7 Requirements Modeling for Embedded Realtime Systems

Ingolf Krüger, Claudiu Farcas, Emilia Farcas, and Massimiliano Menarini

University of California, San Diego, USA
{ikrueger,cfarcas,efarcas,mmenarini}@ucsd.edu

Abstract. Requirements engineering is the process of defining the goals and constraints of the system and specifying the system's domain of operation. Requirements activities may span the entire life cycle of the system development, refining the system specification and ultimately leading to an implementation. This chapter presents methodologies for the entire process from identifying requirements, modeling the domain, and mapping requirements to architectures.

We detail multiple activities, approaches, and aspects of the requirements gathering process, with the ultimate goal of guiding the reader in selecting and handling the most appropriate process for the entire lifecycle of a project. Special focus is placed on the challenges posed by the embedded systems. We present several modeling approaches for requirements engineering and ways of integrating real-time extensions and quality properties into the models. From requirements models we guide the reader in deriving architectures as realizations of core requirements and present an example alongside with a formal verification approach based on the SPIN model checker.

7.1 Introduction and Overview

Requirements engineering is arguably one of the most important and least-well-understood [1] development activities. It can have a positive effect on the overall development process – systems that actually provide value to their stakeholders, i.e. systems for which there exists a good understanding of what the requirements are, as well as a match between the system's requirements and the implementation, are generally considered a success if implemented within the available resources. At the same time, it is well-known that errors made during the activities that pertain to requirements analysis and management are hard to detect and costly to fix as time progresses through the development process.

In this chapter, we discuss the challenges and opportunities of the requirements engineering process for complex embedded real-time systems (ERS) as they arise in domains such as automotive, avionics, medical, communications and entertainment systems to name but a few examples. This system class is of high economic relevance and significant technical complexity – more than 98% of processors are "embedded" [2]. In automotive systems, for instance, up to 90% of all innovations are influenced by software-enabled electronics. In some high-end vehicles more than 60 different electronic control units (ECUs), interconnected using multiple communication bus technologies and hundreds of signals, together provide thousands of externally observable functions. Heterogeneity and distribution lead to high numbers of different configurations and variants. A wide functional variety from hard real-time safety critical engine control to comfort electronics and infotainment systems, long product life cycles, demanding time-to-market, and a strong need for competitive per-piece costs compound the technical challenges. All of these aspects are directly or indirectly related to the discovery, articulation, quality assurance and continued management of a highly diverse and interdisciplinary requirements set.

In the following paragraphs we elaborate further on the challenges of requirements engineering for ERS. This account draws heavily on our experiences with automotive systems [3, 4, 5, 6, 7]; furthermore, we provide an overview of the remainder of this chapter.

7.1.1 What's in a Requirement?

Before we can gain an understanding of why requirements engineering for ERS is challenging, we first have to identify what we mean by the term "requirement". We view a requirement as a documented need of what a product or service should be or do. A requirement also identifies the necessary attributes, capabilities, characteristics, or qualities that have value or utility to a stakeholder. While sufficiently intuitive at first glance, this definition leaves open what terms such as "attribute", "quality" and "value" mean concretely. Unless these terms are precisely defined, of course, it is difficult – if not impossible – to identify whether a given requirement is well articulated, let alone whether a given system correctly implements this requirement.

Nevertheless, this definition brings forward a number of important concerns for capturing and managing requirements. First, it is important to observe that requirements are connected to stakeholders [8]. For each requirement there should be an identified party with a vested interest in seeing the requirement implemented. This implies also that the requirements, collectively, need to articulate values of the stakeholders of the system under consideration. Stakeholders include (and are not limited to) the customer who commissions and accepts the system, regulatory bodies, marketing and productization entities, suppliers, integrators, developers, architects and maintainers, and end-users. Consequently, models, techniques and tools for documenting and managing requirements necessarily need to be able to reflect the various different views [9] that each stakeholder group brings to the table. For instance, marketing representatives may articulate requirements aspects that relate functionality with cost, end-users may articulate usability requirements aspects [10, 11, 12, 13], and maintainers may articulate requirements at readability of the source code from which the software sub-system is compiled.

To address different stakeholder views, the literature distinguishes various classifications for requirements. At the highest level, there is a distinction between *business*, *product*, and *process requirements*. Here, business requirements refer to specifications of what the business wants to achieve with a specific project. An example for a business requirement is "Offer the safest car on the road today". Requirements such as this are typically directly linked to an enterprise objective, such as "Being the world leader in safe vehicles" and associated business and marketing accounts. Product requirements often encompass the functionality and operational infrastructure that is required to implement the business requirements. This includes a specification of the functions as well as the hardware/software context in which they are to be implemented. Process requirements typically refer to how the development of the system under consideration is to come about, whether and how it needs to be certified, and what methods are to be applied during development for documentation, implementation and quality assurance.

Another traditional classification distinguishes *functional* from *non-functional* requirements. Here, functional requirements include those that determine what the system is supposed to do – this amounts to a specification of the *operational capabilities* provided by the system. Non-functional requirements [14, 15] (some authors also refer to them as *quality requirements*) then are defined as constraints at the implementation level of the functional requirements. Such constraints include product requirements[1] (usability [13], performance and efficiency [16, 17, 18], reliability [19, 20] and portability [21]), organizational requirements (delivery, implementation, adherence to standards and regulations) [15], and external requirements (interoperability [14, 22], ethics, and legislation). Functional requirements address the operational capabilities of the system, non-functional requirements define the context in which these capabilities come about.

Characteristics of ERS often blur the line between the functional and non-functional requirements. Consider, as an example, the air bag controller for a modern car. The requirement "Within 10 milliseconds after impact, the airbag is to be fully inflated" identifies an operational aspect ("after impact, the airbag is to be fully inflated"), and a performance constraint ("within 10 milliseconds after impact"). Clearly, the stakeholder group "driver and passengers" would rightly argue that an airbag that misses its deadline is not a functioning product. Hence, this requirement will likely be perceived as a functional requirement, despite its performance aspect.

As we shall see below, time plays a critical role in specifying requirements for ERS. Therefore, it often becomes part of the underlying system model in relation to which requirements are expressed. In other words, timing constraints become part of the operational capabilities of the system under consideration. Then, requirements with timing constraints naturally fall into the category of functional requirements. In general, however, it often makes sense to distinguish functional and non-functional requirements aspects, and to allow specification of both aspects in a singular requirement.

[1] Note that this refers to a subset of what we called product requirements in the preceding paragraph.

Independently from the chosen requirements classification scheme, the way in which we are able to articulate requirements has a significant impact on how useful requirements engineering is for all other development activities. We need to be able to determine effectively and efficiently, (a) whether stakeholder values and associated constraints at the system under consideration are captured accurately in the requirements, and (b) whether the implementation is faithful to the requirements as captured. (a) and (b) are commonly referred to as the requirements validation and verification problems, respectively.

For a comprehensive approach to requirements engineering for ERS, we need to be able to address all relevant requirements aspects of the system under consideration such that they can be effectively and efficiently validated and verified. This is a hard challenge in general and specifically so for ERS, as we will elaborate in the following paragraphs.

7.1.2 Why Requirements Engineering for ERS Is Hard

We now discuss key forces that influence the difficulty of eliciting and managing requirements for ERS; many of these forces interrelate. We have already pointed out that precisely defining what a requirement is, is a challenge in and by itself. A common solution for model-based engineering approaches is to create a *system model*, which is a mathematical representation of the central phenomena exhibited by the system class into which the concrete system under development falls. We can then formally define requirements as constraints at the system model. These constraints reduce the set of all possible instances of the model to those that fulfill the requirement.

We are interested in system models that are close to the problem domain. For this purpose, system models built on top of process algebras, timed and untimed, finite and infinite automata, temporal logics, partial orders, or streams and stream, relations, and games have been devised to address a broad spectrum of properties while placing bounds on the computational complexity of validation and verification. Hence, in requirements engineering, a key challenge is to identify system models and associated specification languages that allow representation of the key domain concepts [23] within the formal model such that validation and verification are effective and efficient. In the following paragraphs we discuss the key forces that drive the selection of an appropriate system model for ERS.

Requirements do change over the lifecycle of a product [24, 25, 26] [27]. The romantic idea that requirements can be captured completely at product inception, frozen, and then implemented to satisfaction is an illusion for all but the most trivial systems. Consequently, requirements need to be managed actively throughout the product lifecycle [28, 29], from inception to retirement.

Embedding – ERS are embedded into a context by definition. This context is typically another product, such as a car, an airplane, or a medical device. The ERS often plays the role of a controller of physical processes in which the overall product engages. This means that the ERS needs to interface with the context into which it is embedded, and thus needs to have a model of its environment to appropriately react to changes in this environment.

Furthermore, each ERS has a *unit cost* associated with it. There is a natural incentive to reduce the unit cost to reduce the overall product's cost (or to increase profit margins). Consequently, ERS are equipped with just enough computational and storage resources (as in hardware) to fulfill their desired function. Especially in mass markets, such as in automotive or entertainment electronics, where comparatively tiny unit cost savings have an enormous impact on overall profitability, cost reduction at the unit level is a critical and driving business requirement. Of course, if the overall product consists of multiple ERS, this can result in many local optimizations at the expense of realizing global cost savings across ERS. As we shall see, below, this is compounded by the distributed development model for ERS, and by the absence of integrated system models that would allow articulation and optimization of cost and functionality across component ERS.

Multi-Disciplinary Stakeholder Communities. Clearly, describing the interface between the ERS and its physical environment necessitates that requirements can express the properties and constraints of the requirements that are relevant for this interface. This alone already necessitates a multi-disciplinary approach to requirements engineering for ERS. A car, for instance, provides a physical context in which mechanical engineers are key domain experts. Add to this electrical engineers for the hardware context of the ERS, as well as Human-Machine-Interface (HMI) experts for the usability aspects, to obtain an initial set of disciplines involved in ERS development besides the requirements engineers and software developers, testers and maintainers with computer science background. Each of these stakeholder groups has a different view [30] on the system under consideration, and has its own domain concepts and associated ways of articulating them. This necessitates that the chosen system model allows expression of requirements from all relevant stakeholder groups.

Two techniques allow dealing with the resulting complexity: (1) introducing mechanisms for expressing different views onto, or abstraction levels of the same system model – in the language of the respective stakeholder group, and (2) enabling the co-existence of multiple different system models, each of which is tailored to a particular stakeholder group, or view; then the challenge is how to mediate between requirements specified in the respective system models to arrive at a consistent [31, 32, 33, 34], integrated requirements specification for the overall product. This mediation can take on various forms; typically it will consist of an elaboration of the domain entities (or abstractions thereof) shared by multiple source models and the key relationships between these entities, as well as projection relations between the source models and the mediation model.

Consistency and Realizability – If there is more than one requirement needed to specify a system of interest (and for all but the most trivial ones, there is), we have to address consistency between requirements. The underlying question is whether the requirements as specified allow any system to be constructed such that all requirements are fulfilled.

