2 Model-Based Integration

Holger Giese[1], Stefan Neumann[1], Oliver Niggemann[2], and Bernhard Schätz[3]

[1] Hasso Plattner Institute at the University of Potsdam, Germany
{holger.giese,stefan.neumann}@hpi.uni-potsdam.de
[2] Fraunhofer IOSB - Competence Center Industrial Automation, Lemgo, Germany
oliver.niggemann@iitb.fraunhofer.de
[3] fortiss GmbH, München, Germany
schaetz@fortiss.org

Abstract. The integration of different development activities and artifacts into a single coherent system is a major challenge for the development of complex embedded real-time systems. For complex software the functional integration alone is a major undertaking, in the case of embedded real-time systems we in addition have to cope with all the affected system characteristics such as real-time behavior, resource consumption, and behavior in the case of failures.

In this chapter we will discuss the state-of-the-art of model-based integration. Therefore, we will clarify the terminology concerning integration, provide a classification of the integration challenges for complex embedded real-time systems, and outline the fundamental techniques employed to cope with the integration challenges. This framework is then used to explain the current standard practice concerning integration of hardware and software for functional development as well as function integration. Furthermore, a number of advanced proposal how to address some of the remaining integration challenges such as AUTOSAR and MECHATRONIC UML using model-based concepts are presented using the framework.

2.1 Introduction

One of the major challenges for the development of complex embedded real-time systems is the integration of different development artifacts into a single coherent system. The integration problems we have to face are exacerbated even further as today's advanced embedded real-time systems tend to contain more functionality than in former times, are often expected to exhibit adaptive behavior and take advantage of wireless or local networking, or even have to be classified as system of systems rather than systems alone.[1]

For complex software systems integrating the functional aspects alone is often already a major challenge. However, in the case of embedded real-time systems we usually cannot restrict our attention to an abstract view on the software only. In addition we have to cope with real-time behavior, resource consumption, and behavior in the case of failures. All these system characteristics have

[1] See [1] for a discussion of the resulting integration efforts.

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 17–54, 2010.
© Springer-Verlag Berlin Heidelberg 2010

to be covered to ensure a proper integration. Therefore, the integration of complex embedded real-time systems has not only to cover the integration of highly complex functionality, but also to take care of additional relevant system characteristics such as hard real-time constraints, a proper use of resources, and dependable operation with respect to severe reliability, availability and safety requirements. Usually, it is not sufficient to only consider the software. Effects of the hardware and the run-time environment as well as low-level design and implementation decisions may be relevant.

When developing complex systems besides the technical aspects also process issues as well as organization aspects become relevant (cf. [2]). However, we will further restrict our discussion to the technical integration of development artifacts looking in particular at model-based development and ignore process and organizational issues of integration. In [3] several approaches are characterized which support the model-based development of embedded control systems and cope with several integration problems. Selected domain-specific modeling languages (like AADL [4]) and tool sets (like Fujaba [5]) are characterized concerning the integration problems which are solved by the particular approach. In [6] several effects are discussed which result from the partitioning of a system in subsystems or features, the separate development and the later combination and integration. In this work different types of integration properties are discussed, as well as what kinds of effects exist concerning these properties and how to cope with them. In [6] the focus is set to shared resources, communication features and interacting control concerning the partitioning and later composition of the overall system. This chapter in contrast focuses on the underlying problem and integration concepts and provides a characterization of existing integration problems and fundamental integration techniques. It presents their role by discussing the standard approach to hardware integration and function integration as well as several advanced model-based integration approaches.

In essence, the integration problem we have to face is related to the fundamental fact that we as humans somehow have to divide the problem into less complex elements that can be handled independently or with limited dependencies. While from a system engineering perspective [2] the technical integration problem can be restricted to the problem of integrating complete components, for software and embedded real-time systems the integration can also happen at the level of abstract software components or even abstract software or hardware models. Therefore, integration is usually related to some form of combining artifacts, which is often proceeded by some form of separation that divides the development task into the later combined artifacts.

This need for integration also at the level of software and hardware components rather than complete systems has several reasons. First the reuse of already developed artifacts either as developed or including some adaption in order to reduce costs is one major driver. In addition, the complexity of today's embedded real-time systems often makes it necessary to separate the development of subsystems within a single organization or across a network of suppliers, sub-suppliers and the manufacturer (in the case of the automotive domain called

OEM). While the following discussed aspects and characteristics concerning the integration are fundamental for several types of embedded systems the discussed examples focus on the automotive domain. However, the assumptions and conclusions made are also valid for the other domains. An example for the latter case is the automotive domain, where the traditional model of division of labor is that the OEM integrates complete subsystems including software and hardware developed by their suppliers. However, today this traditional model of division of labor where only complete subsystems combining software and hardware are integrated is no longer valid and therefore also integration scenarios where different software has to be integrated on the same hardware are relevant.

In this chapter we will present a framework to discuss how model-based integration can in particular facilitate the outlined problem of the technical integration of development artifacts. Therefore, we will first clarify the terminology concerning integration, provide a classification of the integration challenges for complex embedded real-time systems, and review the main techniques employed to cope with the integration challenge in Section 2.2. Then, this framework will be employed in Section 2.3 to explain the current standard practice concerning integration of hardware and software for functional development as well as function integration. A number of advanced proposals for how to address some of the remaining integration challenges such as AUTOSAR and Mechatronic UML using model-based concepts are discussed also by means of the framework in Section 2.4. In this section we also review other approaches and discuss which integration problems of our classification are covered and which fundamental techniques are employed. Finally, we discuss which challenges have been addressed and which open problems remain and provide our final conclusions and outlook on expected further research in Section 2.5 and close the chapter with some final remarks.

2.2 Integration

No uniform definition for the term integration can be found in the literature (cf. [2]). Therefore, we will at first outline the terminology employed in this chapter, a classification for the integration challenges and the fundamental techniques employed to support integration. These elements provide a conceptual framework that is then used throughout the rest of the chapter to explain how different approaches address integration.

2.2.1 Terminology

The counterpart of the integration is the division or decomposition of a system into subsystems. At the more abstract level discussed here it relates to the division of development artifacts and not necessarily complete subsystems. Note that the division can happen explicitly when a development artifact is used to plan and document the decomposition or implicitly when either ad hoc or traditionally certain parts of a solution are developed separately. It is also worth

mentioning that any such decomposition usually goes hand in hand with a breakdown and refinement of requirements.

The division used to derive development artifacts of reduced complexity consequently leads to the need to integrate the results of the separate tasks to derive the originally intended complete development artifact at the required level of detail. The integration itself therefore consists of the activities required to achieve a proper composition when combining development artifacts.[2]

Therefore, consequently, integration activities can be found during the decomposition of a system into subsystems as well as during its composition from the subsystems, and when related subsystems are developed in parallel. This can include preventive activities such as the definition of abstract interfaces or analytic activities such as integration testing which check system requirements that could not be addressed at the subsystem level. In addition, all activities to solve problems during the composition are included; this includes problems encountered during the system division or during the parallel development.

Integration essentially happens when in a development task multiple development artifacts serve as input and the specific combination of versions and variants of these development artifacts has not been considered beforehand. Many of these development artifacts are in fact models that represent some abstract view on specific aspects of the envisioned or already existing version and /or variant of a system or subsystems. Therefore, also in the case of classical integration usually models are already of paramount importance. If these models are furthermore not only paper-and-pencil models, but are employed to derive other development artifacts or facilitate specific integration steps, we consider this to be one form of model-based integration.

Given a development task and a fixed level of detail, the decomposition step results in a number of components with reduced dependencies. At a coarse grain level the resulting structure is referred to as architecture which decomposes the problem into components connected via links. The overall architectures capture the different components and their structure in the form of links between them.

For our discussion it is useful to further distinguish two cases of architectural decomposition: In the first case of a hierarchical architecture the components are modules at the same conceptual level (e.g., an architecture with separate components for motor and steering control) while in a layered architecture the decomposition provides layers where one layer is operating on top of the underlying one (e.g., application, operating system, hardware). When developing using such layers sometimes the models abstract from the layer beneath by means of modeling concepts and thus no explicit initial decomposition is visible.

Taking the many different facets of composition into account, we can define *integration* as the development activities which are employed to ensure a proper composition when multiple conceptually development aspects are combined which usually results in new or changed development artifacts.

[2] We do not consider the problem here that the different artifacts have been defined using different models of communication, computation, or causality as discussed in Chapter 1.

The composition of different software modules into a larger module is an example of combined conceptual development aspects being used as concrete development artifacts. Here decomposition as well as composition is done explicitly.

On the other hand, the combination of a software module with its underlying hardware relies on an implicit decomposition from the underlying layer. Afterwards, aspects are composed implicitly when enriching the model to include more information of that layer. This enrichment happens usually in several steps where additional conceptual development aspects such as limited precision, execution times or memory consumption are integrated with development artifacts that beforehand were not included in these aspects (by initially abstracting from them).

2.2.2 Classification of Integration Problems

Several problems are encountered when integrating different development artifacts. However, most of the relevant kind of conflicts can be covered by the classification of integration problems depicted in Figure 2.1.

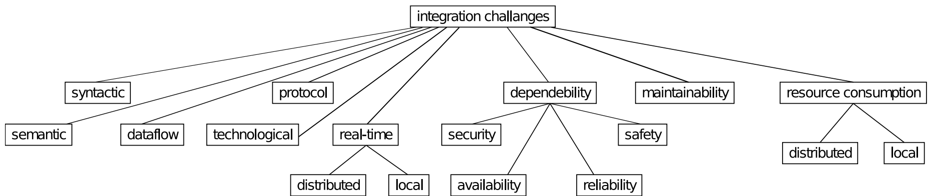


Fig. 2.1. Classification of Integration Problems

The outlined classification includes technological aspects refering to the ability to integrate at all on the basis of some technology and the syntactical aspect covering that the parts have to be integrated concerning the exchanged or shared data as well as the offered or required operations. Furthermore, we have the full semantic, which ensures that the encoding of data and side effects of the offered operations are compatible, the protocol, which addresses issues such as non-uniform service availability the synchronization and control flow between different parts, and dataflow, which covers the specific needs for composing dataflow computations as required for most control algorithms, are included.

While technological and syntactical integration are often addressed upfront during decomposition (e.g., AUTOSAR in Section 2.4.1), more demanding integration problems such as semantic, protocol or dataflow are today often only handled when it comes to composition.

