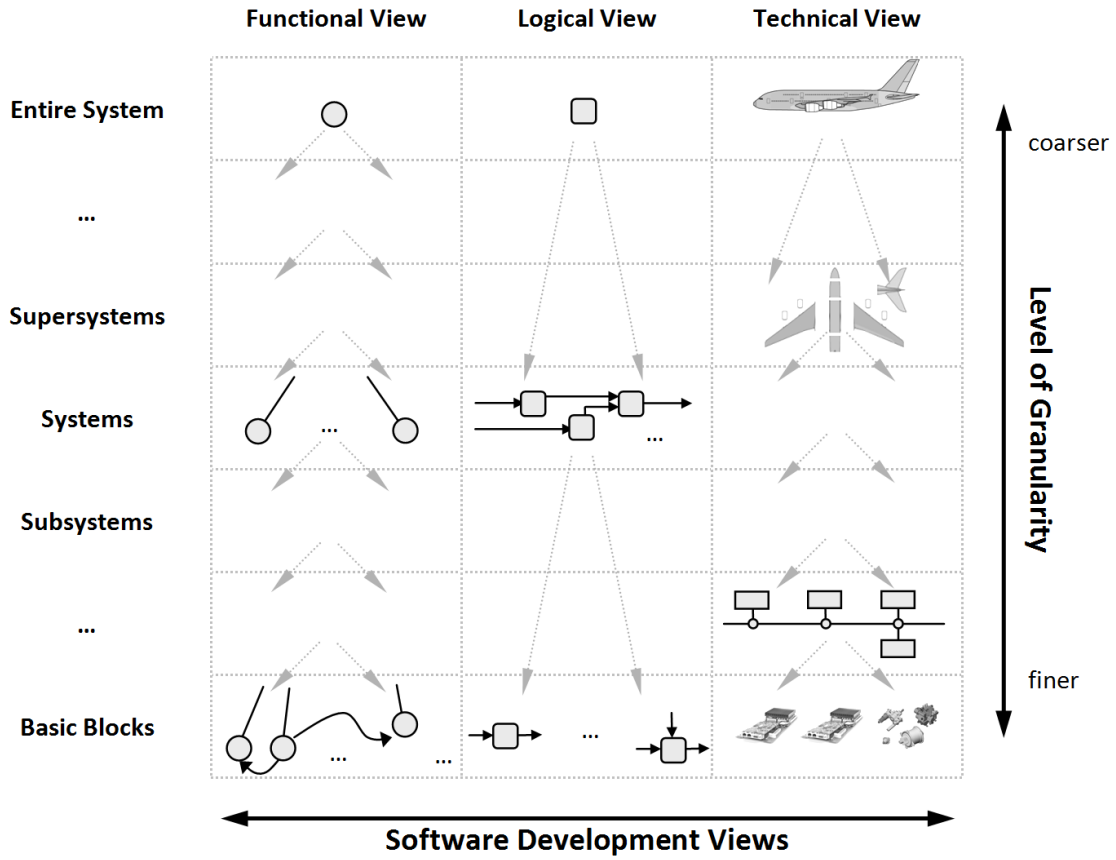


Figure 2: Two approaches to achieve abstraction: a) different levels of granularity (vertical), and b) different software development views (horizontal)

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).

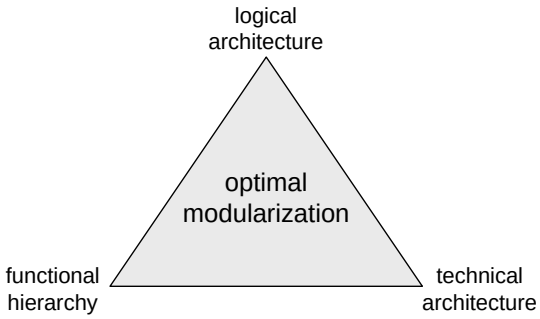


Figure 3: Contradictory factors of influences for the system decomposition.

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

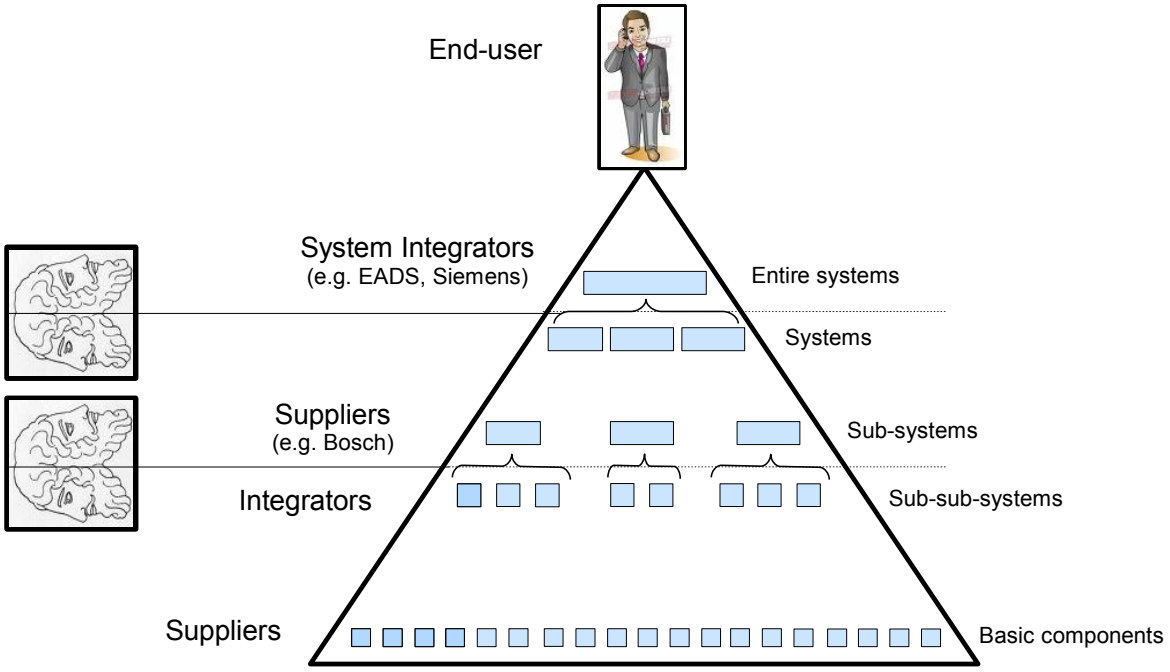


Figure 4: Suppliers vs. Integrators

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.