For instance, if we represent requirements as predicates on the underlying system model, then we can express relationships between requirements using logical conjunction and disjunction: conjunction expresses that of two requirements both must be satisfied together; disjunction indicates alternatives among requirements. If we add negation as a logical construct, we can also express "anti-requirements", i.e. requirements that must not be fulfilled [35]. This then allows us to express conditional requirements of the form "if requirement A is fulfilled, then requirement B must be fulfilled also". The overall requirements specification can then be interpreted as a logical formula involving predicates and the mentioned logical connectives. The question then arises whether the set of tape valuations constrained by this logical formula has any elements in it.

As we have described in the preceding paragraph, we have to be able to represent different views on the set of requirements to different stakeholder groups. Consequently, we have to concern ourselves with the consistency of the resulting composite views. We have to add requirements that further restrict the set of possible models to those that are realizable.

Outsourcing and Distributed Development – ERS are, in general, developed in the interplay between an Original Equipment Manufacturer (OEM) and a supplier. The OEM is responsible for the product into which the ERS integrates. The supplier is responsible for the ERS. This necessitates precise and expressive requirements specifications [36] that elucidate the interplay between the ERS and its environment. In practice, requirements are seldom expressed precisely enough, successful projects resort to cooperative and joint development between the OEM and the supplier to ensure short feedback cycles to iteratively refine the requirements. The precision to which we are able to articulate requirements between OEM and supplier has a direct impact on the number of iteration cycles needed, and the ability for both parties to verify and validate the requirements they were able to specify.

Furthermore, the distribution of responsibility [37] between the OEM as the system integrator, and the supplier has lead to two distinguishable levels of abstraction in requirements specification, called *user* and *system requirements*, respectively. User requirements are gathered by the OEM and articulate the OEM's expectations at the outcome of the supplier's development efforts. The supplier responds to a user requirements specification with a system requirements specification that details the user requirements, and incorporates further business, product and process requirements from the supplier's point of view.

Multi-Functionality [7] means that an ERS provides not one singular, but multiple distinguishable, individually valuable functions (also called *features*) to their environment. For instance, cell phones provide calendaring, email and a wide variety of productivity and entertainment functions in addition to the all but mundane functions of placing and receiving calls.

This necessitates that requirements for ERS explicitly address the partiality of individual functions *and* precise specifications of how the individual functions integrate into the whole. In particular, the requirements need be be explicit about desirable and undesirable *feature interaction.* A desirable feature interactions emerges from the interplay between two features to the (sometimes unforeseen) benefit of a stakeholder. Undesirable feature interactions, on the other hand, reduce the value of the integrated system to a stakeholder. Again, this calls for explicit means to determine consistency among requirements for the ERS and the environment into which it is embedded.

Heterogeneity – Requirements for ERS that control physical processes often are most succinctly represented in terms of the mathematical models that are used to describe the physical processes. For instance, requirements for automotive systems need to capture the vehicle's continuous movement through space and time. The adequate mathematical model for this movement will involve differential equations. Specifically, the field of *control theory* was developed precisely to study the phenomena that arise in the interactions between the physical world and controllers that seek to influence the environment to effect a desirable condition. Any comprehensive requirements specification technique that attempts to be successful in the automotive domain, therefore, needs to be able to capture continuous behaviors (in the underlying mathematical models) to facilitate interaction with control engineers. At the same time, a car is a good example for the need to also express mixed discrete-continuous and purely discrete ERS [7].

Distribution and Integration – As mentioned above, the OEM typically acts as the integrator of a set of independently developed ERS. Consequently, the desired behavior for the integrated product emerges from the interplay of the functionality provided by the sub-systems. For the system models underlying requirements specifications this means that they need to be able to express phenomena of concurrency and synchronization. Depending on the product and the OEM, these ERS are developed by a variety of different, competing suppliers. As a consequence, the functionality valuable to the end-user is scattered across a wide variety of subsystems. This places a tremendous integration challenge on the OEM – this finds its expression today in intense and costly integration, calibration and testing activities in which the OEM engages when *all* subsystems finally are available.

Often, there is no overarching requirements specification addressing the integration challenge – the user requirements suppliers see are then underspecified in terms of these integration requirements, and the OEM has to work around the resulting implicit assumptions the suppliers make. Furthermore, as mentioned before, the absence of an overarching understanding of the integration requirements results in poor resource optimization *across* the ERS of an integrated system. Consequently, the requirements models for ERS should explicitly address the scattering of functionality and the resulting integration requirements, as well as the concerns that cut across the individual system components (see below).

Safety-Criticality – ERS in safety-critical products [38, 39, 40] such as cars, airplanes, trains, space ships, power-plant and factory control systems, heart-pacers and other medical devices are safety-critical by association. Much research has been invested into developing system models that allow the specification and verification of safety properties [41, 42, 43, 44] [45, 46, 39, 47].

A remaining research challenge is to provide *domain-specific* system models that allow articulation, validation and verification of safety requirements at the scale of thousands of integrated functions while resolving the dependencies and interactions between the requirements forces described in this section.

As a case in point we note that failure management is a critical concern for many ERS [19, 48] and specifically so in the automotive domain. Yet, none of the widely-used requirements specification techniques for automotive systems even recognizes the notion of failure as a first-class modeling entity. Of course, there are techniques such as Failure Mode and Effect Analysis (FMEA) [49] and Fault Tree Analysis (FTA) [50] – however, these techniques are rarely applied at the inception and requirements modeling phase, but rather reserved for an after-the-fact analysis, when the subsystems have already been developed. We will pick up this topic below, when we discuss cross-cutting concerns, as well as in Section 7.4, in our case study.

Multi-scale Timing, Asynchronous vs. Synchronous Communication – Time plays a critical role in ERS. Many models of physical phenomena depend on time as a variable; for instance, velocity is the derivative of position in time. ERS control properties are consequently frequently specified in the language of differential equations. However, this is typically already at the level of a solution (in the sense of specifying a particular controller) rather than at the level of a requirement. This is facilitated by the available tool support for control system development (see below), which favors the graphical specification of particular solutions rather than requirements.

In general, most system models favor a particular model of time (continuous vs. discrete), which results in awkward requirements specifications for systems with mixed discrete-continuous timing properties. Similarly, most system models favor a specific communication model (message/time-synchronous vs. message/time-asynchronous), which again can result in awkward expressions of requirements for integrated systems with mixed types of communication requirements. Of course, in concrete examples, such as cars, we find a broad range of timing constraints on scales of milliseconds (motor control) to tenths of seconds (comfort functions) to seconds (navigation). Similarly, we find a wide variety of communication mechanisms ranging from time- and message-synchronous communication within an ERS to time- and message-asynchronous communication beyond vehicle boundaries for remote operation of vehicle functions.

It is one of the key research challenges in ERS to reconcile multiple time and communication models within such that the corresponding requirements can be expressed lucidly. In practice, the timing requirements are often only informally stated on a per-component basis, following some intuitive, or implicit understanding of overall end-to-end response-time requirements, or implementation-technology-dependent constraints (processor cycle times, cache hit-rates, communication bus throughput and latencies). Using simulation and testing as the main tools, the system is then instrumented to determine worst-case execution times. The results are then matched against the per-component timing requirements.

Long Product Life Cycles – Products containing ERS typically have long product life cycles from inception to retirement. This means that over this lifecycle, many changes in the environment of any particular ERS may occur: other components may be exchanged, or the product may be placed into previously unanticipated environments. This, in turn, means that requirements specifications (and thus the system models supporting them) need to be durable and accessible throughout the product lifecycle.

Time-to-Market While product life cycles are long, especially in consumer mass-products, time-to-market is constantly under scrutiny for reduction to react more rapidly to changes in consumer, environmental or regulatory needs. A case in point is the rapidly increasing demand for hybrid vehicles on the backdrop of rapidly rising fuel prices.

This impacts requirements engineering for ERS in the sense that it needs to be able to respond to rapidly changing requirements to facilitate the agility and flexibility needs of its container. In particular, the degree to which requirements can be specified in a modular fashion will have significant impact on how rapidly the parts of an integrated system of ERS can be adapted to support changes to the product as a whole. Note that here, we are referring to the structuring of the requirements, rather than the structuring of the resulting architecture; the latter is also an important, albeit separate, topic.

Product Lines and Re-Configuration – Similarly, to amortize costs and respond to market needs, OEMs often develop platform strategies and product lines so as to be able to reuse significant parts of an integrated system, while adapting others. This leads to the challenge of managing multiple different versions of requirements sets, which correspond to multiple different configurations of the integrated system. In the automotive example, some product families have variation points amounting to hundreds of thousands of different configurations customers can order. Some of these configuration options are necessitated by regional laws and regulations, others stem from different options for feature sets of the vehicle.

Again, the major impact for requirements engineering is on the management capabilities of the associated requirements documents, so as to avoid costly rework; also, there is impact on the ability to verify consistency of the requirements configurations.

Influence of Hardware Architectures – Sometimes, products containing ERS evolve from limited feature sets to thousands of software-enabled features; again, the automotive domain is a telling example. From the beginnings of the use of electronically controlled fuel injection to today the amount and impact of software deployed in the car has grown exponentially. Now automotive engineers are faced with the challenge of integrating, and supplying power for, 30 to 80 Electronic Control Units (ECUs) per car, depending on the target market (budget vs luxury). This challenge includes calibrating the timing between the various ECUs in their attempt to communicate with other ECUs in the vehicle – largely due to the scattering of functionality across the various ECUs. To some degree, this challenge is an artifact of dominant legacy architectures that were adopted initially, and never reconsidered as more and more functions entered the vehicle – often due to long-time licensing agreements and cost savings of reuse.

The impact on requirements engineering for ERS is that such legacy architectures often become requirements constraints that also need to be articulated in the requirements model. At the same time there is a tradeoff between articulating legacy architecture requirements and writing requirements that describe more of the “what”, rather than a particular “how”. This tradeoff must be resolved at the level of the overall engineering process.

Deployment, Update, Diagnostics, and Maintenance – Because of their embedding, ERS are less accessible for deployment, update and maintenance tasks than desktop or laptop computers. Nevertheless, long product life cycles ultimately necessitate updates to the software or hardware components of an ERS. Especially in distributed, integrated ERS such as in cars and airplanes, however, updates are (as of this writing) difficult to deploy. As mentioned above, in such systems many ERS originate from different competing suppliers, each applying their own strategies, technologies and methods for deployment, update and maintenance (if any). This again points to the specification of dedicated requirements for these important development tasks at the integration level.

Therein lies significant research potential – today’s formal requirements specification techniques have yet to broaden the range of requirements types they address. Most system models (as we shall see in Section 7.2) assume static system structures and mappings from behaviors to these system structures. This renders precise specification of deployment, update and maintenance requirements all but impossible.

The drive towards service-oriented architectures (SOAs) that has gained significant momentum in the world of business information systems is slowly gaining ground in the ERS domain as well. One of the fundamental premises of SOAs is that the location of a function is secondary to its interface due to static and dynamic advertisements, registration and binding techniques. This dynamicity today still clashes with the imperative of unit cost savings, and thus scarce resources per ERS. As attention shifts from per-unit development to integrated networks of ERS, we expect this reservation to give way to an understanding of global reuse, dependability potentials and cost-savings.