The dependability resp. quality-of-service aspects include that the composition has to fulfill given reliability goals, availability requirements, exclude unacceptable safety or security problems, and still ensures maintainability. The real-time behavior may include the local behavior but also distributed real-time

compatibility. For the resource consumption, either node limitations such as CPU time, network limitations such as bandwidth or even system limitations such as the overall power consumption may be relevant.

Embedded systems differ from standard software systems in particular when it comes to integration problems related to real-time or resource requirements. As real-time requirements, severe resource constraints, and more demanding dependability requirements have to be fulfilled by an integrated system with often rather limited hardware, a large fraction of the development efforts have to be spent on addressing them explicitly. In standard software development, developers use resources rather excessively and remain at a higher level of abstraction, and thus avoiding having to optimize their solution for a specific hardware. In the embedded world in contrast this luxury is not possible and a sufficient behavior has to be achieved with rather constraint resources and thus all kinds of hardware-related constraints such as execution times and resource consumption have to be addressed explicitly.

2.2.3 Fundamental Integration Techniques

To exemplify the rather general considerations presented so far, we will now look into a number of fundamental techniques for the proper integration. As the steps undertaken to divide the labor and the effort for the final integration of the independently achieved results can only be understood in combination, we will introduce the basic principles for both aspects at once and then discuss their interplay.

Explicit Horizontal Decomposition & Composition

The first fundamental concept operates at a constant level of abstraction and looks into an explicit decomposition into subsystems at the same level (horizontal decomposition). In order to achieve a suitable decoupling of the separately considered parts some form of separation of concerns [7] as well as information hiding [8] are usually employed, too. For example, architectural aspects (concerns) covered by modules or components which provide interfaces. Using an interface allows to decompose the architecture and hide implementation details (information hiding) at the same horizontal abstraction level.[3]

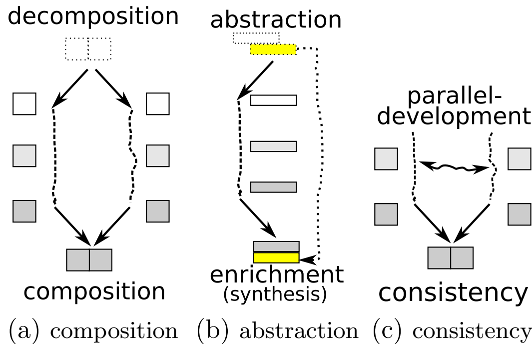
This form of horizontal decomposition of the system usually permits the subsystems to be developed in parallel and work on disjoint sets of development artifacts. In addition to the description of the system, this decomposition also happens for the requirements which are broken down from the system into its subsystems.

The explicit composition brings together subsystems which have been developed in parallel. In the ideal case all relevant system or subsystem characteristics are captured during the decomposition and are guaranteed when doing the composition. However, often this is not the case. For example, when using separation

[3] Some techniques (e.g., information hiding) also support vertical abstraction like described later.

of concerns several aspects are often not covered during decomposition but become relevant when doing the composition (potentially in a later development stage) or when the composition not only exhibits the characteristics of its components but also characteristics which are determined by the composition (sometimes call emergent) itself. It is particularly relevant for the integration that all system requirements that have not been broken down into subsystems requirements are checked for the composition result. This includes that characteristics such as deadlocks which can often not be predicted when doing the decomposition have to be addressed when doing the composition. Therefore, depending on the question of which characteristics are compositional or not resp. which requirements have been broken down to local properties of the subsystems more or fewer characteristics of the composition have to be checked at composition time to ensure a proper integration.

The standard case for composition is that the individual constituent parts are simply combined by some generic form of composition (e.g., scheduling in the case of processes on an operating system). More advanced cases employ declarative constraints contained in the specification of the components to ensure that the composition behaves properly (e.g., scheduling with guaranteed deadline in case of processes on a real-time operating system).



(a) composition (b) abstraction (c) consistency

Fig. 2.2. Fundamental techniques employed to approach integration

The resulting interplay of decomposition and composition is depicted in Figure 2.2 (a). At a rather high level of abstract the system is decomposed into two or more subsystems that are developed in parallel. These subsystems, which are then further elaborated in parallel, are composed later on according to the decomposition done upfront.

It has to be noted that in contrast to more restricted interpretations of the term integration (cf. [9]) we do not limit integration to the case where checks at composition time are employed, but also include the case where upfront activities such as defining interfaces ensure a proper composition.

Module interfaces are the classical example for the decomposition / composition. If done proberly as in many programming languages, the definition of the

Table 2.1. Coverage of integration aspects by modules with syntactical interfaces

Integration problems	Technique	Explanation
syntactical	D/(S)	Is checked already during decomposition guarantee proper integration during composition. Synthesize syntactical correct implementation of interfaces.
technological	S	try to solve problem using synthesis, e.g., generating compatible C-Code for the implementation of interfaces.
semantically	C	Is checked only late during composition.

(legend: D = during decomposition – C = during composition – S = by synthesis).

syntactical interfaces at the time of the decomposition guarantees that during the later composition no syntactical integration problem can result. In Table 2.1 this result is depicted using a classification that focuses on the question when the integration problem becomes visible. The case *during decomposition* (D) implies that the integration is guaranteed upfront, while the *during composition* case (C) results in a risk that an integration problem is detected rather late. The case *by synthesis* (S) refers to automated techniques that can generate a solution that solves the integration problem in principle. However, usually no guarantee can be given that the synthesis will thus be able to find a solution as there might be not resolvable conflicts.

Vertical Abstraction & Enrichment

Another fundamental concept to separate details during the development of a system is vertical abstraction and enrichment. Compared to the previously described horizontal decomposition and composition, where the abstraction level is the same when doing the composition or decomposition, vertical abstraction and enrichment change the abstraction level itself.

In the abstraction step we omit details of the envisioned system and focus on the characteristics that are relevant for the current development step. The abstraction can be seen as a form of implicit separation by omitting the details for a certain time. These omitted details are then later added to the development artifacts when enriching them.

The abstraction concept can be employed to ease development when there is only a unidirectional dependency between the upfront-addressed details and the omitted ones. Often architectural layers are employed to realize the abstraction independent of the concrete omitted details (application, operating system, hardware). The fact that the lower architectural layers do not depend on the higher layers in combination with standardized interface for the lower-level layers allows that the higher layers could be developed more or less independent, and omitted details are either filled in by the lower levels or can be considered later on.

The design characteristics that are affected by the combination are then usually considered later. E.g., first the end-to-end timing constraints are considered and the specific timing resulting from the integration with the operating system and the hardware layer is considered later. As the real-time scheduling can only be considered in combination with the concrete real-time operating system, the analysis of the real-time characteristics are only addressed when both

are integrated. When abstraction is used in this manner, dependencies between the separated layers often seriously constrain the decoupling of the development activities for the subsystems. The development artifacts of the different activities depend on each other such that the ordering of the development activities reflects the usage of development artifacts of other activities. E.g., details are stepwise added to the development artifacts during function development (see Section 2.3.1) where in each step the effects of other layers (runtime system, hardware, ...) is added.

Vertical enrichment is the counterpart of abstraction where new characteristics are added to a development artifact. We have to further distinguish two fundamentally different forms of how enrichment can occur. Either the detail adds new characteristics to the development artifact. In this case no constraint between the abstract development artifact and the added detail exists (e.g., a logical untimed model is enriched by timing information). On the other hand the development artifact may already contain some abstract information about some system characteristic which constraints the added details (e.g., an idealized model equal to some set of differential equations has to be developed into a model equal to a set of difference equation scheme). Often the development starts with such an abstract artifact and abstraction is used rather than explicitly applied. In this case you either want to have refinement when the more detailed structure and behavior is included in the more abstract one or approximation such that the more detailed structure and behavior is similar to the idealized abstract one.

In the case of refinement the abstraction already includes the refined behavior as one possibility and thus checking crucial properties for the abstraction can guarantee these properties also for the refinement. In contrast, for an approximation it holds that the abstraction is an idealization and that the behavior which could be observed for the enrichment should be somehow similar. Therefor, here the opposite observation holds that only in the case a required property does not hold for the approximation it can also not be expected to hold for any enrichment (even though this is not necessarily always the case). Thus, refinement can be used to guarantee the absence of failures upfront, while approximation can be employed to detect possible integration failures upfront. A fully inconsistent enrichment which is neither a refinement nor an approximation would mean to redo all the construction work contained in the abstract development artifact and thus is usually not intended. However, it must be mentioned that different characteristics of a development artifact may be enriched differently and thus some may be refined while others are approximated.

In both cases the enrichment is some form of implicit (vertical) composition as a new or yet only insufficient covered development aspect is now considered. The initially not considered development aspect is then brought back into the picture and thus the related information about the related layer beneath is implicitly composed with the model that beforehand used abstraction to omit that information. An example for such a case is the usage of abstraction layers, e.g., hiding communication details or hardware properties like in case of the different layers of the AUTOSAR architecture (see Section 2.4.1).

Like in the case of (horizontal) composition again synthesis can be used to automatically apply enrichment. Depending on the applied form of enrichment respectively the previously applied abstraction, synthesis does not guarantee in any case that all desired properties are fulfilled (e.g., no schedule for a set of tasks can be synthesized). In many cases enrichment is automatically applied using synthesis, e.g., in the case of automatic C-code-generation supported for embedded systems.

As depicted in Figure 2.2 (b), the initial abstraction allows to omit a development aspect and later consider it when enriching the model in that respect.

Table 2.2. Coverage of integration aspects by the different approaches

Integration problems	Technique	Explanation
syntactical	A	Abstraction guarantees that the composition is syntactically correct.
technological	A/E(S)	Abstraction and enrichment (potentially by synthesis) provides some technological compatibility.
real-time local	E	The initial abstraction does not provide any guarantees.

(legend: A = during abstraction – E = during enrichment – S = by synthesis).

The explicit consideration of real-time constraints for a software function in a subsequent development step is an example for an abstraction and enrichment step. Upfront, the developer abstracts from the timing issues and instead focuses on the functional aspect of the solution. Then, in a later step the derived functional solution is enriched with timing information in the form of deadlines etc. As outlined in Table 2.2 the initial abstraction step does not provide any guarantee for the later enrichment and thus the integration problem has to be addressed late when the enrichment happens.

Often horizontal decomposition & composition, where parts of the system are decomposed at a specific abstraction level and vertical abstraction & enrichment, where the level of abstraction changes, are used in combination. An example for such a situation is when different parts or subsystems are developed by different stakeholders and one has only an abstract view on a subsystem provided by a supplier while other parts are available on a more detailed level.