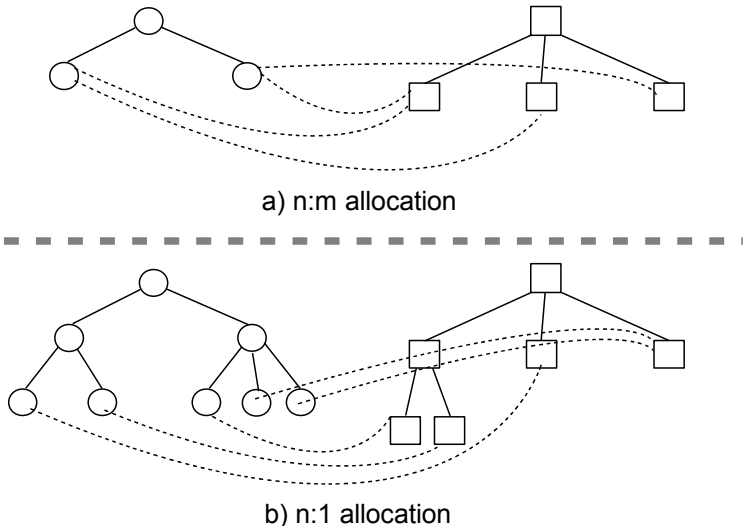


Figure 5: Decomposition stages: a) insufficient; b) allocation/deployment possible

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

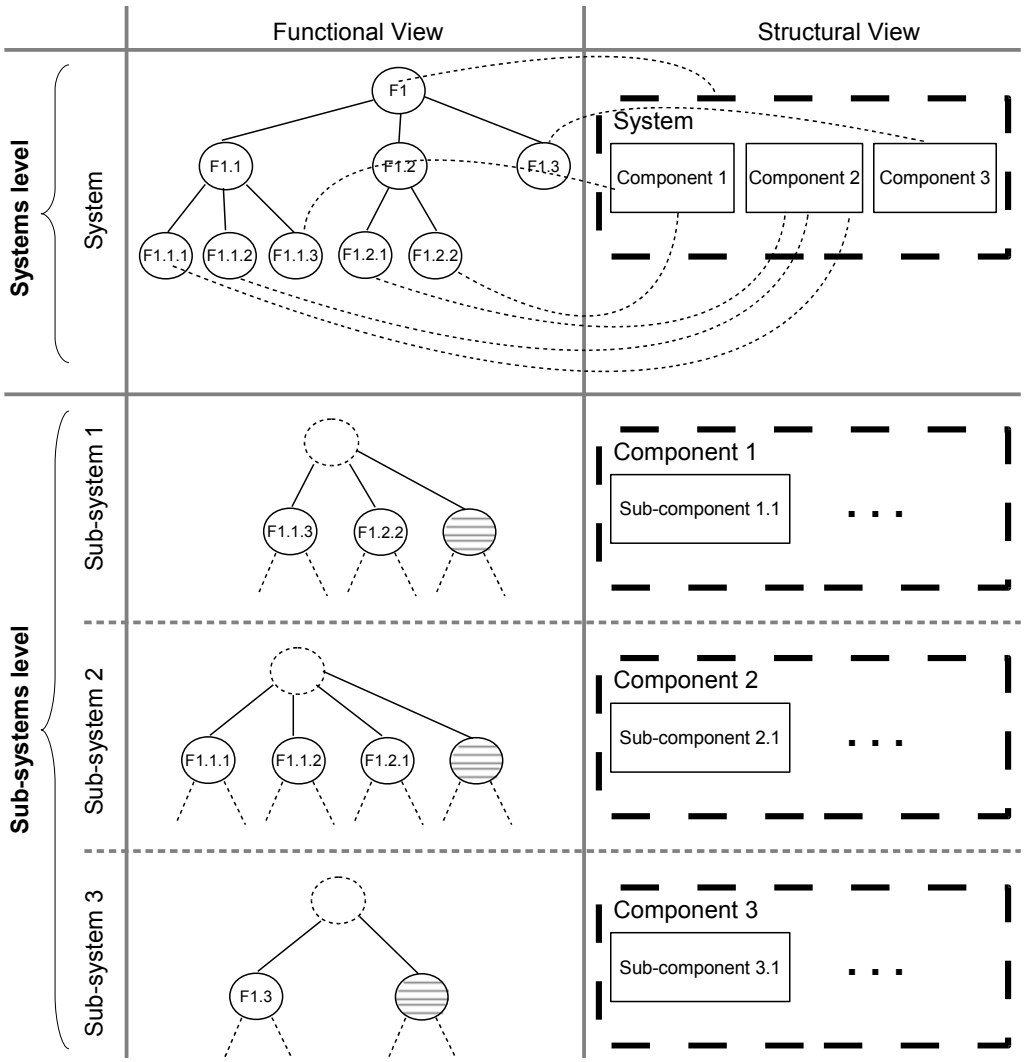


Figure 6: Structural decomposition of the system defines the sub-systems situated at the subsequent level of granularity

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.