As technical solutions for these concerns are on the doorstep so should be the models for specifying the corresponding requirements – yet, there is very little support in contemporary, widely accepted requirements modeling and engineering approaches.

Quality and Cross-Cutting Concerns – Deployment, update and maintenance are good examples of cross-cutting ERS concerns that have little to no support in today’s system models and corresponding systematic requirements engineering techniques. Of course, the list of cross-cutting concerns does not stop there. Other important ones are availability, fail-safety (across units in a set of networked ERS), security [51, 15], and policy/governance. Availability and fail-safety are addressed in this and other chapters explicitly, hence we focus on the other two requirements aspects here.

Security has been regarded as a secondary concern for a long time in ERS development. After all, most ERS were assumed to be inaccessible from outside of the product they were embedded into. This, of course, has changed radically with the increased networking among and beyond ERS [52] – suddenly, for instance, we find that the CLS functionality of a car is accessible via the Internet (to support remote unlocking to recover locked-in keys). Signals in car networks today are rarely encrypted, and can thus be reengineered, and reproduced in malevolent ways (for instance to gain unauthorized entry into the vehicle).

Policy and governance also come into play in networked ERS – together they address the question under which circumstances a party can (or must) perform a particular action in the system. In our automotive example this becomes important in identifying who has the authority to unlock the vehicle; another scenario is the prevention of unauthorized after-market components to participate in the exchange between the authorized ERS.

Similarly, the challenge of *diagnostics* – what and how much data to collect at what locations and time points, to identify the root cause of a failure during system operation – is an area of active research with little explicit system modeling support, and consequently no broadly accepted formal requirements specification techniques.

Furthermore, all ERS undergo a set of distinct operational modes, such as *initializing*, *idling*, *operating*, *resetting* and *suspending*, to name a few examples. From a modeling point of view this can be addressed with regular state-based modeling techniques such as state machines or activity diagrams. However, there is a need to explicitly provide access to all or some of these modes at the ERS-environment interface, especially in a networked ERS consisting of multiple subsystems. This need arises both for monitoring purposes and to ensure that sets of components can be steered into defined operational modes together (say, for start-up, shut-down, and testing).

Traceability – Because the requirements spectrum of ERS is vast and highly heterogeneous, traceability becomes a particularly daunting task. Success is again bound to our ability to articulate requirements at increasingly high levels of detail, and to validate and verify requirements at different levels of abstraction against each other. Furthermore, methods are needed that establish a trace between architecture specifications and implementations at various levels of abstraction and the requirements that are implemented at these levels. This challenge is again compounded by the distributed nature of the OEM-supplier relationship, and the desire to support product lines with vast numbers of possible system configurations, as well as by the tight coupling between requirements specifications and the target hardware/system platform onto which the ERS functionality is to be deployed.

At the level of system architecture specification, Model-Driven Architecture (MDA) [53] has taken a step into a more tractable direction – here, we distinguish between a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM can largely be regarded as a highly detailed requirements model that captures the core system entities and their interactions without specifying how these are implemented. The PSM, on the other hand, captures all aspects of the deployment architecture. Then a mapping between a PIM and multiple

PSMs can be be established to capture multiple different deployments for the same functionality set. However, further research is needed to lift the degree of abstraction from PIMs to genuine user and system requirements specifications.

Tool Landscape – A wide range of commercial and academic tools for requirements engineering and management have been developed. Few, however, cover even a small subset of the concerns we have brought forward in the preceding paragraphs to any degree of satisfaction. We attribute this largely to the absence of comprehensive system models and associated requirements specification techniques and standardized architectures that adequately capture and reduce the complexities of ERS specific to particular domains.

Of course, tools such as DOORS [54], Rational RequisitePro [55], and Cradle [56] have displayed their utility in the *management* [57] (as in organization and version control) of requirements once they are elicited. Tools such as Matlab/Simulink/Stateflow allow detailed architecture design of and even generation of efficient code for controllers for which requirements are well understood. However, the challenge of finding adequate system models and requirements specification techniques for systematic requirements discovery and refinement remain. Yes, DOORS integrates with other UML-based tools for requirements elaboration. However, for UML and its derivatives a wide variety of the challenges posed above are unsolved as of yet; we name just a few examples: consistency of description techniques and resulting requirements specifications, efficient and effective validation and verification at the model level, notations and models for system (re-)configuration, support for cross-cutting concerns in the requirements models, including failure, safety, security, and policy/governance

7.1.3 Summary and Outline

In the preceding paragraphs we have identified a broad range of challenges that render precise requirements specifications of ERS particularly difficult. We have started by identifying requirements as the expression of stakeholder values, and have established a connection between mathematical system models and requirements formalized as predicates (or constraints) over these system models. Then we have called out a number of requirements aspects that a comprehensive requirements engineering approach needs to be able to articulate and manage throughout the development process. Key challenges arise from the multi-disciplinary and heterogeneous nature of ERS requirements, their distribution, domain-specifics such as a broad range of timing specification needs, deployment, update and maintenance requirements and the associated quality, validation and verification concerns.

No currently available tool or integrated tool set addresses all of these concerns comprehensively. We conclude that this necessitates further research and development in both academia and industry – this volume is evidence of the significant research progress to date.

The remainder of this chapter is structured as follows: In Section 7.2 we review a broad range of requirements engineering techniques proposed in the literature – this provides an overview to what degree the mentioned concerns are addressed in today's models and techniques. In the absence of a formal, comprehensive requirements engineering technique, we briefly recall key best-practices of requirements engineering and how they relate to model-based development (Section 7.2) for ERS. We discuss the relationship between requirements and their traceability to architecture, and from there to implementation, in Section 7.3. In Section 7.4 we give an example for capturing safety requirements of an automotive Central Locking System using structural and behavioral modeling techniques so that these requirements can be formally verified.

7.2 Requirements Specifications and Modeling for ERS

Modeling plays an important role in all requirement engineering activities, serving as a common interface to domain analysis, requirements elicitation, specification, assessment, documentation, and evolution. Initially, domain models are created to describe the existing system for which the software should be built, covering stakeholders, human actors that interact with the system, hardware devices, and the environment in which the system will operate. In addition to behavior, domain models define "the language" of the system by capturing domain entities in a structural way [58]. Then, deficiencies in the existing system and objectives for the target system are more clearly identified. During requirements elicitation, alternative models for the target system are created, which may define different boundaries between the target system and its environment. Models can help in defining the questions for stakeholders and surfacing hidden requirements. Ultimately, the requirements have to be mapped to the precise specification of the system and the mapping should be kept up to date during the evolution of requirements or the architecture.

After requirements are specified (more or less formally), the specifications are checked for errors such as incompleteness, contradictions, ambiguities, inadequacies in respect to the real needs – which all can have disastrous effects on the system development costs and the quality of the resulted product. The choice of modeling notations is often a tradeoff between readability and powerful reasoning techniques: natural language is very flexible, useful for communicating requirements, but can not capture relationships and is often an expression of subjective reasoning [59, 60]; applied/semi-formal models (e.g., entity-relationship diagrams, UML diagrams, structured analysis) typically have a graphical representation which is very useful when communicating with stakeholders and often offers simulation and animation capabilities; and formal notations (e.g., KAOS [61, 62], RML, Telos, SCR [63, 28], process algebra, Promela/SPIN [64]) capture precise semantics, which supports rich verification techniques.

7.2.1 Requirements Models

Many challenges of requirements engineering span multiple application domains. For instance, business concerns such as conflicts from multiple viewpoints over requirements of different stakeholders are present in domains as diverse as business information systems, financial applications, avionics, car OEMs and suppliers, etc. Hence, in this subsection we first briefly present general techniques for requirements modeling that have a broader scope and can be applied on a variety of domains. As embedded systems may require dedicated techniques for some aspects such as timing, determinism, and formal verification of safety properties, we then describe particular techniques for ERS.

Business modeling Goal-based approaches such as KAOS [61, 62] and i* [65, 66] focus on modeling goal hierarchies to capture the objectives of the system, the associated tasks, and resources. The explicit modeling of goals helps in checking the requirements completeness – the requirements are complete if they are sufficient to meet the goal they are refining [67]. In KAOS [61, 62], the set of high-level goals are iteratively refined using AND/OR decomposition, obtaining a graph structure. KAOS allows to define agents and the actions they are capable of, and the goals can be operationalized into constraints assigned to individual agents. Each term is formally defined in temporal logic; therefore, a main contribution of KAOS is to prove that goal refinement is correct and complete [68], which implies proving that requirements correspond to system goals. Furthermore, [69] shows how conflicts between goals can be formally detected. The i* [65, 66] framework focuses on two models: the strategic rationale model describes the goals of the actors and the interactions between goals and tasks within each actor, whereas the strategic dependency model focuses on the relationships between actors such as dependencies on the goals or resources from other actors, or dependencies on tasks that other actors should perform. With such models, properties such as viability of an agent's plan or the fulfillment of a commitment between agents can be verified.

Another approach is to focus on business processes (workflows), business rules, and the services the system provides [70]. For this purpose, UML activity and collaboration diagrams can be used to show how actors collaborate to perform tasks. Moreover, UML class diagrams can show the roles of actors within the domain and can be used to capture business rules, although often in an implicit way through the class composition and multiplicity constraints. In UML, business rules, as well as pre- and post- conditions, can be explicitly specified in Object Constraint Language (OCL) [71].

Modeling information and behavior is an important part of the requirements specification process dealing with the structure of the system in terms of entities and their relationships; the behavior in terms of states and events that determine state transitions; and interactions in terms of communication patterns, dataflows between system components, parallelism, concurrency coordination, and dependencies – especially temporal dependencies in the case of ERS.

One way for specifying the structure of the systems is to use entity relationship (E-R) diagrams to capture domain concepts and data models. Although E-R diagrams are just notations, the concepts of objects, classes, attributes, and instances map well to domain entities and enable an easy transition to object-oriented system design. This ease of transition from requirements to design is sometimes a drawback as it becomes difficult to distinguish the real user requirements and their rationale from design decisions inferred from underspecified requirements. Also, focusing on single use cases may prevent the development of the system vision or the "big-picture". In such cases, the solution resides in operating with partial system specifications through an agile development process that iteratively refines the requirements and constructs the vision of the final system. Standards such as UML can be used to achieve consistency between models developed in different iterations.

Modeling the system behavior is generally accomplished using variants of finite state machines (FSM) [72, 73] and notations such as Dataflow Diagrams (DFD) [74]. The Structured Analysis is a data oriented approach for conceptual modeling initially intended for information systems and later adapted to ERS. It presents a development/transition path from an indicative model of the current system to an optative model of the new system. This methodology facilitates communication between stakeholders and system builder as it does not require software development expertise and can be easily used in domain terms. Abstractions and partitioning of the system into subsystems with clear boundaries make it easier to handle larger projects. However, a major drawback comes from the confusion between modeling the problem that the system is intended to solve and modeling the actual solution. Also in particular for ERS, the timing aspects are mostly invisible in the system model, making later tracing between the system behavior and its requirements a difficult task [75, 76].