Consistency & Synchronization

A third fundamental concept to handle integration issues is to not only decompose the problem initially and resolve integration problems later, but somehow reflect the dependencies between the different artifacts throughout the parallel development.

The first approach is to check the consistency of the models and resolve the issue immediately or at least in the near future rather than waiting for the time of integration. This case of consistency refers to horizontal consistency [10] and takes care that no conflicts arise when the models developed in parallel are later integrated.

Another option which provides a higher degree of automation is model synchronization [11] where the equivalent parts of two models are automatically kept

consistent. Like for consistency, the relevant case here is only horizontal synchronization of parallel-developed models (analogous to horizontal model transformations [12]).

The main benefit of both approaches is that in contrast to the two former ones the independently developed model can more freely evolve without resulting in harm later on. In the case of decomposition and composition in contrast somehow the basis for the separation is fixed after doing the upfront decomposition. Also in the case of abstraction and enrichment the separation is somehow a-priori fixed when doing the abstraction and it thus too only provides limited degrees of freedom when enriching the model later on. However, this higher degree of flexibility can only be preserved as long as the consistency rep. synchronization is keeping track of the dependencies to prevent integration problems later on.

On the other hand, unless fully automated as in the case of model synchronization there is the permanent need to resolve inconsistencies during the parallel development and therefore both parallel development activities might be slowed down considerably. Therefore, the sketched benefit comes with the drawback that no "fully" parallel and independent development is really possible.

If during the parallel development the consistency is checked as depicted in Figure 2.2 (c), integration problems during the later composition can be prevented. Co-simulation of different models, which are developed independently or for analysis purposes (like in the case of plant-models), is one example where different development activities are checked for their consistency (see Table 2.3). It is important to note that this also enables consistent changes of the interface between the initially separated subsystems. Without consistency checks interface changes would endanger the proper composition later on.

Table 2.3. Coverage of integration aspects by the different approaches

Integration problems	Technique	Explanation
semantic	P	The co-simulation helps to find inconsistent behaving development artifacts.

(legend: P = during parallel development).

Combinations in Practice

It is important to note that in practice instead of these pure cases of parallel and sequential processing you will encounter partially ordered activities that are coupled by the production and use of different versions of development artifacts depending on the employed decompositions and abstractions.

A frequently employed approach which combines horizontal decomposition with vertical abstraction is interfaces. When planning the decomposition, e.g., when horizontally decomposing a system into subsystems, interfaces are used to capture at a more abstract level the dependencies between the components. Additionally such interfaces can be also defined between layers at different level of abstraction. If the dependencies are properly designed in the interfaces this prevents that related integration problems will be encountered during composition. However, interfaces usually only cover a very restricted subset of the

component characteristics and they only prevent integration problems for that restricted set of characteristics. Examples where different sets of characteristics are covered by interfaces presented later in the chapter in the case of AUTOSAR and Mechatronic UML.

Using an abstract model of the environment or the other subsystems is another technique used in engineering which combines decomposition and abstraction. This is particularly useful when a simple interface will not capture all required properties of the environment properly. Please note that such an environment model together with the subsystem model can be checked during the parallel development against requirements of the system which could not have been broken down into subsystem requirements due to their non-compositional nature (like the reactive interplay between the plant and implemented control functionality). In the case of control engineering so-called plant models are employed to capture that part of the environment which is relevant. Simulation runs check that the given control requirements are met. The function development described in the next section is a typical example where environment models play a prominent role.

Another approach to derive a valid composition at a more detailed level is a dedicated manual or automated synthesis step. The synthesis step generates a solution which fulfills the constraints (e.g., fixed schedule for a dedicated hardware and software stack) configuring an underlying layer or determining an additional glue component. To fulfill several resource constraints the synthesis can also target to minimize the resulting resource consumption (e.g., synthesizing a minimal runtime kernel which only includes the necessary modules/functionality). Alternatively, an online solution is often employed (e.g., real-time scheduling in the case of processes on a real-time operating system). In this case usually an additional check at integration time is required that evaluates whether the constraints can be met (e.g., scheduability checks). Examples for synthesis approaches for the local real-time integration problem are Mechatronic UML (see Section 2.4.2) and TDL (see Chapter 5).

Depending on the specific domain and the severity of the encountered conflicts quite different means for the resolution of the integration problems are applicable. A technique can only be employed when the resulting solution adheres to the specific constraints of the domain (e.g., in a domain with high cost pressure such as the automotive domain using more powerful hardware and additional abstraction layers are often not affordable). Also the development efforts as well as the scalability of composition techniques are important factors that have to be considered. E.g., in the automotive domain the high cost pressure does not allow the intensive use of more powerful hardware and additional abstraction layers. However, in domains where safety issues prevail, like in the case of avionics systems, more advanced concepts such as IMA exist allowing the modular verfication of decomposed system parts [13]. Also the development efforts as well as the scalability of composition techniques are important factors that have to be considered.

2.3 State-of-the-Art Approach

The construction of current complex embedded systems – as found, e.g., in the avionics or automotive domain – is characterized by two sources of complexity: First, these systems are composed of *interacting distributed components.* Second, these components are developed *in parallel by different suppliers.* Thus, integration of these components becomes a core issue during development, especially if individual components of different suppliers are combined by the equipment manufacturer on a single electronic control unit.

To overcome the *problem of late integration*, often a model-based integration approach is chosen, allowing the development process to be modularized. Here, the techniques of *decomposition* and *enrichment* as described in Section 2.2.3 are used to obtain two *orthogonal dimension* of development: By using *functional decomposition*, braking the system down into separate functions or components, these functions can be constructed, validated, and verified independently. By using *incremental enrichment*, going from functional via logical to technical models, these components can then be safely integrated on a common platform.

To effectively support such a development process, two prerequisites are necessary: On the one hand, the approach must support the *description of the system at different levels of abstraction*, to support a stepwise enrichment of the models of an individual function. This ensures that the more detailed model respects the limitations of the more abstract model. On the other hand, the approach most support the *combination of the models of all functions* at each level of abstraction, to enable a safe integration of the overall system. This ensures that the combined functionality implements the intended overall behavior.

To support the different levels of abstraction, increasingly model-based approaches are used, especially for control functionality. Typically, here the functional, logical, and technical level are realized by function-oriented models (e.g., MATLAB/Simulink or ASCET-MD), by software-oriented models (e.g., TargetLink or ASCET-SD), and prototyping or pre-production platforms.

Currently, parallel engineering is achieved by decomposing the system into several components at the functional level. Thereafter, the different functionalities are developed separately. Composition is achieved mainly at the platform level by defining the realization of the joint interfaces (e.g., the exchanged bus messages).

Obviously, due to increasing dependencies between formerly independent functionalities such an approach requires to support a combination of the top-level functions. Furthermore, since obviously the late integration can cause inconsistencies (e.g., when fixing different discretizations of joint interface signals during the construction of the software model), current approaches specifically provide support for a safe integration at earlier levels (e.g., Intecrio or SystemDesk). Additionally, platforms (e.g., IMA[4], AUTOSAR[5]) and a corresponding development environment eliminate the need for the manual integration at the platform

[4] Integrated Modular Avionics.

[5] Automotive Open System Architecture.

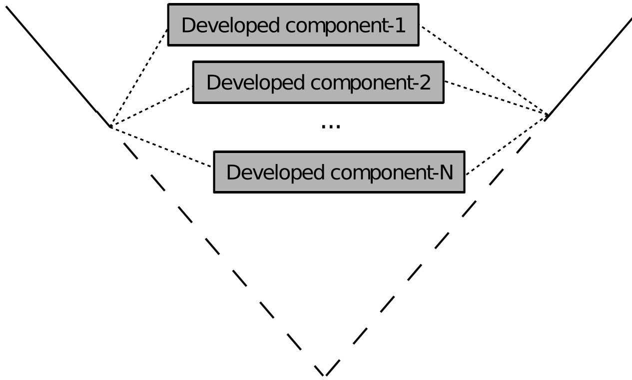


Fig. 2.3. Parallel development within the V-model (according to [14])

level, lifting the integration to the software level. Besides the functional aspects, such an approach requires to include system wide properties like real-time requirements.

As mentioned above and illustrated, e.g., in [14], the standard engineering process combines parallel and incremental development, allowing to develop components or functions in parallel while especially taking into account platform and other restrictions (e.g., real-time and resource restrictions) in a stepwise manner. The parallel development of multiple components is illustrated in Figure 2.3, showing that parallel development is achieved by forking the development for each component or function in the design phase, joining these components in the integration phase. Each component or function, as described in the multiple V-model shown in Figure 2.4, is iteratively developed with increasing level of detail via a simulation, prototyping, and pre-production stage.

As indicated in Figure 2.5, the development of functions at different levels of abstraction affects not only the design of the functions via the different models used at these levels; it furthermore affects the different validation and verification phases in the multiple V-model, e.g., concerning the used models of the environment or test cases. However, it also requires providing an integration of the

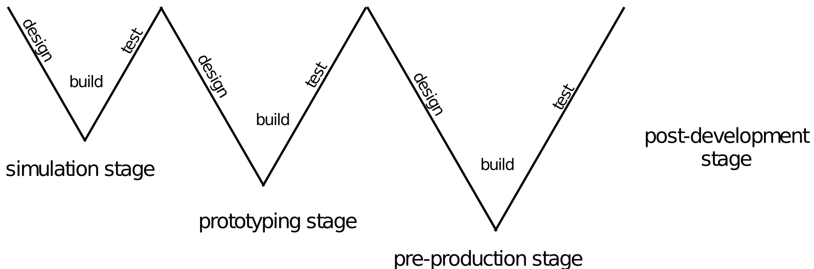


Fig. 2.4. Stages of the multiple V-model (according to [14])

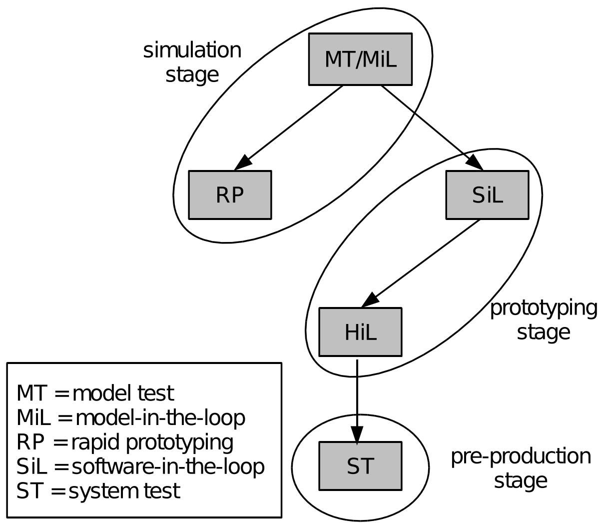


Fig. 2.5. Test and simulation activities within the different stages (according to [14])

functions at these different levels to support the early validation or verification of the system under development.