Several variants of this approach exist, Structured Analysis and Design Technique (SADT) [77], Structured Analysis and System Specification (SASS) [74], Structured System Analysis (SSA) [78], Structured Requirements Definition (SRD) [79]. SASS is the closest relative of the classic structured analysis technique. SADT is a semi-formal technique supports the formalization of the declarative part of the system, but uses natural language for the requirements themselves. It provides a data model linked through consistency rules with a model for operations. It also uses activity diagrams instead of dataflow diagrams and distinguishes control data from process data. SSA uses a notation similar with [74], but adds data access diagrams to describe contents of data stores. SRD introduces the idea of building separate models for each perspective and then merging them.

Specific Requirements Models for Embedded Systems A wide range of real-time systems encountered in industrial environments, power plants, cars, airplanes, can be modeled and reasoned about as "embedded systems", because of the role of the computing system in controlling a physical process and the integration of the two aspects of "controlling" and "controlled" into a common system [80].

Modeling the requirements for embedded systems is crucial to be able to verify their behavior. Correcting requirements errors, under-/over- specifications, or similar imprecisions later in the development cycle can be extremely expensive [81, 82]. "The importance of determinism cannot be overestimated; deterministic systems are one order of magnitude simpler to specify, debug, and analyze than nondeterministic ones." [83]. Hence, formal models for specifying the requirements of ERS try to prevent costly errors [43] or that may ultimately lead to accidents.

SCR Tabular notations [84] have been used for decades to specify requirements for readability reasons. The *Software Cost Reduction* (SCR) requirements methodology [63, 28] was introduced for engineers working on the software for embedded systems. It was later refined for complete systems to incorporate both functional and nonfunctional requirements [85, 86, 87]. The method promotes a tabular notation for specifying requirements, a finite state machine model, and special constructs for expressing constraints such as modes, terms, conditions, events, inputs and outputs [63]. The method associates a table for each output, term, or mode class of the specification and enables system decomposition into smaller, more manageable parts.

Faulk's [88] initial formal foundations of this method use various classes of tables as total functions and mode classes as finite-state machines defined over events. There are *monitored* and *controlled* variables and *input* and *output* data items (provided by external devices such as sensors and actuators), where a monitored variable reflects the effect of the environment on the system behavior and a controlled variable reflects the control of the system on some environmental aspect. Events denote changes of value in the entities forming the system, where input events are trigged by the environment, whereas conditional events may also be triggered by internal system computations.

The Four-Variable Model [85] extends this method to systems by including critical aspects of timing and accuracy as mathematical relations on monitored and controlled variables. For complex systems several mode classes may operate in parallel. [86] introduces another similar abstract model. A specialized form [89, 90] of the Four Variables model is used as formal foundation for a tool suite [91, 92] consisting of a specification editor to create and maintain specifications, simulator for symbolically executing the specified system, automated consistency checker [93], and verifier for critical properties such as timing [90, 94]. These tools enable the developer to ensure proper syntax, type correctness, completeness of variable and mode class definitions, mode reachability and proper setting of initial values in all modes, disjointness (i.e., unique defined entities), coverage and acyclic dependencies.

The CoRE methodology [95] tries to address the shortcomings of its SCR ancestor, namely the lack of structuring mechanisms for variables (e.g., aggregation or generalization), models (e.g., and/or decomposition), and tables (e.g., refinement relationships). [96] proves the scalability of the approach in the context of large-scale avionics systems. [97] provides a practical comparison between SCR and CoRE within the context of a flight guidance system.

Requirements State Machine Language (RSML) [98, 99, 100] is a formal state-machine based hybrid approach using both tabular and graphical notations borrowed from Statecharts [101]. It introduces boolean tables and guards to describe state transitions in one or more high-level state machines that can communicate directly with each other. RSML tables describe transition conditions based on input events and may generate as result output events. Modes are defined explicitly as functions of input variables. The approach employs a state-based black-box model for all system components and their interfaces, which separates the specification of requirements from design aspects and enables formal analysis of the entire system its correctness and robustness [98].

[102] has a similar approach with tables and state machines but uses trace semantics for system analysis. Other specification languages such as Statemate [103], Hatley [104], Ward [105], include various models, yet not all of them are formally defined to enable automatic analysis and behavior verification. ProCos [106] provides a similar language but uses process algebra for the system model.

UML for Embedded Systems – UML can be used at different levels of the development process, especially for requirements modeling and functional design [107]. The high-level models of the system specify the requirements for behavior, domain structure, and QoS properties. The advantage of UML is its capability of modeling both system structure and behavior, specifically the structure of the problem domain and the interaction and collaboration between different agents in the system.

The profile mechanism in UML allows to define families of languages targeted to specific domains and levels of abstractions. For example, [108] presents a UML profile for a platform-based approach to embedded software development using stereotypes to represent platform services and resources that can be assembled together. Standardization activities under OMG include SysML [109] and MARTE [110], a new UML profile for modeling and analysis of Embedded Real-time Systems, in addition to the existing UML profile for Schedulability, Performance and Time [111]. UML currently supports the specification of timing and performance requirements, and could be extended to support also other QoS requirements such as for power consumption and cost.

Several embedded systems require more than one model of computation to reflect the nature of the application domain, whereas UML supports only event-based models. Therefore, several proposals have been made to extend UML: [112] introduces support for continuous-time by using stereotypes to represent continuous variables, time, and derivatives; [113] extends UML with a programming language for hybrid systems; and D-UML [114] introduces a dataflow mechanism (distinguishing between signal ports and data ports) coupled with mathematical equations in UML/Realtime.

SysML [109] customizes and re-uses a sub-set of UML concepts for systems engineering applications. It tries to be a cross-domain solution for modeling entire systems, without making domain-specific description languages obsolete. The SysML "block", which abstracts the software details in UML classes, is a significant extension in the direction of modeling complex ERS, where software is just one aspect besides electronics, mechanics, etc. Blocks can be used to decompose the system into individual parts, with dedicated ports for accessing their internals. SysML also adds requirements modeling as a key aspect of the system development process. It provides requirements diagrams, tree structures, or tables, which not only support the documenting requirements process, but also provide traceability to requirements throughout the design flow, ensuring that requirements are satisfied. SysML groups behavior, structure, analysis, and requirements in a single, integrated system model. It also supports extensions for guarding the information flow and the entities of the system. SysML is an improvement over UML in that it allows to articulate requirements concerns relevant at the system engineering level, including function networks, and requirements allocation to subsystems. However, both UML and SysML lack the binding to a concrete system model that enables formal analysis of requirements and their associated models. Also, there is still too little support for a seamless transition between requirements development and other development activities.

7.2.2 Programming Models

The observable behavior of the ERS is greatly influenced by the underlying programming model used for their construction, which plays a significant role in engineering the system requirements. High-level requirements are decomposed into requirements for individual software components according to the constraints supported by the programming model. For example, the requirement that a vehicle must stop within a given time frame since the driver pressed the brakes may translate into deadline requirements for several tasks and messages. Hence, the interaction between the ERS and its environment is governed by two different views over the notion of *time*: the stakeholders provide requirements in terms of *environment time*, whereas the system is implemented in terms of *software time.* The environment time represents the continuous time flow observable from the external environment of the ERS (i.e., wall-clock time). On the other hand, the software time is a discrete time flow of the ERS itself measured by the number of occurrences of some events such as the pulses of the CPU clock. [115] identifies three real-time programming models: synchronous, scheduled, and timed model.

The Synchronous model assumes that the ERS performs all computation and communication instantaneously [83, 116], and can always keep pace with the environment. This assumption imposes great constraints on the system requirements as an infinitely fast computer is not achievable in practice. Hence, verifying the ERS model through simulation may fail to show that, in practice, the response time of the implemented ERS may still be far from "atomic" and present output jitter.

Depending on the functional requirements of the ERS, existing synchronous languages can be classified under two categories: control-flow and data-flow oriented. The control-flow oriented languages are also imperative languages and are adequate for control-intensive applications such as communication controllers, real-time process control. Esterel [83, 116] has high-level, modular constructs that lead to a real structure of reactive programs based on the semantics of the finite-state Mealy machine. Statecharts [101] has a graphical formalism and it is not fully synchronous. Argos [117] simplifies the formalism of Statecharts and provides full synchrony. The data-flow oriented languages (also known as declarative languages) are appropriate for data-intensive applications such as digital signal processing and steady stream process-control applications. Lustre [118] is a declarative language that supports only the data-flow systems that can be implemented as bounded automata-like programs.

The synchronous approach is used in modeling tools such as Scade [119, 120], which supports the development of real-time controllers on non-distributed platforms or distributed platforms like the Timed-Triggered Architecture [121]. The Scade suite supports the design of continuous dataflows (based on Lustre) with discrete parts realized by a state-machine editor (based on Esterel). The computational models are compatible by transforming values and signals [119]. The Scade Suite is used by Airbus for the development of the critical software embedded in several aircrafts.

The Scheduled model relies on the classical scheduling theory for real-time programming. Functional requirements can be easily accommodated as the ERS may be implemented using sequential languages (e.g., C/C++), or a parallel programming language (e.g., Ada, Occam, CSP, RT-Java). Sequential languages lack concurrency and require a real-time operating system (RTOS) for inter-program communication and synchronization. Parallel languages support concurrency and communication as first-class concepts and typically have specialized run-time support systems. In this model, the software time is no longer an abstract notion equal to zero, but an unpredictable run-time variable influenced by the CPU speed, scheduler, utilization level, etc. Hence, schedulability analysis is necessary to guarantee that all computations complete in the allocated time.

There exist several UML compliant modeling tools that support code generation to C/C++, Java, Ada, different RTOS, and CORBA. UML provides a modeling framework for architecture description and behavior descriptions, but it is still work in progress to properly include real-time aspects in UML 2.0. The first step was made with the UML Profile for Schedulability, Performance, and Time [111]. Predictability analyses include the control flow analysis for sequence diagrams.

The Timed Model abstracts from ERS platform and the software time is always equal with the environment time, such that all computations and communication activities take a fixed logical amount of time, assuming that there is enough soft time to perform the computation under the real-time constraints imposed by the environment. The compiler of the specification language has to verify the time-safety of the computation and guarantee that there is enough software time to complete the computation before its deadline.

There are just a few examples of languages supporting this model, most of them based on the *Logical Execution Time* abstraction introduced by Giotto [122] – a task has a release time when it reads the inputs, and a terminate time when it provides the outputs to the environment. Within this time-span, the way the task executes on the target platform is irrelevant for the environment. Giotto is a high-level time-triggered language, which decouples the timing and functionality aspects, and abstracts from the execution platform. As a meta-language, it describes the intended temporal behavior of a system and expects its functionality as being externally implemented in a general-purpose programming language such as C, Oberon, or Java. XGiotto [123] extends Giotto to support event-driven programming, while preserving the benefits of the timed-model with fixed response-time.

The Timing Definition Language (TDL) [124, 125] adds component support and abstracts from the distributed platforms. It provides a complete tool-chain fully integrated in the Matlab/Simulink suite. The developer can model the functional aspects of the ERS in Simulink and the timing aspects in the integrated TDL visual editor. The ERS can then be verified through simulation, which based on the timed model provides an accurate representation of the ERS behavior. The TDL compiler is extensible through plug-ins such as the bus schedule generator that enables automatic scheduling [126, 127] of the communication in a distributed system. Hence, components can be developed independently regardless of their distribution – this is the so-called transparent distribution [128] feature of the language that preserves the time and value determinism of the application regardless of how its components are deployed in a distributed solution.

7.3 Requirements Engineering Approaches: Processes and Practices

In the preceding two sections we have established why requirements engineering for ERS is inherently difficult, and have surveyed some of the techniques and tools available in the literature to address this difficulty from various angles. From this overview it becomes clear that much progress has been made and many research challenges remain in this important field. In particular, there is "no silver bullet" in sight, nor is there one to be expected. Requirements engineering demands a holistic view on the problem at hand to address the challenges we have brought forward in Section 7.1. In this section, we recall a few of the practices that have emerged in collecting requirements for ERS, with an eye on opportunities for building precise models that can be used throughout the development process. We do not attempt to give complete account of requirements engineering; we refer the reader to [129] for a comprehensive review. Instead, our aim is to draw attention to a few practices that, based on our experience, are particularly valuable for ERS projects.

7.3.1 Requirements Development and Management

It is important to recall that ERS are embedded into a container product; consequently, the requirements engineering process for ERS is embedded into and has to interact with the overall systems engineering process for the container product. This places constraints at the timelines in which requirements engineering for the ERS can occur, determines when requirements artifacts must be delivered into the overall process, and often provides a significant amount of context requirements for the interaction between the ERS and the rest of the system.

Even when not articulated explicitly, requirements play a central role throughout the development process of ERS. Following [129] we distinguish between *requirements development* and *requirements management*. Requirements development refers to all activities that lead to establishing a *requirements baseline* agreed-upon by the project's stakeholders. The baseline describes, as tightly as possible, the original understanding of all project participants about what the system to be built is. The requirements management process then starts from the baseline, and includes all activities required to respond to changes to that baseline. Its major activities include [129]:

– Define a change control process, including a Change Control Board (CCB)
– Maintain change request history
– Assess impact of change requests, and requirements volatility analysis
– Update of requirements baseline per CCB-decisions
– Establish versioning and change management tools

Explicit requirements management has the advantage that phenomena such as requirements creep (more or less sublime addition of requirements without allocation of new resources for their analysis and implementation) and requirements thrashing (a constant barrage of more or less meaningful change requests) become more transparent to all stakeholders, and can thus be addressed at the management level.

The distinction between development and management is important, because it draws explicit attention to the fact that requirements change needs to be explicitly managed throughout the development process.

Note that the core activities involved in requirements development are independent of the type of development process chosen. Any development process, be it plan-driven or agile [130], needs to find out what the system to be built is. The only difference is the value the respective process types place on formal documentation of the requirements, and how frequently the change process is triggered and executed.

A vast set of techniques has been developed and promoted to develop the requirements baseline. [129] identifies *elicitation*, *analysis*, *specification*, and *validation* as the core requirements development activities. Elicitation refers to activities that produce requirements from domain analysis and stakeholder interactions. Analysis refers to the elaboration, refinement and structuring of the requirements previously elicited with an eye towards building high-level design models that establish context for the requirements; this, again, occurs with stakeholder involvement. Specification refers to the prioritization and documentation of the analyzed requirements for transition to establish the requirements baseline. Validation refers to the inspection and testing of the specified requirements before they enter the baseline.

In practice, of course, requirements development is a highly interactive, iterative process, in which elicitation, analysis, specification and validation interleave. Depending on the overall systems engineering process of the product into which the ERS is embedded, these activities also interleave with and are informed by the activities of the overall systems engineering process.

In Section 7.1 we have seen that many requirements aspects for ERS are *cross-cutting* in the sense that they affect not only one component of the resulting system, but relate to an entire network of components. Note that this is *not* primarily a result of unnecessarily distributed architecture design (although this can be a cause as well.) Instead, this phenomenon arises from the inherent complexity of multi-functional systems, where hundreds to multiple thousands of software functions need to be offered and harnessed into a system of systems. Any decomposition of these functions into components will lead to some form of cross-cutting. Failure management is a prime example: no matter how the set of functions is sliced into logical or physical components, failures cannot be effectively managed from within these individual components – communication across components needs to occur to communicate failures, or to take remedying actions.

It is our belief that requirements engineering for ERS necessarily focuses on the *interplay* of the entities that make up the system, and the associated cross-cutting concerns. The rationale behind this is simple: because of the embedded nature of an ERS there is interaction between the ERS and its environment. Therefore, these interactions need to be understood to the maximum extent possible. Furthermore, for all but the most trivial systems, the ERS will itself decompose into a set of interacting components, each of which can be understood as an embedded component as well. Consequently the same rationale applies for the development of the ERS as a unit. In a networked system of ERS all quality properties of the system emerge from the interplay of all constituent ERS. Hence, the cross-cutting concerns that are crucial to defining the overall system's quality are naturally associated with the interplay of the constituent ERS.

Furthermore, we believe that to properly address the requirements aspects enumerated in Section 7.1, *explicit* domain models that speak to these concerns need to be constructed. As we will see in the case study of Section 7.4, creating such explicit domain models enables formal end-to-end analysis at the system of systems integration level – as opposed to the component-by-component level.

Therefore, we see the key activities in the requirements development process to bring out a sufficiently detailed domain model for ERS as follows:

(1) Identify the stakeholder group for the ERS under consideration.
(2) Identify pertinent business and process constraints for the ERS per stakeholder class.
(3) Identify the set of functions expected of the ERS by the stakeholder group.
(4) Identify the internal and external actors and data entities involved in these functions.
(5) Identify the interactions (event-, message-, control-, and data-flows) among the identified actors.
(6) Iterate over the identified functions to identify the actors and data entities needed to address the relevant cross-cutting concerns. Associate these with the interaction model built in activity 5.
(7) Identify operational infrastructure constraints, including mandated deployment contexts.
(8) Document requirements relative to the resulting models of structure and behavior.
(9) Validate requirements based on the resulting models.

Clearly, each of these activities breaks down into a variety of sub-activities and associated techniques; here, we focus on a high-level overview of these activities.

Activity 1 is critically important as per the definition of the term requirement we have given in Section 7.1. Recall that requirements are intimately linked to stakeholder values and, therefore, the set of stakeholders whose values the system is to address needs to be fully understood and modeled explicitly, sometimes via proxy elements, such as sensors.

Activity 2 serves to bring forward requirements aspects that are often neglected initially, and later turn out to be major success factors. This includes an articulation of the cost model that underlies the development process for the ERS and its container system. This is particularly important, at the integration level if the ERS is part of a system of systems. Similarly, this is the place to identify process requirements and laws or other regulations that govern the development of the ERS and its container system. This can influence the resulting domain models by creating data entities, actors or cross-cutting concerns that need to be further analyzed for functional and quality requirements.

Activity 3 is facilitated by a wide variety of techniques, such as use case [131] or user story analysis, stakeholder focus groups, flow analysis (event-, message-, control-, or workflow). For ERS we find it particularly useful to hold focused stakeholder workshops, within and across stakeholder groups to bring out not only per-component, but also across-component requirements. This is particularly critical for end-to-end and cross-cutting requirements aspects such as timing (specifically deadlines or time-budgets), failure modes and management, security, policy/governance, and deployment, update, and maintenance requirements. In these workshops we typically execute steps 4 through 9 together with the workshop participants to create initial domain model candidates on the spot.

Activities 4 through 7 amount to developing an ontology [132] and behavioral model of the core concepts that make up the domain model for the system under consideration. For an ERS this necessarily includes a model of the environment into which the ERS is placed. For the structural aspects of this domain model we favor class diagrams capturing the actor and data classes and their structural relationships. For the associated interaction model, we favor Message Sequence Charts (MSCs) and related interaction specification dialects such as Life Sequence Charts (LSC) [133], which can be augmented with constraints that reflect the cross-cutting concerns (see examples in Section 7.4).

A key observation is that the domain model should provide explicit hooks to associate the cross-cutting concerns with the interactions identified for the domain entities. This ensures (a) that the cross-cutting concerns are in the purview of the project team from the earliest stages as end-to-end aspects, rather than becoming an integration-afterthought, and (b) makes the cross-cutting concerns available for explicit validation and verification, rather than being an implicit, inaccessible aspect of the requirements model.

All entities mentioned in a textual description of a requirement should occur in the resulting model, and for each modeling entity there should be at least one requirement to which they are related.

The domain model is of such paramount importance, because we can derive a variety of other models from it, and use all models together for validation and verification. Derivative models include, for instance, a context-diagram, which shows the system entities *outside* the ERS under development, and what the structural and behavioral relationships between the two are. Furthermore, a useful domain model will capture the operational modes (high-level state transition view), major exceptions and failure modes, and the input/ouput protocols required at the interface of the ERS and its environment. It is central to the success of this exercise that it results in a model that captures the entities and relationships relevant to the problem domain and its associated stakeholder groups. This greatly facilitates validation and verification, as well as the derivation of design and implementation.

Of course, this modeling effort depends on a deep (and deepening) understanding of the problem domain. In recent years, there have been important attempts to help in building this understanding by providing catalogs of *requirements patterns* both ERS-specific, and domain-neutral. For instance, [134] have identified a catalog of ten requirements patterns that address the following concerns:

- *Controller Decompose Pattern:* decomposition of an ERS into subsystems according to responsibilities
- *Actuator-Sensor Pattern:* relationships among sensors, actuators, computational components and associated (environment) models
- *Examiner Pattern:* device monitoring and error logging
- *Fault Handler Pattern:* core entities and models for handling faults in ERS
- *Mask Pattern:* resource mediation for devices with many sensors and actuators
- *Moderator Pattern:* decoupling
- *User Interface Pattern:* reusability and flexibility for user interfaces associated with ERS
- *Channel Pattern:* communication facilitation among components
- *Monitor-Actuator Pattern:* fault management for actuators

Each of these patterns, among others, comes equipped with an explanation of the intent, motivations, constraints, applicability, entities and their structural and behavioral relationships.

Similarly, [135] presents a set of performance-related requirements patterns that are relevant for ERS. This includes patterns for response time, throughput, static and dynamic capacity (memory, computational power), and availability.

Besides these ERS-relevant requirements patterns, [135] also brings forward a rich set of more generic templates. These cover technology choices, standards compliance, inter-system interfaces, data typing and archiving, reporting, flexibility, and access control.