In the following, in Subsection 2.3.1 the development of an individual function at different levels of abstraction is described in more detail, while Subsection 2.3.2 illustrates the possibilities of integrating several functions at these different levels of abstraction.

2.3.1 Function Development

During function development a complex functionality in the form of a control algorithm or reactive behavior controlling a physical process is developed. This functionality is incrementally developed by adding constraints imposed by the plant (environment) as well as the platform in a stepwise fashion. This stepwise integration is accompanied by validation and verification techniques at the different stages. Figure 2.5 shows an overview of these different levels of abstractions as well as the corresponding techniques.

The simulation and prototyping stages use abstracting assumptions to eliminate details both from the environment as well as from the platform. Examples for such assumptions are unlimited HW resources, e.g., eliminating the need to consider execution times or memory consumption, or simplified plant models, e.g., eliminating the need to deal with failures of sensors and actors.

Simulation Stage

At this stage, purely functional models build the basis for development. Commonly, data flow models in form of block diagrams (e.g., ASCET, Simulink) or control flow models in the form of state diagrams (e.g., Statecharts) are used.

Functionality is developed independent from platform and its interfaces to the environment (e.g., A/D and D/A converter). Therefore, these models ignore properties like WCET limitations, characteristics of the HW (e.g., register size), or memory consumption. Typically, these models use values and signals of types that differ from these used within the real system (e.g., floating-point data types instead of fix-point, abstract messages instead of CAN messages). Due to the focus on the logical execution order and the data flow, for verification and validation often simulation of the models is used, using either no plant or a plant model as environment. The goal of this stage is a first proof of concept and the verification and validation of the overall design and control law. Using a plant model during the simulation stage in parallel supports the validation of the developed functionality concerning semantically as well as dataflow integration problems like described in Section 2.2.2.

For validation and verification, at the simulation stage model tests (MT), model-in-the-loop tests (MiL) and rapid control prototyping (RCP) are used.

For MT, one-way simulation of the models – also called one-shot simulation – is used. Here, all input and output values are generated and analyzed for a single execution of the system, abolishing the need for a dynamic interaction with the environment. For MiL, the model of the function is simulated back-to-back with virtual models of the dynamics of the environment in the form of a plant model. The plant model is an abstracted and simplified representation of the real environment, allowing a validation of the functionality. Due to the abstraction of the plant model, issues like the calibration of the functionality generally cannot be treated at his level. For a more accurate validation of the models, RCP can be used together with the models of the simulation stage. To that end, typically high-performance RCP-HW is used, allowing platform restrictions to be ignored (e.g., computations times, floating point vs. fix point). By replacing the plant model by the actual plant (or a close equivalent), RCP allows the functionality to be validated against the real plant including the real-time behavior of the plant, access special-purpose HW, and use actual actuators and sensors.

Prototyping Stage

At this stage, many aspects like real-time properties and resource restrictions removed by the platform abstraction in the simulation stage are taken into account. The focus of this stage is the implementation of the designed functionality and its validation and verification. Depending on the level of abstraction, only software aspects (e.g., modularization, used data types) or also hardware aspects (e.g., computation times and storage restrictions) can be addressed.

On the software level, software models are executed on a host computer. Unlike to MT and MiL, the software models additionally consider implementation aspects like discretization of the functionality in the value and the time dimension, e.g., by using fix-point arithmetics and task schedules. In practices, software models are often generated from the functional models via parameterized autocoders (e.g., ASCET-SD, TargetLink, or Embedded Coder) and in such a way that at the technological level the integration is supported via synthesis. An essential part of the prototyping stage consists in the verification that the models

from this stage are an enrichment of those of the simulation stage and intended semantics are fulfilled, as mentioned in Section 2.2.3. Here, enrichments dealing with refinement or approximation are often semi-automatically provided by a target code generator during discretization. By using a specific host platform, some aspects of the final platform are still ignored. Typically, resource restrictions are not examined or only in a simplistic fashion. Using such host platforms, verification and validation via MiL can be covered by corresponding techniques via software in the loop (SiL) test.

To again include more platform restrictions, software models to be executed on the target processor can also be used in this stage. Such an integration of software and target hardware can be done on different levels. The target processor can be used to run a processor in the loop (PiL) simulation, providing extended debugging or calibration functionalities. Therefore, in PiL simulation often the used platform is different to the final one, and often Evaluation Boards are used providing additional interfaces for debugging and calibrating. Alternatively, the real hardware (the ECU) can be used for HiL simulation, allowing reliable results to be obtained, e.g., also w.r.t. execution times, which may be affected by the additional debugging and calibration functionalities. In both cases, the environment is simulated via the use of a simplified and abstract model of the real plant. However, in the PiL approach often simplified models are sufficient, while HiL approaches in general use fine-grained models often requiring the use of real-time systems for execution. Furthermore, within the HiL simulation additional system parts can be included, e.g., legacy ECUs or even mechanical parts (e.g., the throttle of an otherwise simulated engine). By using more detailed execution platforms (e.g., in the case of PiL simulation or in the case the real ECU is used) additional properties like execution times and the resource consumption can be evaluated at least for the local case by using the concept of enrichment as introduced in Section 2.2.3.

Pre-Production Stage

Within the pre-production test the system is tested against external influences of the environment. This includes the effects on the system due to environmental conditions (like temperature, shock or vibration). The system under test is built of prototyping HW fulfilling the required specifications for the end product. The goal of the tests is to identify and fix problems and to measure the robustness of the system as early as possible.

2.3.2 Function Integration

The function integration happens at the latest when the functions are integrated during the pre-production stage. However, in contrast to such a big bang integration usually whole functional groups are integrated beforehand using replacements for the missing rest of the system in order to ease the integration testing.

The current practice for integrating multiple software modules on one node is characterized by the following stepwise partially manual process: (1) *Specification:*

The interface and decomposition of the software into modules are specified on a high abstraction level while mainly functional properties are targeted (if at all), then (2) *Partitioning:* The software is partitioned into concurrent modules resp. logical threads with appropriate periods to make it run on a real-time operating system or kernel (usually without adequate analysis), (3) *Implementation:* The software is implemented (often manually), (4) *Integration:* The threads are combined using either static schedules or concurrent threads in a RTOS or kernel and it is verified that the software fulfills all real-time constraints in its given environment. In the case the implemented software is combined with an RTOS, at least at the functional level the technological aspect concerning the integration is supported by composing the software with a standardized execution platform (e.g., an OSEK RTOS). Additionally scheduling analysis of the used RTOS tasks can be applied to analyze and validate the real-time behavior, at least for the local case. If the real-time constraints do not hold, partitioning, implementation and integration have to be repeated. Repeating this cycle a number of times is usually very costly but often unavoidable.

Prototyping Stage

The outlined early integration of functional groups might be addressed already at the prototyping stage when integrating the control algorithm with more appropriate substitutes for the final hardware. This could happen using the prototypes of other functions as well as their final version depending on the availability.

Pre-Production Stage

In the last stage of the multiple v-model (pre-production stage) the real system is build including the real plant.

The system is tested within the real-life environment to ensure that all requirements are met, including conformance to relevant standards like industrial or governmental ones. The build system is close to the later product and some calibration and configuration can be done. Tests concerning functional and non-functional properties are possible but fixing problems concerning properties of one of the earlier stages could not or only with extensive effort be done. E.g., to change the system architecture or the design of the control functions is rarely possible at this stage.

Encountered Problems

When the system is initially decomposed several properties (e.g., real-time properties or needed resources) are not considered. When composing the developed parts these properties can lead to crucial problems, potentially leading to extensive changes related to the earlier development stages. Furthermore, the composition of the developed parts can result in characteristics which are caused only by the composition and not by the characteristics of the components itself. For example, if several independent developed components have to use the same communication channel (e.g., a shared bus) problems can occur which could be hardly detected when components are tested and simulated individually.

Decomposition at the system level is almost done at the same architectural level (while additionally layered architectures play an important role, e.g., in case of the integration of operating system properties like scheduling).

2.3.3 Discussion

The outlined process of decomposition, functional development and system integration represents a nearly optimal solution for systems with more or less independent functions and functions which require only a usual control law. Table 2.4 and 2.5 summarize the integration aspects concerning function development and function integration, which are somehow supported by the described standard approach. Unfortunately today's embedded real-time systems often include much more sophisticated designs where an overwhelming number of functions exist that have to interact in complex ways to achieve the envisioned overall functionality. Therefore, this style of development and integration often result in severe problems perhaps being detected rather later during system integration. These can be true for

- interface compatibility problems,
- protocol compatibility problems,
- dependability issues,
- real-time behavior and
- resource consumption.

In all these cases, the rework required to fix such problems if encountered during system integration can be quite costly.

Table 2.4. State-of-the-Art: coverage of integration aspects during FD

Integration problems	FD	Explanation
technological	A/E(S)	via code generation, standardized tools (e.g., MATLAB)
syntactical	D/C	define interface to sensors and actuators of the plant
semantic	A/E(S)/P	enrichment at each stage using (initial abstract) refined SW (potentially synthesized) and HW models; simulation of the SW models in combination with the model of the plant
dataflow	A/E(S)/P	enrichment at each stage using (initial abstract) refined SW (potentially synthesized) and HW models; simulation of the SW models in combination with the model of the plant
real-time compatibility		
local	E/P	enrichment at each stage using refined SW and HW models; consistency with HW checked during parallel development by means of simulation techniques, e.g., PiL simulation
distributed		
resource consumption		
local	E/P	enrichment at each stage using refined SW and HW models; consistency with HW checked during parallel development by means of simulation techniques, e.g., prototyping stage
distributed		

(legend: FD = function development – A = during abstraction – C = during composition – D = during decomposition – E = during enrichment – P = during parallel development – S = by synthesis).