We call out activity 6 explicitly, because it is key to obtaining comprehensive requirements models for ERS. For each identified function of the ERS, the requirements pertaining to all quality properties [129] (availability, efficiency, flexibility, integrity, interoperability, reliability, robustness, usability, maintainability, portability, reusability, testability, security, safety, deployment, update, maintenance) should be iterated over to derive specific requirements that pertain to these qualities. Again, all of these qualities are cross-cutting in nature, and intimately linked to interactions among the identified system entities, or to the container system. Many of these qualities can thus be addressed at the integration- rather than the per-component-level. Automotive manufacturers and suppliers, for instance, have recognized this and are working together to provide a car-wide "middleware" that addresses some of these qualities *across* the components of the vehicular ERS networks.

Activity 7 serves to identify requirements that derive from the technical context of the ERS. Often, the technical infrastructure into which an ERS has to integrate is fixed long before the ERS proper is conceived. Then, this technical infrastructure injects deployment constraints into the ERS requirements set. In a clean-slate development, of course, one would seek to avoid this, or at least design the technical infrastructure after the integration requirements are sufficiently understood. In reality, however, legacy technical infrastructures exist and have to be considered. This interrelates with activity 6, of course, because some of the cross-cutting concerns may be discharged by the technical infrastructure if the latter is functionally rich enough. In any case, the capabilities of this infrastructure need to be carefully examined so as to know which of the cross-cutting concerns need to be lifted explicitly into the requirements model, and which ones are readily dealt with in the infrastructure.

Activity 8 refers to articulating the gathered requirements and their associated domain models in the form chosen by the project or mandated by a process requirement. For ERS this typically involves writing a *requirements document* that defines the scope, stakeholders, context, and all business, product, and process requirements elicited as part of the previous process activities. Discussion of an elaborate requirements document outline is beyond the scope of this text; we refer the reader to [129] for an example. However, we note that the material gathered in the previous activities typically provides a rich and authoritative source for this documentation activity.

Activity 9 can build on the models created in the preceding activities. The typical methods practiced for validation today are inspection, prototyping and simulation, automated consistency checking and verification. Each of these is facilitated greatly by detailed requirements models, as well as by broad stakeholder participation.

In reality, of course, all these activities will occur in an iterative, often interleaved fashion, rather than being executed in a prescribed sequence. The product of executing these activities, however, is a comprehensive requirements model for the ERS under development.

7.4 Example: Failure Management in Automotive Software

In the automotive domain, software has become the enabling technology for almost all safety-critical and comfort functions offered to the customer. The features supported by automotive software and electronics are increasingly dependent on the interactions of distinct components designed by different suppliers. Because of the increasing level of interaction between different components, industry standards, including OSGi [136] and AMI-C [137], introduce service based software-architectures and corresponding middleware layers as modeling and deployment abstractions. This marks a significant shift from component- to service-oriented software development in the automotive domain.

A major technological advantage of a service-based vehicle-electronics software architecture over a traditional component-based one is the ability to move the hardware-module-oriented partitioning of the vehicle system to a later point in the design cycle, allowing greater flexibility in integrating functions into hardware and potential elimination of redundant hardware across the vehicle. To exploit this advantage it is desirable to be able to model the vehicular software architecture on multiple levels, from static models of software structure to executable, time-accurate models of the actual system. This, in turn requires specifications for services that are sufficiently formal to allow tools to be built that check the integrated architecture for consistency and completeness, and to allow modeling tools to use the service-oriented specifications directly.

In the following, we illustrate the applicability of a service-oriented approach to model parts of the Central Locking System (CLS) found in typical modern cars. The CLS in the described form acts as a representative for similar problems in automotive control electronics and distributed, reactive systems in other application domains. We present aspects of requirements modeling, deriving a corresponding architecture, and performing safety-checking on the system model.

7.4.1 Central Locking System (CLS)

In modern cars, even a simple function such as locking the vehicle, i.e. central locking system (CLS), interacts with a significant number of other functions. There are not only interactions with the obvious modules, such as those controlling the individual door locks, but with less obvious systems as well, such as the vehicle speed sensor (to implement lock on drive away), the exterior lights (for remote lock acknowledgment) and the radio tuner and seat controllers (for setting driver preferences on unlock). The various interacting features in such a system are distributed across a number of different component modules, which are typically produced by different suppliers. As interactions between different subsystems increase, the features themselves become distributed across a number of components. This leads to increasing integration issues as features come to be implemented by software produced independently by a number of different suppliers.

Although we are considering a simplified version of the CLS for our study, it is evident that, given the size and distributed nature of the system, it is practically impossible to describe all the behaviors of all components involved completely. Instead, we only have a partial view on the requirements of the overall system.

7.4.2 Modeling the CLS Requirements

In the previous sections of this chapter, we have suggested a process for eliciting and managing requirements in 9 points. In this section, we present this approach using the CLS example. We demonstrate the use of a Service ADL to capture both the CLS system architecture and a set of dependability requirements along with formal verification techniques to verify the implementation of the dependability requirements.

1. Stakeholders Identification represents the first step of our requirement management process and prescribes the identification of the stakeholders for the system under consideration. In our case, the groups interested in the systems are obviously "the driver and passengers" of the car that will use the given CLS. Other groups come from the car development team, for example, the "engineering team" that designs the electro mechanical actuators for locking and unlocking the car. Each supplier is also a stakeholder that will have to agree on the final integrated design and can impose constraints to other parts of the system. In addition, marketing, cost, safety and legal considerations have great influence in establishing requirements for the vehicle.

2. Business and process constraints per stakeholder is the second step, which mandates to identify for each stakeholder the pertinent class of business or process constraints. In this case, we can analyze the concerns of the "safety and regulations" stakeholder and identify some critical requirements. One of such requirements is that "all the doors of a car shall be unlocked after an accident".

This type of regulation is not detailed enough to be a requirement directly. We first need to have a proper model of the car system and of the CLS to be able to articulate it further.

3. Identify functions expected by stakeholder – to support the previously stated rule, the "safety and regulations" stakeholder assumes that there exists in the system a function to detect an accident and a function to unlock all doors. It is important to notice that there will also be a timing constraint on the time interval between the accident and the unlocking of the car. For example, we can assume that requirement (1) is "the system shall unlock all doors of the car within half a second from the detection of an accident". The problem with such definition is that we need to define how an accident is detected and how reliably. Moreover, during an accident there could be failures in the system, which limit the functionality of the unlocking mechanism. This requirement could then be complemented by requirement (2) stating that "even if one electronic control unit of the car completely fails in an accident requirement 1 must be fulfilled".

4. Identify actors and data for the functions is the fourth step, where we analyze the requirements and functions identified so far, which leads to a number of use cases and actors. We identify the actors that participate in the services of the system under development, abstract from the concrete system elements and identify the communication roles. These roles will likely map to a variety of different component configurations depending on the concrete make and model under consideration. For instance, in a concrete implementation, the central controller (Control) and the lock management (LM) might end up on the same

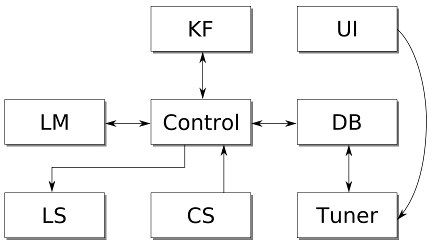

Fig. 7.1. Components and relationships in the CLS example

ECU, whereas the database (DB) and the lighting system (LS) might reside on others. Figure 7.1 depicts a simple configuration with each role being implemented by a system component. We indicate components using labeled boxes, and directed communication channels between them using labeled arrows. In a real car, most of these entities would be implemented on different ECUs (KF being a likely exception).

5. Identify interactions among actors – the starting point for this step is analyzing the set of relevant "use cases". In our case, we use message sequence charts to capture the identified use cases. Some of the use cases for the CLS are: locking, unlocking, lock_doors, unlock_doors, transfer_key_ID, and handle_crash. For reasons of brevity, we consider only a subset of these services here; we refer the reader to [138] for details.

transfer_key_ID is part of the unlocking process and associates seat and mirror positions, as well as tuner settings with the driver's key. handle_crash is a cross-cutting service that can interrupt all others, it captures the functionality that whenever a crash signal occurs the CLS has to unlock all doors.

While both the unlocking of the car and the transfer of a key ID are triggered by the user pressing a key on the key fob, we consider these two use cases separately because there exist keys that can unlock the car (mechanically, for instance) but do not transmit key identifiers. Therefore, separating use cases and corresponding requirements enable more modularity and reuse across different models of cars.

To capture the interaction patterns defining services we use an extended version of Message Sequence Charts (MSC) [139, 140]. MSCs have proved useful as a graphical representation of key interaction protocols, originally in the telecommunications domain. They also form the basis for interaction models in the most recent rendition of the UML [141]. In our extended MSC notation, each MSC consists of a set of axes, each labeled with the name of a role (instead of a class or component name). Roles map to components in a later design step of the development process. An axis represents a certain segment of the behavior displayed by the component implementing the corresponding role. Arrows in MSCs denote communication. An arrow starts at the axis of the sender; the axis at which the head of the arrow ends designates the recipient. Intuitively, the order in which the arrows occur (from top to bottom) within an MSC defines possible sequences of interactions among the depicted roles. We also use labeled boxes in

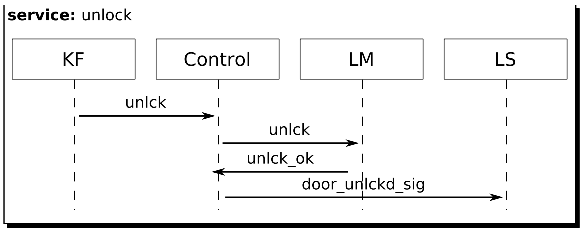

Fig. 7.2. MSC for "unlocking"

our MSCs to indicate alternatives and unbounded repetitions. High-level MSCs (HMSCs) indicate sequences of, alternatives between and repetitions of services in two-dimensional graphs - the nodes of the graph are references to MSCs, to be substituted by their respective interaction specifications. HMSCs can be translated into basic MSCs without loss of information [140].

Figure 7.2 shows an example; here we depict the interactions defining the "unlocking" service. It consists of a triggering message "unlck" from the key fob to the central controller. The latter forwards the "unlck" message to the lock management (LM). By introducing the LM role we abstract from the concrete number of locks present in the vehicle (doors front/back, trunk, moonroof, windows, security system, etc.). When the locks have been operated, LM returns an "ok" message to the control role. Upon its receipt, the control role issues a "door_unlckd_sig" message to the lighting system role, which handles the signaling of the locks' states to the driver. Clearly, this is just one course of actions that may happen during the execution of the unlocking service. The extended MSC dialect we use enables succinct specification of such alternatives [140, 142].

The next use case we turn into a service is "transfer_key_ID". Upon receipt of an unlck message the control role sends a getID message to the key fob; KF sends the id to Control, which relays it to the DB (see Figure 7.3). Again, Control switches from state LCKD to UNLD in the course of executing the service. The preceding two services are overlapping in the sense that both share references to the unlck message and states LCKD/UNLD. To compose these services into an overall service specification we have to identify the overlapping messages, and "synchronize" the execution of the services on these joint messages.