Table 2.5. State-of-the-Art: coverage of integration aspects during FI

Integration problems	FI	Explanation
technological	A/C	later integration on standard platforms such as OSEK allow upfront abstraction
syntactical	A/(D)/C	upfront definition of components and their interfaces (not standard)
semantic	C/(P)	checked during integration testing – potentially using functional groups in parallel with replacement for the rest of the system
protocol	C	checked during integration testing
dataflow	C/(P)	checked when programming and compiling the integrated code, checked during integration testing – potentially using functional groups in parallel with replacement for the rest of the system
real-time compatibility		
local	C	scheduling analysis of the integrated tasks
distributed	C/(P)	using simulation techniques (later in the development lifecycle) – potentially using functional groups in parallel with replacement for the rest of the system
resource consumption		
local	C	resource analysis of the integrated tasks and their code
distributed	C	analysis of the bus allocation for the integration of multiple nodes

(legend: FI = function integration – A = during abstraction – C = during composition – D = during decomposition – P = during parallel development –).

2.4 Advanced Model-Based Solutions

A number of model-based approaches for the development of complex embedded real-time systems have been proposed to avoid some of the problems encountered for the standard approach for system development as outlined in the preceding section. We will first review an industrial approach developed in the automotive domain. The approach deals with some integration problems present in complex and highly heterogeneous systems. In addition, we will discuss a particular academic proposal and a number of related approaches that try to address other challenging integration problems.

2.4.1 AUTOSAR

AUTOSAR has been founded 2003 as an industrial standardization body for the automotive industry and is now supported by all major car manufacturers, their suppliers, semiconductor producers, and tool suppliers (see also [15, 16]). AUTOSAR aims at the improvement of the software development and integration for Electronic Control Units (ECUs) by providing standards for software architectures and software modeling techniques.

AUTOSAR responds to an increasing number of problems with the development of ECUs, especially software development and software integration:

Traditionally manufacturers bought dedicated ECUs for dedicated functionalities from ECU suppliers, e.g., an engine ECU or a left-door ECU. The increase of functionalities has led therefore to an increase of the number of ECUs – reaching numbers of up to 80 ECUs in some vehicles. A further increase of the number of ECUs is hardly feasible: Function integration in such settings means

integrating ECUs via communication busses; a feat which becomes more and more difficult when the number of functionalities increases and functionalities need to communicate intensively with each other.

This dilemma is solved by breaking with the tradition of equating functionalities and hardware modules (i.e. ECUs). Instead software – not hardware – becomes the means for implementing functionalities. This decouples the number of ECUs from the increasing number of functionalities. But on the other hand this entails changes in the development process: Manufacturers will not continue to buy production-ready ECUs from suppliers. Instead they will buy software modules, i.e. functionalities, and (generic) hardware platforms separately. So function integration then means software integration.

In the automotive industry, software has so far not been a product; just finding a pricing schema for software will therefore be a challenge in itself. Furthermore, in the future several software functionalities must be integrated on one hardware platform consisting of one or more interconnected ECUs. This will either be done by traditional suppliers or by manufacturers which try to extend their fields of competence – and try to become more independent from suppliers.

Another problem addressed by AUTOSAR is the testing and quality assurance of the developed ECUs. Would it be just for the increasing number of ECUs, the problem could be solved by also increasing the number of ECU tests. But nowadays most functionalities communicate with other functionalities (which may be on other ECUs) or high-level functionalities are even implemented by combining already existing functionalities—being an example for the emerging behavior described in Section 2.2.3.

Testing such distributed functionalities is difficult since all possible communication combinations between the building functionalities must be covered – causing in the worst-case an exponential increase of necessary test cases. Traditional engineering fields such as electrical engineering have faced this problem for decades and have mainly come up with three main solutions: (*i*) Systems are decomposed into separate components, in doing so dependencies inbetween are minimized – corresponding to the ideas from Section 2.2.3. (*ii*) Components are tested separately. Tested and used components are then reused in later projects – thus avoiding unnecessary and error-prone new implementations. (*iii*) Errors are taken into consideration, e.g., by means of safety margins and diagnosis functions.

AUTOSAR is trying to apply these ideas – to some extent – to automotive software architectures. Generally speaking AUTOSAR comprises three main activities:

(1) Introducing Software Component Models
(2) Standardizing Basic Software Modules
(3) Standardized Application Interfaces

Compared to the integration problems discussed in Section 2.2.2 in the current release version 3.1 of AUTOSAR primarily syntactical aspects are supported in the form of standardized interfaces for components and modules. Other key issues like semantics, protocol integration or real-time properties are not covered.

How other aspects can be included into the AUTOSAR standard is discussed for the case of real-time properties in [17].

(1) Software Component Models
AUTOSAR introduces a software component model for automotive application software. By this, AUTOSAR wants to facilitate (*i*) software reuse and (*ii*) software exchange between different parties. Software components can either be atomic (i.e. components implemented as C Code) or compositions which are formed by interconnected software components. In AUTOSAR all applications of a vehicle are part of one overall software composition – i.e. independent of their distribution on several ECUs. This top-level composition forms the vehicle's software architecture. From a software developer's point of view, AUTOSAR software components mainly introduce standardized C-APIs to communicate with other software components, either on the same ECU or via the communication bus. No direct calls to C code contained in other software components or to C code of basic software modules such as drivers are allowed anymore. This makes the software independent of other components, hardware and basic software modules – and hence reusable. By using APIs in form of C-APIs not only syntactical aspects but also technological ones are treated.

These standardized C-APIs are based on a classical port concept. I.e. a software component A will not send a signal directly to software component B anymore; e.g., it will not directly call a method $B_receiveSignal()$. Instead A sends this signal to one of its own ports; ports being just proxies for other software components. A software architect later on connects (normally in a modeling tool) A's port to the corresponding receiving port of B. The same concept is used to connect software components to basic software modules such as I/O drivers, the COM stack, or error management modules.

The usage of standardized C-APIs to communicate with a component's environment has several advantages: Applications become reusable, hardware and basic software dependencies are eliminated, communications become clearly visible. But on the other hand, standardized C-APIs also mean that (*i*) existing C code must be wrapped or modified and (*ii*) existing tool chains such as code generators must be adapted. These changes to the development process may well delay the introduction of new AUTOSAR concepts.

This leaves one key question unanswered. How are A and B connected on the C code level? Who generates the gluing code to implement the signal transfer from A's sending port to B's receiving port? In AUTOSAR this is done by a middleware layer, the so-called Run-Time Environment (RTE). The RTE comprises the C code for the definitions of the C-API commands used within software components. Since resource consumption and process usage are important for ECU software development, this RTE code is generated for each ECU individually.

Here two key concepts of software integration can be seen implemented: (*i*) Software components and compositions are an example for composition and decomposition (see Section 2.2.3). (*ii*) The standardized C-API generated by the RTE abstracts the underlying hardware platform and is therefore an example

of abstraction (see Section 2.2.3). Thus, the AUTOSAR framework supports the composition and decomposition at the same conceptual level using software components and compositions, while the RTE is an example where abstraction is used in case of a layered architecture to allow the decomposition of the system at different conceptual levels like discussed in Section 2.2.1.

From a process point of view, application software components and compositions are modeled first and then mapped onto ECUs – ECUs and their basic software can also be described by AUTOSAR. The developer then connects the application software components to basic software modules such as I/O drivers or the operating system. These basic software modules also have to be configured appropriately. E.g., tasks have to be defined for the operating system. Then the RTE can be generated which connects software components to other software components and to basic software modules.

(2) Standardized Basic Software

AUTOSAR also standardizes the C-API and the configuration files for ECU basic software modules such as I/O drivers, COM stack, operating system, error manager, mode management, and network management. One goal was to make basic software modules interchangable, i.e. a COM stack from provider 1 should be used with an operating system from provider 2. Furthermore configuration settings should become more reusable, i.e. the configuration files from an older project (where provider 1 was used), should also be usable for a newer project (where provider 2 is used). The introduction of standardized basic software modules also eases the implementation of the RTE because the to-be abstracted basic software becomes more uniform.

The basic software modules are organized into several layers, making them an example for the vertical abstraction described in Section 2.2.3 supporting syntactical as well as technological aspects.

(3) Standardized Application Interfaces

Another activity from AUTOSAR is the standardization of the APIs for application software components. E.g. the interior light control software used by different manufacturers should have the same API. By this, manufacturers and suppliers hope for fewer redundant implementations of the same functionality by different software providers and for easier integration processes. Of course, only commodity modules are standardized—no standardization is planned in competitive areas.

As model-based integration is the key issue here, it is worth reviewing the AUTOSAR approach with regard to improvements of the ECU integration process. First of all, several integration steps exist: basic software integration, ECU integration, and ECU system integration. In the following, AUTOSAR's contribution to these integration steps are assessed using the categories from Figure 2.1.

Basic Software Integration: Basic software is composed of different software modules: operating system, drivers, services, and communication stack – most of these modules are further decomposed into different submodules. AUTOSAR makes the integration of these modules into one basic software layer easier by

means of syntactical, technological and data flow agreements: *(i)* The decomposition is standardized including the C-APIs between modules and *(ii)* configurations are expressed using a standardized set of parameters – being an example of the "Decomposition & Composition" principle of Section 2.2.3.

AUTOSAR does not address the key issue of semantic and protocol integration: In the standard, the precise behavior of the modules is not defined in a formal way, leading to integration problems such as incorrect emerging behavior – especially since the basic software is implemented by different software suppliers. This problem is worsened by the large number of interacting and behavior-influencing parameters. Furthermore real-time issues are not modeled in a satisfying manner, leading to problems with the temporal features of the integrated software system.

All these drawbacks lead to situations where, e.g., one basic software layer from software supplier A behaves differently to an equally configured basic software layer of supplier B – a situation rendering the reuse of configurations and the exchange of software modules almost useless. A solution could be a precise, executable model of the basic software behavior including the effects of parameter settings.

ECU Integration: In this step, the application software components on one ECU are integrated with each other and with the basic software layer. Unlike with the basic software, the decomposition and the C-API cannot be standardized in most cases – except for AUTOSAR's limited " Standardized Application Interfaces" activity explained above. This make the application of the "Decomposition & Composition" principle harder, in fact the interface (and port) principle from Section 2.2.3 must be used: modules (i.e. components) do not refer directly to each other but refer indirectly to each other via ports and interfaces. The data flow between modules is modeled by means of connections between ports; at run-time these connections are implemented by the RTE middleware (see above). So again, AUTOSAR solves to some extent the syntactic, technological and data flow integration problem (see Figure 2.1).

The introduction of the component/interface software engineering pattern causes the need for explicit software architectures; which in turn causes the need for software architects, for a separate software design step in the development process and for appropriate tools. This significant change to the development process is one of the challenges when AUTOSAR is introduced: Software architecture models must be synchronized with existing models (e.g., behavior models), new tools must be tested, and new development teams must be established.

Just like with the basic software integration, semantic, protocol, real-time, and resource consumption integration problems are not addressed sufficiently. I.e. predictions about the functional and real-time behavior of the integrated ECU cannot be made. Since AUTOSAR does not cover algorithmic models, a solution to the semantic and protocol problem cannot be expected.

Predicting the precise resource consumption of the ECU (i.e. processor and memory usage) of the integrated software is another unsolved but highly relevant

issue. Estimation techniques and simulation approaches might help in the future to ease this problem.

ECU System Integration: In this final step, the ECUs are integrated into the overall ECU network. The main agreement or contract between the ECUs is the communication configuration, i.e. the messages and signals used to transport information on the communication network. In traditional development processes, this configuration is defined first – and it is defined manually. In AUTOSAR, this configuration is derived automatically from the mapping of the software architecture on the hardware topology. This eases the integration since software architecture and network configuration are therefore synchronized automatically – an example of the synchronization principle of Section 2.2.3.

Real-time problems such as too high message delays on the network are not addressed. Neither are resource consumption problems such as too-high network loads. Again, due the lack of behavior models in AUTOSAR, problems concerning the dynamic interaction between ECUs (semantic and protocol problems) cannot be expected to be solved by AUTOSAR. Dependability issues such as redundancy are also relevant but are currently not solved satisfyingly by AUTOSAR.

AUTOSAR is continuing to extend the standard (see, e.g., [18]). Currently AUTOSAR works on topics such as variant management, MultiCore support, functional safety, and the modeling of timing information such as end-to-end timing on the application level – this may ease the real-time integration problems.

This short overview of AUTOSAR's role in the automotive software integration process shows that AUTOSAR helps mainly with statical, functional integration problems such as syntactic, data flow, or technology issues. Dynamical problems such as semantic and protocol issues are not solved, neither are non-functional issues such as the estimation of resource consumptions. So AUTOSAR is not the "golden bullet" for integration but only a first step towards a software-aware development process in the automotive industry.

Several studies have shown that AUTOSAR requires significant changes of the development processes and of current business models: Software becomes a product, software models must be created, new roles – e.g., a software architect – must be established, manufacturers try to become software integrators, and new tools must be introduced.

This leads to a problem that goes beyond simple missing features of AUTOSAR such as insufficient support for dynamic, real-time or non-functional integration aspects like in the case of needed resources: AUTOSAR's approach to software engineering has been, from the very beginning on, based on the component-oriented software engineering paradigm – mainly influenced by the EAST project (see [19]). This paradigm requires an explicit software architecture defined as inter-communicating software components. And it requires therefore an explicit software architect, an explicit tool chain for software architectures, explicit verification and testing strategies for software architectures, and especially an explicit software architecture design step in the development process.

Table 2.6. Coverage of integration aspects using AUTOSAR

Integration problems	AUTOSAR	Explanation
technological	D/E/S	C-APIs; provides standardized platform and supports code generation; code generation for the implementation of the RTE provided by tools
syntactical	D/A/E	AUTOSAR standardized APIs and means to define components and ports; virtual function bus provides realization; provides layered architecture with interfaces between
semantic	C	checked during integration testing
protocol	C	checked during integration testing
dataflow	C	checked when programming and compiling the integrated code, checked during integration testing
maintainability	D/A	decomposition the software architecture; abstraction via standardized interfaces between different layer
real-time compatibility		
local	C	scheduling analysis of the integrated AUTOSAR/OSEK tasks
distributed	C	using simulation techniques later in the development lifecycle
resource consumption		
local	C	resource analysis of the integrated tasks and their code
distributed	C	analysis of the bus allocation for the integration of multiple nodes

(legend: A = during abstraction – C = during composition – D = during decomposition – E = during enrichment –

S = by synthesis)

While these requirements can be met in classical computer science domains such as business software or telecommunication, this must not be true for the automotive software development. This domain possesses an established development process based on ideas from control theory and signal processing – and it possesses an adequate established tool chain and adequately trained developers. So one might ask whether AUTOSAR should have chosen a software architecture paradigm leveraging established procedures. And one might ask whether, instead of choosing a software architecture approach from a technical (computer science) point of view, AUTOSAR should have chosen an approach which would minimize changes to existing development processes and which would exploit strengths of automotive's long-term and successful software development history.

To give an example: Data-centric software engineering approaches (see [20, 21] for details) couple software components via a signal repository. Components may either write or read signals in the repository. Unlike with component-oriented approaches, no explicit software architectures are required – and therefore no separate tool chains and fewer changes to development processes are needed. And such an approach also resembles the existing automotive development process where ECUs communicate via communication buses, i.e. via a common pool of bus signals. Of course, this does not mean that a data-centric approach would solve all problems. But it may serve as an example that fundamentally different alternatives would have existed and might have demanded fewer changes to the established development process.

2.4.2 MECHATRONIC UML

As outlined in Section 2.3.2, the current practice for model-based development of software components with hard real-time constraints – whether AUTOSAR is employed or not – is characterized by the following step-wise partially manual process (1) *Specification*, (2) *Partitioning*, (3) *Implementation*, and (4) *Integration* which has to be repeated when the integration is not able to fulfill the required real-time constraints.

Consequently, it would be attractive to extend the idea of model-driven architecture (MDA) [22, 23] to design software for embedded hard real-time systems. When using MDA for such systems, the developer would have to specify the so-called *Platform-Independent Model (PIM)* which describes the system behavior including the real-time constraints which must be met. Ideally, a tool would then automatically partition the specification and map it to the *Platform-Specific Model (PSM)*, based on a *Platform Model (PM)* that provides details about the target platform. The PSM describes the active objects and their scheduling parameters which are required to implement the system behavior, specified by the PIM. In the next step, the PSM would be compiled automatically into the platform-specific implementation which guarantees a correct implementation of the PIM's semantics. The implementation would guarantee the real-time constraints by construction and thus, no verification of the real-time constraints is required. This would make the above mentioned manual steps *(3) Implementation* and *(4) Integration* unnecessary. However, the UML standard as well as proposed extensions for embedded real-time systems [24, 25, 26, 27, 28, 29, 30, 31] fail to provide a proper basis for this as the suggested models are not sufficient to talk about platform-independent real-time behavior.

The MECHATRONIC UML approach (MUML) [32] in contrast provides the missing platform-independent real-time models and also supports MDA for embedded real-time systems [33]. Therefore, by applying MUML the sketched iterative manual process often followed today in practice can be avoided by using the automatic mapping of a PIM to a PSM that is appropriate for real-time systems. In addition to (1) MDA for embedded real-time systems, MUML provides support for two particular problematic cases for integration embedded real-time systems: (2) the real-time coordination of embedded real-time systems and (3) their safety analysis. Tool support for (MUML) is provided in the form of the Fujaba real-time tool suite, which offers a wide range of UML based diagrams, the appropriate extension for the specification of real-time properties as well as modelchecking and consistency analysis support [34].

(1) MDA for Embedded Real-Time Systems

The structure of embedded real-time systems consist of a complex architecture of components. UML [35] despite it shortcomings can be considered as the standard to model complex software systems even in the real-time domain [28, 29, 30, 31]. MUML therefore supports to specify the architecture and complex real-time communication between the components by UML component diagrams and patterns respectively [36].

The semantics of the UML State Machines assumes the transitions to be fired within zero-time cannot be realized in practice and the pragmatic interpretation that zero-time means *fast enough* is only helpful in simple systems where a single periodic deadline can characterize for the whole state machine and it states what *fast enough* means. Therefore, in mUML *Real-Time Statecharts* (RTSC) [32, 37] extend UML State Machines to allow the explicit specification of the really required timing. Transitions are not assumed to fire *infinitely fast*, which is unrealistic on real physical devices (especially when considering the execution of the actions attached to the transitions), but it is possible to specify deadlines for each transition which in turn determine what *fast enough* really is. Similar to the notion in timed automata [38, 39] clocks and clock invariants are employed to describe when transitions are enabled and what the minimum time and the maximum time (d_0, t_{ans}) for finishing the execution of a transition has to be (more details see [33]).

Generating a PSM, consisting of active objects and deadlines, that guarantee the real-time constraints as specified in the model is of course only possible, when the model does not contain any conflicts between the declarative elements such as time guards and time invariants. A possible conflict is, for example, when multiple real-time constraints are contradicting and thus no behavior exists which fulfills them (time-stopping deadlock). To exclude such conflicts, the full state space of a Real-Time Statechart model has to be checked in the general case. As outlined in [33], model checking with UPPAAL and static analysis techniques can be employed to exclude such conflicts.

In order to generate the PSM, WCETs are required for all actions (side effects, *entry()*, *exit()*, and *do()*- operations) and for the elementary instructions that build the code fragments realizing the Real-Time Statechart behavior (e.g. checking guards, raising events, etc.).

As the WCETs are platform-dependent, we first deploy our components (whose behavior is each specified by a Real-Time Statechart) by a UML deployment diagram. In such a deployment diagram, we assign the component instances of our systems to dedicated nodes and the cross node links to available network connections in form of busses or direct communication links. Given such an assignment, we can further look into the specific characteristics of the different nodes as described in the platform model.

To analyze the resulting model with platform-specific annotations, we extend our timed automata model for model checking as well as our static analysis technique such that it also reflects the WCET behavior of the side effects of the transitions (cf. [33]).

After modeling and analyzing the PIM with components and Real-Time Statecharts and specifying the platform-specific WCET information in the PM and the deployment, we have to map the components and links to active objects and to network and communication links to come up with the final platform-specific model. In our case the PSM can be described by the UML Profile for Schedulability, Performance, and Time [29], as it allows the specification of priorities, periods, and deadlines for active objects. We use it as a platform-*specific* model,

as these values, which we derive automatically from the platform-*independent* model, are different for different platforms. For such a PSM we can derive code that guarantees the in the PIM and PSM specified timing constraints for Real-Time Java and C++.

While MUML has been developed in the context of a research project different case studies have been realized like described in [40] using an evaluation platform equipped with a 40 Mhz Power PC processor. For the derivation of WCETs the tool Bound-T[6] has been employed within the evaluation example described in [41].