6. Identify elements to address cross-cutting concerns – along the main functional requirements for CLS we have also identified a cross-cutting requirement. In case of an accident, all doors need to be unlocked immediately. This concern comes from safety regulations that cars need to fulfill. Even if this concern is not part of the normal functions performed by CLS, it imposes a new behavior that interacts with the normal locking and unlocking behavior previously defined. Therefore, we need to identify structural elements and messages that are affected by this behavior. Moreover, we need to identify when the new behavior appears.

The handle_crash service has a particularly simple interaction pattern (see Figure 7.4): whenever the control role receives an "impact" message it responds by sending "unlck" to the lock management role, resulting in the unlocking of the

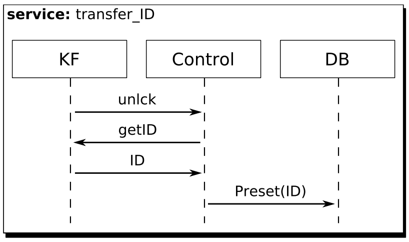

Fig. 7.3. MSC for "transfer_key_id"

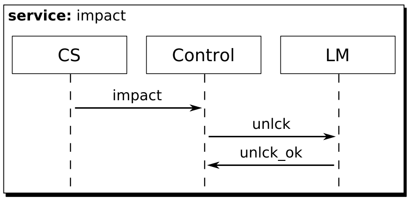

Fig. 7.4. MSC for "handle_crash"

vehicle. Methodologically this can also be handled by introducing a "preemption" concept that treats the response of the control role as the handling of a preemption triggered by the "impact" message.

7. Identify operational constraints – the ERS domain is characterized by tight timing constraints that can originate from several requirements. In our case study, we can consider, for example, the time constraints implied by the emergency unlocking requirement. We can capture such information in our service models using a modified MSC syntax.

Figure 7.5 shows the *unlock* function. The graphical syntax we use is derived from MSCs as described in [140, 142]. Upon receipt of the *unlck* message from KF, Control issues an *unlck* message to LM. Once LM acknowledges this with an *ok* message, Control requests signaling of the unlocking from LM by means of a *door_unld_sig* message, then returns *ok* to the keyfob.

The MSCs of Figure 7.5 is augmented with interaction deadlines, indicated by means of a labeled dashed line. The *unlock* function has a deadline of 150 ms. This means that the vehicle must be unlocked and the signaling must have occurred within 150 ms according to the interaction specification.

The deadlines we introduced in the MSC represent additional constraint that enable capturing QoS requirements directly in the service models.

8. Document requirements relative to the models – in our requirement elicitation process we also develop deployment models with at least a partial view of

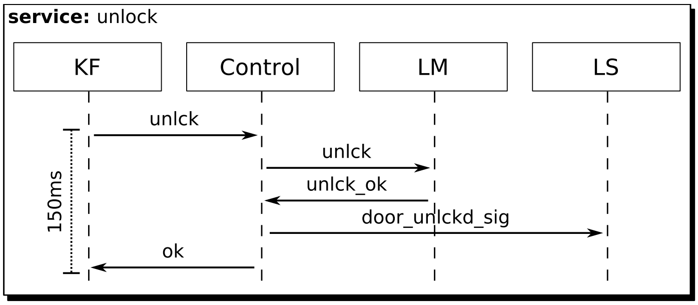

Fig. 7.5. A version of the unlock MSC with a QoS requirement added

the deployment environment. In our case study, such model is useful to identify possible failing components and ensure that the critical requirement of emergency unlocking all doors is fulfilled even when some component fails.

In our case study, we create a component model defining the ECUs that will run the CLS and the communication networks used to deliver messages to them. The behavior of each component is defined by assigning it one or more of the roles identified in our service models. This step of mapping the logical services to a concrete deployment model, makes the outcome highly specialized to the vehicle under design. On the other hand, the same functionality is often needed across vehicle platforms; this is certainly true for the CLS, which today is a standard feature across manufacturers and product lines. Therefore, the mapping process has to be repeated again to yield another specialized solution for each target platform. Because the requirements are clearly separated and we distinguish between one logical model and a deployment model, only the part of the work that deals directly with the deployment model has to be repeated while developing different car models.

The outcome of a traditional process would be eight separate component specifications; each individual component specification is complete in the sense that it has to address all the different functions the component in question might be involved in. In particular, the crosscutting nature of the functionality is lost when we look at each individual component; this results in the mentioned labor- and cost-intensive integration effort in late development stages.

We can obtain a trivial deployment domain model from the role domain model by removing the distinction between components and roles; then, each component implements precisely one role. In this state of affairs, the role domain model and the deployment domain model coincide. Another extreme case is to map all roles to a single component; this again is a trivial affair, because we simply need to treat the role domain model as a specification for the "internals" (the substructure) of one encompassing component. The most interesting and methodologically challenging case arises when we map multiple roles onto the same deployment component. All other cases (such as mapping a single role onto multiple components) can be dealt with by refactoring / refining the role domain model first, and then establishing the mapping to the deployment domain model.

In our case study, we choose to have six ECUs where we map the roles identified in our process. Figure 7.6 shows the corresponding deployment domain model. Our ADL enable us to specify communication busses (the big CAN BUS block in the middle of the figure), and electronic control unit connected to communication media (the six ECU blocks). An ECU can perform more than one role. For example in Figure 7.6 ECU1 plays the role of Control and DB, and ECU2 plays the role of UI and Tuner. On the other hand, the same role can be played by more than on ECU. This is the case of the CS role played both by ECU5 and ECU6. The reasons to replicate a role can be multiple. In the case of CS (the crash sensor role), the replication enables the detection of a crash even if one of the sensors fails.

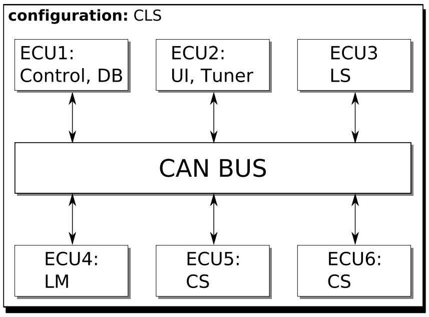

Fig. 7.6. CLS Deployment Architecture

If we work with strictly hierarchical component models such as the ones of UML2, UML-RT, or AutoFocus [143], one way to establish the mapping of multiple roles onto a single component is to take the role domain model as a staring point, and to replace the roles in question by a single component having the same input and output channels as the replaced roles taken together. Then, the entire network of replaced roles with their supporting channels becomes the hierarchical "child" of the freshly introduced component. This process can be repeated recursively into all hierarchically decomposed composites, until all role labels have been turned into component labels.

9. Validate requirements – failure management is particularly effective if it is performed throughout the development process[144] – rather than, as often happens, as an afterthought. For this reason, we raise awareness of failures already from the very early phases of the software and systems engineering process, during the requirements gathering phase. To this end, we have created a comprehensive taxonomy for failures and failure management entities. Failure taxonomy is a domain specific concept [144]. Our model-based failure management approach [145], leverages the interaction descriptions captured by services to identify, at run time, deviations from the specified behavior.

We enrich our standard service-oriented methodology with special services to manage failures. Hence, a key mechanism for dealing with failures is to define and decouple *Unmanaged* and *Managed* Services (see Figure 7.7). The Unmanaged Services are responsible for providing the required functionalities without considering failures, whereas the Managed Services enable the detection of failures and the implementation of mitigation strategies that avoid, or recover from, failures.

We also employ two special types of Services: *Detectors* and *Mitigators* (similar to the detector/corrector approach [146]). A Detector can detect the occurrence of a Failure based on its Effect (see Figure 7.9. This relation binds the Detector to the observable results of failures. Therefore, it is important to define what type of Effects a failure can have, and then to create appropriate Detectors.

The Detector detects the possible occurrence of a failure based on a *Detection Strategy.* One possible Detection Strategy is based on *Interactions.* In this case, a Detector compares the communication patterns captured in the service

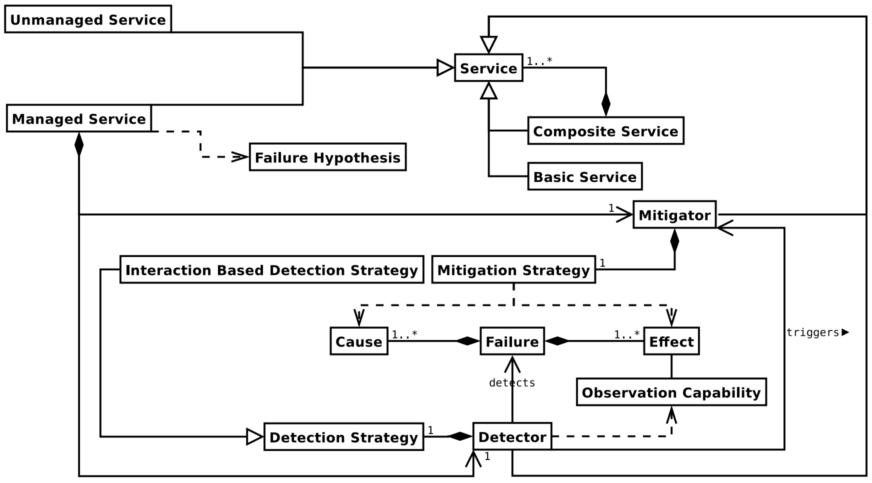

Fig. 7.7. Models of services

specification with the ones of the running system; then, it applies a mitigation strategy when behaviors don't match the specification. Mitigators are services that modify the interaction pattern of the system to recover from failure conditions.

Managed Services are a type of Services and, therefore, they can also be a component of a Composite Service. In particular, it is possible to have Managed Services that are composed of other Managed Services. Each one of them will have a Detector and a Mitigator that will address failures at its level. Using this schema, by hierarchically composing simpler services in more complex ones, and by adding Detectors and Mitigators to the various component services, it is possible to achieve a fine level of granularity in managing failures.

Each Detector is associated with a corresponding Mitigator. Upon detection of a failure, the Detector activates the corresponding Mitigator responsible for managing that specific failure. A Mitigator is another specific Service that is responsible for resolving the faulty state in order to maintain the safety of the system. A Mitigator applies its corresponding Mitigation Strategy to resolve the faulty state. Following the strategy pattern, decoupling the definition of the mitigation strategy from the entity that applies it provides flexibility to the model by allowing future changes to the strategy that is applicable to a specific failure without the need to make any additional modifications to other elements in the system.