Table 2.7. Coverage of integration aspects using MUML

Integration problems	MUML	Explanation
technological	D*	platform-independent model and code generation for map those the a specific platform (*but only realized for one)
syntactical	D/C	models capture components and ports; mapping to code provides realization
semantic	(D)	model checking of the models prevent some semantic integration problems
protocol	D	model checking of the models exclude protocol-related integration problems
dataflow	D	dataflow part of the interface and modular syntax checks guarantee proper dataflow specification
dependability		
safety	D	compositional hazard analysis of the models enable up-front guarantees; requires HW reliability data
real-time compatibility		
local	S	generated task periods and scheduling analysis guarantee correct timing
distributed	D	generated local tasks plus model checking guarantee correct distributed timing
resource consumption		
local		
distributed		

(legend: A = during abstraction – C = during composition – D = during decomposition – S = by synthesis).

(2) Correct Real-Time Coordination

As MUML further provides a compositional verification approach for the real-time coordination of systems of systems with reconfiguration [36, 42, 43, 44]. It further allow the model-based analysis of interoperability problems for the functional and real-time behavior. By extending UML components the syntactical compatibility (data, operations, ...) and semantic compatibility (data, operations, ...) is guaranteed while a run-time environment guarantees technological compatibility. In [45] is outlined how MUML interfaces also take care of the execution order of dataflow computations employed for evaluating control algorithms. The extended port specifications by means of RTSC together with the mentioned verification further ensure protocol compatibility (non uniform service availability, synchronization) and real-time compatibility.

(3) Safe Real-Time Systems

In addition, an approach for a compositional safety analysis [46, 47] permits to do a model-based upfront analysis of the resulting system safety when decomposing

[6] http://www.tidorum.fi/bound-t/

the systems into components in the form of an architecture. Therefore, safety issues have not to be addressed later when integrating the components into the overall system.

To sum-up, Table 2.7 provides an overview of the integration problems more or less covered by the MUML approach. Like mentioned before MUML has been developed in the context of a research project. While several studies have shown the applicability of the approach a coherent professional tool chain currently does not exist.

2.4.3 Other Approaches

A number of other approaches using models that also address several of the integration problems outlined in Figure 2.1. We will provide only a sketch of their benefits in the following text and refer to the referenced literature resp. chapters for more information.

Infrastructure Abstraction

Several approaches address like MUML the problem that real-time issues can in the traditional approach only be addressed rather late and that the resolution of related integration problems can become quite costly. The time-triggered approach [48] addresses this problem at the hardware and network level and provides a platform where the different real-time communication issues can be clearly separated with respect to time and dependability. An approach which similar to MUML address the timing problem at a single node is Giotto [49] as well as its successor TDL (see Chapter 5). Here a virtual machine guarantees that time constraints specified in the specification are guaranteed by the execution environment also allowing the higher level abstractions to be analyzed to detect protocol compatibility problems.

Other approaches try to synthesize a proper task allocation from a given software model [50] in order to meet the timing requirements. In addition, besides avoiding the integration problem a model-based analysis of the composition of distributed real-time embedded system may also be simply beneficial by enabling an earlier analysis [51].

An approach targeting to an earlier analysis concerning the later used HW infrastructure, including multiple nodes and the communication path between them is described in [52]. Virtual execution platforms, which represent the later used HW infrastructure, are used to provide an execution environment for simulation purpose, taking characteristics like the execution time into account.

An approach somehow in the middle is platform-based design [53] where no full abstraction is provided but instead the stepwise realization of higher-level abstractions by means of underlying platform components are the main design step. This often allows to reduce the integration problems as designs are derived by proper combinations of components with some degree of built-in compatibility as they together represent a platform and not a ragtag group of components. In some cases even correctness-by-construction can be achieved [54] by means of a platform-based approach.

The problem is also related to problem of heterogeneity (technology as well as semantics) which is in particular problematic when the artifacts to be integrated have not been decomposed upfront with the same model of computation (see Chapter 1).

Interfaces and Component Models for Integration

Like advocated in AUTOSAR and the mUML approach, interfaces and component models are a suitable concept to address integration issues upfront when decomposing a system. Related approaches propose extended interface for component-based design [55, 56] also covering stateless and stateful protocol behavior as well as real-time behavior. In [57], like in [45] for mUML, interfaces that take care of the execution order of dataflow as employed for computing control algorithms are presented.

One more component-oriented approach is the rich component model [58] that has been proposed mainly targeting reuse but also support early checking to avoid integration problems. Another is the Behavior-Interaction-Priority (BIP) component framework [59, 60] that can ensure the proper deadlock-free composition using a much simpler check of the resulting dependency graph rather than the complete component synchronization and thus permits to do it upfront when decomposing a system.

In [3] several approaches are discussed that provide specific DSLs for component models (EAST-ADL and AADL): For example, AADL [4] is a DSL for the development of embedded real-time systems which supports the description and analysis of the system architecture addressing the integration of SW and HW parts which can be developed by different stakeholders. EAST-ADL is a DSL and architecture description language which is based on UML and SysML. One key aspect of EAST-ADL is the usage of abstraction and an according system model is structured with several abstraction layers. The EAST-ADL language can be used within a tool, like it is done in the form of Papyrus for EAST-ADL.

Integrated Model-Based Development

Another thread of work focuses on a proper representation of all required system characteristics by means of models and their consistent further elaboration. In [3] several approaches are discussed that provide tool support (Fujaba [5], GeneralStore [61], ToolNet [62] and IDM [63]) for the integrated model-based development of embedded systems.

For keeping the different models, potentially used in different tools consistent, model-transformation and model-synchronization techniques can be used. In [64] the authors describe how modelsynchronization is used to keep AUTOSAR and SysML models consistent.

In addition in [65] model-integrated computing (MIC) as a paradigm to address the integration problems for embedded real-time systems. In [66] it is advocated that the MIC approach employing a number of domain-specific languages (DSL), supporting the proper consistency of the different model, doing frequent model analysis by mapping these models to available analysis tools, and generating refined models as well as code (synthesis) can help to substantially

reduce the later experienced integration problems. In [67] an implementation of model-based integration for the development of avionics systems is evaluated. Within this evaluation the benefits of the Model-Based Integration of Embedded Software (MoBIES) development process[7] and a special DSL (ESML) for the development of avionic systems are evaluated.

2.5 Summary

If we review the presented results, we can conclude that a number of promising approaches for different problems exist, while no solution for the overall problem seems available. This impression is also confirmed by the coverage of the integration problems summarized in Table 2.8.

Table 2.8. Coverage of integration aspects by the different approaches

Integration problems	FD	FI	AUTOSAR	mUML	Other approaches
technological	A/E(S)	A/C	D/E/S	(D)	
syntactical	D/C	A/(D)/C	D/A/E	D(UML)	D [55, 56, 59, 60]
semantic	D/E(S)/P	C/P	C	D(UML)	D [55, 56, 59, 60]
protocol		C	C	D	D [55, 56, 59, 60]
dataflow	A/E(S)/P	C/(P)	C	D	D [57]
dependability/ quality of service					
reliability		(C)			
availability		(C)			
safety		(C)		D	D [58]
security		(C)			
maintainability			D		
real-time compatibility					
local	E/P	C	C	S	S [49], see TDL in Chapter 6
distributed		C/(P)	C	D	D [56, 58], S [48, 50]
resource consumption					
local	E/P	C	C		
distributed		C	C		D [48]

(legend: FD = function development – FI = function integration A = during abstraction – C = during composition – D = during decomposition – E = during enrichment – P = during parallel development – S = by synthesis).

If we review the summary of the findings depicted in Table 2.8, we can make the following specific observations:

– It seems that the need for support of the functional integration problem at the technological and syntactical level has been identified also in industry and AUTOSAR or related approaches start to address them in a standardized manner.

[7] A project funded by the Defense Advanced Research Projects Agency (DARPA).

- In contrast semantic, protocol or dataflow issues are at first addressed by academic research projects but in practice they are addressed rather late if at all (compare Section 2.3). Here it seems beneficial if the existing research results could be transferred into industrial strength solutions to minimize the integration costs by addressing these issues earlier in the development life cycle. However, as these issues related to some extent to formal modeling, it is not clear whether such a transfer is really possible taking the existing workforce and their educational background into account.
- The integration of non-functional dependability resp. quality-of-service aspects is besides safety not very well covered either by industrial nor research approaches. This is probably due to the fact that these are often system properties which could not be easily established using in a compositional manner and indicates that these topics require much more attention from the research community.
- Concerning real-time compatibility we can observe that several well-suited research results have been achieved and some of them are in a transition phase to industrial praxis (e.g., see TDL in Chapter 5). These solution promises to ease the integration efforts considerably as they permit to exclude that the integration problems are detected rather late resulting in enormous costs due to the required rework of the integrated solutions.
- Finally, the resource consumption is a currently rather superficially covered aspect. However, the importance of hardware costs in fields like the automotive domain as well as the increasing importance of energy efficient and resource-aware solutions will make this aspect another highly relevant research topic. Here also the problem seems to be that resource consumption is a system property that is not easily addressed in a compositional manner.

If we take a look at the overall picture, we can see that handling an integration problem at composition time (C) as advocated in the traditional functional development and functional integration is in principle always possible. However, there is a clear trend that model-based integration results in a front loading where instead of costly efforts to handle integration problems after the fact these problems are upfront addressed by decomposition/composition (D), abstraction/enrichment (A) or parallel development & consistency (P). The more mature approaches are those ones where instead of checking integration problems late when combining the different system constituents, the seperation in the form of decomposition or abstraction provides already the basis to exclude or limit most of these problems upfront (see also [9] for a related observation based on several industrial studies).

However, it also became apparent that besides MUML and rich components [58] most approaches provide a rather isolated solution to one integration problem. Therefore, the main challenge for integration seems to be establishing a comprehensive solution that covers not only the rather simple problems such as syntax and technology compatibility but also most of the challenging aspects such as protocol compatibility, dataflow compatibility and real-time behavior. As most proposals for these advanced integration concepts have not yet been employed thoroughly in

industrial practice and may have contradicting constraints, it is not clear whether such an "integration of advanced integration concepts" is really feasible.

Therefore, the current challenge is not only to develop better solutions of the outlined separate integration problems (cf. Figure 2.1) but also to combine the existing solutions into overall integration approaches that provide a coherent solution that covers all required integration problems. It can be expected that suitable overall integration approaches have to be tailored for the specific domain of embedded real-time systems such as AUTOSAR while its ingredients will often be applicable in several domains.

Acknowledgements

We thank Ingolf Krüger and Florence Maraninchi for their feedback on earlier versions of the paper.

References

[1] Lane, J.A., Boehm, B.: System of systems lead system integrators: Where do they spend their time and what makes them more or less efficient? Systems Engineering 11(1), 81–91 (2008)
[2] Sage, A.P., Lynch, C.L.: Systems integration and architecting: An overview of principles, practices, and perspectives. Systems Engineering 1(3), 176–227 (1998)
[3] Chen, D., Torngren, M., Shi, J., Gerard, S., Lonn, H., Servat, D., Stromberg, M., Arzen, K.E.: Model integration in the development of embedded control systems - a characterization of current research efforts. In: 2006 IEEE International Symposium on Computer-Aided Control Systems Design, October 4-6, pp. 1187–1193 (2006)
[4] Feiler, P., Gluch, D., Hudak, J.: The architecture analysis & design language (aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA (2006)
[5] Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. International Journal on Software Tools for Technology Transfer (STTT) 6(3), 203–218 (2004)
[6] Mosterman, P.J., Ghidella, J., Friedman, J.: Model-based design for system integration. In: Second CDEN International Conference on Design Education, Innovation, and Practice, Kananaskis, Alberta, Canada, July 18-20 (2005)
[7] Dijkstra, E.W.: On the role of scientific thought, pp. 60–66. Springer, New York (1982)
[8] Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems Into Modules. Communications of the ACM 15(12), 1053–1058 (1972)
[9] Bosch, J., Bosch-Sijtsema, P.: From integration to composition: On the impact of software product lines, global development and ecosystems. Journal of Systems and Software 83(1), 67–76 (2010) (SI: Top Scholars)
[10] Küster, J.M., Engels, G.: Consistency management within model-based object-oriented development of components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 157–176. Springer, Heidelberg (2004)

[11] Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1) (March 2009) (online first: 3/2008)
[12] Mellor, S., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Reading (2004)
[13] Watkins, C.B.: Modular Verification: Testing a Subset of Integrated Modular Avionics in Isolation. In: 25th Digital Avionics Systems Conference, 2006 IEEE/AIAA, Portland, OR, IEEE Xplore (2006)
[14] Broekman, B., Notenboom, E.: Testing Embedded Software. Addison-Wesley, Reading (2003)
[15] AUTOSAR: Web page, `http://www.autosar.org/`
[16] Fennel, H., Bunzel, S., et al.: H.H.: Achievements and Exploitation of the AUTOSAR Development Partnership. In: Convergence, Detroit, USA (2006) (SAE 2006-21-0019)
[17] Richter, K.: On the Complexity of Adding Real-Time Properties to the AUTOSAR Software Component Model. In: Proc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany (October 2007)
[18] Fürst, S.: AUTOSAR - A World Wide Standard is on the Road. In: 14th International VDI Congress Electronic Systems for Motor Vehicles, Baden-Baden, Germany (October 2009)
[19] Lönn, H.: Far east: Modeling an automotive software architecture using the east adl. In: ICSE 2004 workshop on Software Engineering for Automotive Systems, SEAS (2004)
[20] Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The information bus: an architecture for extensible distributed systems. In: SOSP 1993: Proceedings of the fourteenth ACM symposium on Operating systems principles, pp. 58–68. ACM, New York (1993)
[21] Pardo-Castellote, G.: OMG Data-Distribution Service: Architectural Overview. In: International Conference on Distributed Computing Systems Workshops, p. 200 (2003)
[22] Allen, P. (ed.): The OMG's Model Driven Architecture. Component Development Strategies, The Monthly Newsletter from the Cutter Information Corp. on Managing and Developing Component-Based Systems, vol. XII (January 2002)
[23] Object Management Group: MDA Guide Version 1.0, Document omg/2003-05-01 (May 2003)
[24] Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., Chichester (1994)
[25] Awad, M., Kuusela, J., Ziegler, J.: Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion. Prentice Hall, Englewood Cliffs (1996)
[26] Douglass, B.P.: Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd edn. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (October 1999)
[27] Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley, Reading (January 2000)
[28] Bichler, L., Radermacher, A., Schürr, A.: Evaluation uml extensions for modeling realtime systems. In: Proc. on the 2002 IEEE Workshop on Object-oriented Realtime-dependable Systems, WORDS 2002, San Diego, USA, pp. 271–278. IEEE Computer Society Press, Los Alamitos (2002)

[29] Object Management Group: UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02 (September 2002)

[30] Gu, Z., Kodase, S., Wang, S., Shin, K.G.: A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada (2003)

[31] Masse, J., Kim, S., Hong, S.: Tool Set Implementation for Scenario-based Multithreading of UML-RT Models and Experimental Validation. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada (May 2003)

[32] Burmester, S., Giese, H., Tichy, M.: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 47–61. Springer, Heidelberg (2005)

[33] Burmester, S., Giese, H., Schäfer, W.: Model-driven architecture for hard real-time systems: From platform independent models to code. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 25–40. Springer, Heidelberg (2005)

[34] Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In: ICSE 2005: Proceedings of the 27th International Conference on Software Engineering, pp. 670–671. ACM Press, New York (2005)

[35] Object Management Group: UML 2.0 Superstructure Specification. Document: ptc/04-10-02 (convenience document) (October 2004)

[36] Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland. ACM Press, New York (2003)

[37] Giese, H., Burmester, S.: Real-Time Statechart Semantics. TechReport tr-ri-03-239, University of Paderborn (2003)

[38] Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology 1(1) (1997)

[39] Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-Time Systems. In: Proc. of IEEE Symposium on Logic in Computer Science (1992)

[40] Tichy, M., Giese, H., Seibel, A.: Story Diagrams in Real-Time Software. In: Giese, H., Westfechtel, B. (eds.) Proc. of the 4th International Fujaba Days, Bayreuth, Germany. Volume tr-ri-06-275 of Technical Report. University of Paderborn, pp. 15–22 (September 2006)

[41] Henkler, S., Oberthur, S., Giese, H., Seibel, A.: Model-Driven Runtime Resource Predictions for Advanced Mechatronic Systems with Dynamic Data Structures. In: Proc. of 13th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), May 5-6. IEEE Computer Society Press, Los Alamitos (accepted 2010)

[42] Burmester, S., Giese, H., Hirsch, M., Schilling, D.: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: Proceedings of the International Workshop on Specification and vaildation of UML models for Real Time and embedded Systems, SVERTS 2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML 2004 (October 2004)

[43] Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28th International Conference on Software Engineering (ICSE), Shanghai, China. (2006)
[44] Becker, B., Giese, H.: On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In: Proc. of 11th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), May 5-7, pp. 203–210. IEEE Computer Society Press, Los Alamitos (2008)
[45] Burmester, S., Giese, H., Gambuzza, A., Oberschelp, O.: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: Bobeanu, C. (ed.) Proc. of European Simulation and Modelling Conference (ESMc 2004), Paris, France, pp. 66–73. EOROSIS Publications (2004)
[46] Giese, H., Tichy, M., Schilling, D.: Compositional Hazard Analysis of UML Components and Deployment Models. In: Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.) SAFECOMP 2004. LNCS, vol. 3219, pp. 166–179. Springer, Heidelberg (2004)
[47] Giese, H., Tichy, M.: Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 156–169. Springer, Heidelberg (2006)
[48] Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE 91(1), 112–126 (2003)
[49] Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE 91(1) (January 2003)
[50] Wang, S., Shin, K.G.: Task construction for model-based design of embedded control software. IEEE Trans. Software Eng. 32(4), 254–264 (2006)
[51] Madl, G., Abdelwahed, S.: Model-based analysis of distributed real-time embedded system composition. In: EMSOFT 2005: Proceedings of the 5th ACM international conference on Embedded software, pp. 371–374. ACM, New York (2005)
[52] Krause, M., Bringmann, O., Hergenhan, A., Tabanoglu, G., Rosentiel, W.: Timing simulation of interconnected AUTOSAR software-components. In: DATE 2007: Proceedings of the conference on Design, automation and test in Europe, San Jose, CA, USA, EDA Consortium, pp. 474–479 (2007)
[53] Sangiovanni-Vincentelli, A.: Defining platform-based design. EEDesign of EETimes (February 2002)
[54] Horowitz, B., Liebman, J., Ma, C., Koo, T., Sangiovanni-Vincentelli, A., Sastry, S.: Platform-based embedded software design and system integration for autonomous vehicles. Proceedings of the IEEE 91(1), 198–211 (2003)
[55] de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
[56] Thiele, L., Wandeler, E., Stoimenov, N.: Real-time interfaces for composing real-time systems. In: EMSOFT 2006: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 34–43. ACM, New York (2006)
[57] Zhou, Y., Lee, E.A.: A causality interface for deadlock analysis in dataflow. In: EMSOFT 2006: Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 44–52. ACM, New York (2006)
[58] Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., Böde, E.: Boosting re-use of embedded automotive applications through rich components. In: Proc. of Foundations of Interface Technologies 2005, FIT 2005 (2005)
[59] Gössler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. 55(1-3), 161–183 (2005)

[60] Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in bip. In: EMSOFT 2007: Proceedings of the 7th ACM & IEEE international conference on Embedded software, pp. 11–20. ACM, New York (2007)

[61] Reichmann, C., Markus, K., Graf, P., Müller-Glaser, K.D.: Generalstore - a case-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In: ECBS 2004: Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems, Washington, DC, USA, p. 225. IEEE Computer Society, Los Alamitos (2004)

[62] Altheide, F., Dörr, H., Schürr, A.: Requirements to a Framework for sustainable Integration of System Development Tools. In: Stoewer, H., Garnier, L. (eds.) Proc. of the 3rd European Systems Engineering Conference (EuSEC 2002), Toulouse, AFIS PC Chairs, pp. 53–57 (2002)

[63] Karsai, G., Lang, A., Neema, S.: Design patterns for open tool integration. Software and System Modeling 4(2), 157–170 (2005)

[64] Giese, H., Hildebrandt, S., Neumann, S.: Towards Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization. In: 5th Workshop on Model-Based Development of Embedded Systems, MBEES (2009)

[65] Sztipanovits, J., Karsai, G.: Model-Integrated Computing. Computer 30(4), 110–111 (1997)

[66] Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty: Model-integrated development of embedded software. Proceedings of the IEEE 91, 145–164 (2003)

[67] Schulte, M.: Model-based integration of reusable component-based avionics systems - a case study. In: ISORC 2005: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, USA, pp. 62–71. IEEE Computer Society, Los Alamitos (2005)