This model allows us to compose a predefined Unmanaged Service with a Detector and its associated Mitigator in order to add failure management to it, thus, creating a Composite Managed Service. If multiple failures are supported for one Service, it will be wrapped in multiple layers of Detectors and Mitigators. This capability provides a seamless means to manage the failures that are found in further iterations of the design/development process, without redefining the existing Services. Figure 7.8 shows an example of a managed service for our case

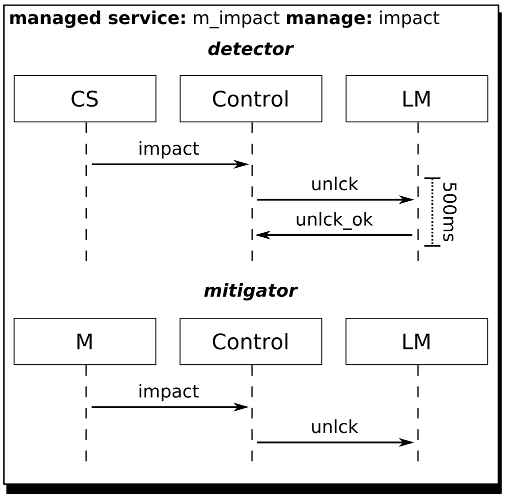

Fig. 7.8. Managed impact service

study. In the ADL fragment depicted in this figure, the detector identified if the impact service takes more than 500ms to acknowledge the unlocking of the doors after an impact, and in this case, it executes a mitigation service where an additional mitigation role (role M) repeats the unlck command.

An ontology guides the identification of failures and the activation of additional services that mitigate the effects of failures. We enrich the logical and deployment models typical of any MDA with a failure hypothesis that captures what physical and logical entities can fail in a system. It also provides a formal basis to reason about system correctness in presence of failures. Figure 7.9 shows the extended failure taxonomy using UML2 class diagram notation[147]. It captures the relationships between failures and our means for detecting and managing them. The central entity of this taxonomy is a *Failure*. A Failure has one or more *Cause*s and one or more *Effect*s. A failure Cause is very dependent on the application domain and could be due to either a software problem, i.e., *Software Failure*, or a hardware problem, i.e., *Hardware Failure*.

When a failure is detected, the system needs to mitigate it. This is done by following certain *Mitigation Strategies*. The Mitigation Strategy we must apply to deal with failures depends both on the associated Effects and their Causes. We identify two main strategies: *Runtime Strategy* and *Architectural Strategy*. Depending on the application domain, when a duplicated message is detected at runtime, *Ignore Message* can be a feasible Runtime Mitigation Strategy. Similarly, when a message loss is detected, *Resend Message* is a candidate Runtime Mitigation Strategy if properly supported by the interaction protocol between the exchanging parties. *Replicate Component* and *Failsafe Mode* are typical *Architectural Strategies*.

Following the outlined approach, we have lifted the management of failures to the logical architecture and started dealing with them from the early stages

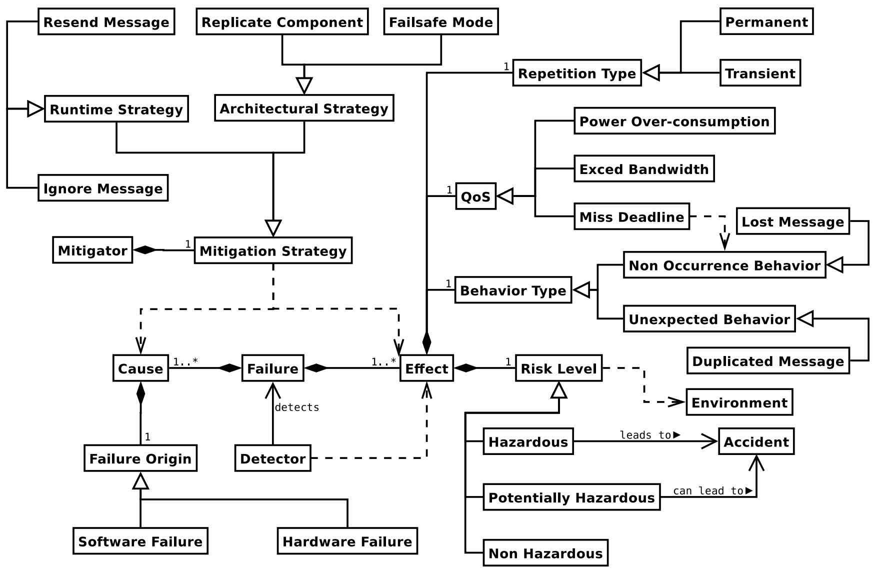

Fig. 7.9. Failure ontology

of the development process within the requirement elicitation phase. Once we have formal models of the services and a deployment architecture, along with a failure hypothesis, we can use a set of tools we developed to verify that the proposed software architecture indeed fulfills the given requirements.

The first step is to obtain an executable model form the services captured by our ADL – in [140] an algorithm to obtain state machines from MSC models is discussed. We have developed a tool [148] that can parse the service ADL and leverage the state machine synthesis algorithm to create a Promela [64] model that can be used to verify the property of a system using the SPIN model checker [149].

Each service is interpreted as a partial input/output function which defines the contributions of all participating roles to a communication pattern. The tool we have implemented uses all MSCs that define the system model to obtain a total representation of the global behavior of the system. Then it projects all messages sent or received by one role on a state machine that defines only the contribution of that role to the interactions of the system. Moreover, to cater for possible failures, the tool adds a sink state with guarded transitions from all other states. As result of this process we have one state machine for each role.

Once the tool has created all state machines for all roles, it can generate the Promela code. For each ECU in the deployment model and for each role mapped to them, the Promela code contains a concurrent process. Appropriate channel variable are used to map the communication channels of the service models to the proper ECUs. Additionally, a failure injector function, implementing the failure hypothesis is created in the Promela code. Failures are injected by killing roles (enabling the transition to the sink state) or disrupting communication channels (removing messages from channels).

Using our Service ADL for managing failures and our Promela code generator we have been able to verify the architecture of our CLS case study and ensure that the chosen architecture supports the safety requirement of unlocking all doors during an accident.

7.4.3 Discussion

In this example, we have seen that services require composition operators not generally available in component-oriented development: the concept of overlapping components is not very common. Roles, on the other hand, by definition capture a partial view on all components playing that role – to be composed with other partial views to produce the overall behavior of the component under consideration. The composition of the services as elicited above translates into a service specification. The mapping from a service specification to a set of components implementing the services in the next phase of the development process is a design step. This step entails fixing a component architecture, and an association between the components and the roles they play to support the given set of services.

In the CLS example, we could decide, for instance, to have just one component to implement the Control and LM (lock management) roles. This gives rise to a component-oriented "deployment" architecture. If the target architecture supports the definition and deployment of individual services, however, we can encapsulate the interaction protocols contained in each of the extended MSCs we have presented, and publish those as individually accessible services within service-oriented software architectures as outlined above.

We can also apply a bottom-up scheme for interaction composition. Deadlines can be applied to basic interactions. For instance, we define a deadline for a single message or a message sequence. For each composition operation, we apply defined rules that constrain the deadlines of the composite interactions. In this case, sequential composition leads to the addition of the operands' deadlines, loops to a multiplication, parallel, and join composition to the selection of the minimum deadline. All deadlines can be tightened manually. A less restrictive composition alternative (in comparison with applying the minimum constraint for join composition) would be to only consider a newly defined deadline for the composite. Doing so would allow the modeler to provide a different interpretation for the more complex composite function – it can be more than the sum of its parts. However, this may not yield a true refinement of the specification in the bottom-up sense, because the composite may not fulfill all QoS properties of the composed interactions anymore. Practical considerations would determine the concrete composition scheme used. We chose the composition variant that maintains all properties of basic interactions and allows for methodological refinement. We are aware that this is more restrictive to the modeler and requires more frequent modifications or refactorings of the specification.

In terms of methodology, we can also apply top-down refinement of deadlines, while still fulfilling all properties of bottom-up composition as described above. Starting from deadlines for entire functions, we allow the modeler to provide specific deadlines to parts of the interaction, as long as the overall deadlines are still satisfiable.

7.5 Summary and Outlook

Requirements engineering for Embedded Real-time Systems (ERS) is a tremendous challenge. In this chapter we (a) have highlighted the key aspects that render ERS requirements engineering difficult, (b) have discussed prominent approaches in the literature that tackle portions of these aspects, (c) have presented key activities that can help in model engineering for ERS across development processes, and (d) have shown how these activities play together to model and validate central failure management requirements in an automotive case study.

Clearly, there is no single technique that addresses the entire spectrum of requirements aspects from timing to distribution to failure management to local and cost drivers. Specifically, because many relevant ERS are, in fact, network-integrated systems of systems, most quality requirements are, in fact, concerns that cut across all components of the integrated system. This necessitates a modeling approach with due emphasis on the interactions among the parts to define the function of the whole system. Such an approach needs to provide models for interactions, but also for augmenting these interactions with constraints that address the cross-cutting requirements.

We have sketched the beginnings of such an approach by identifying key requirements elicitation activities for ERS, and how they can be used to produce structural and behavioral aspects of a corresponding domain model. In specifying the cross-cutting concerns we have identified interaction diagrams, such as extended UML sequence diagrams or Message Sequence Charts as a useful tool. The subsequent case study showed how to exploit this extensive domain modeling approach for the elicitation of domain-specific failure models ranging from logical to deployment architectures. The failure models capture a broad range of failures and associated detection and mitigation strategies. For a subset of these we have shown how to automatically generate [140, 148] verification models targeting Promela/SPIN to establish (or refute) fail-safety of a given architecture model. This technique can be utilized in validating requirements (does the architecture model properly reflect our understanding of fail-safety for the system under consideration?), or even the verification of proposed candidate architectures (do they fulfill the fail-safety requirement?)

This case study shows a pathway to modeling and model exploitation for ERS and can be expanded further to cover an increasingly rich set of requirements aspects. Generalizing from this example to obtain requirements engineering processes techniques and tools for a wide range of specific application domains is one promising area for future research. Another one is the seamless transition from gathered functional and (cross-cutting) quality aspects to re-configurable deployment architectures.

Acknowledgments. Much of the overview of requirements challenges is influenced by our long-time collaborations with automotive partners, as well as academic collaborators. We would like to acknowledge the influences of Manfred Broy, Alexander Pretschner, Bernhard Rumpe, Bran Selic, Wolfgang Pree, KV Prasad, Ed Nelson, Chris Salzmann, and Thomas Stauner. The authors are grateful for discussions with participants of the MBEERTS Dagstuhl Seminar, as well as for the insightful comments of the reviewers of this paper. Our work was partially supported by NSF grant CCF-0702791. Financial support came also from the California Institute for Telecommunications and Information Technology (Calit2).

References

[1] Shaw, M.: Prospects for an engineering discipline of software. IEEE Software 7(6), 15–24 (1990)

[2] Halfhill, R.T.: Embedded market breaks new ground, Embedded Processor Watch, vol. 82 (2000)

[3] Broy, M., Krüger, I.H., Meisinger, M. (eds.): ASWSD 2004. LNCS, vol. 4147. Springer, Heidelberg (2006)

[4] Broy, M., Krüger, I.H., Meisinger, M. (eds.): ASWSD 2006. LNCS, vol. 4922. Springer, Heidelberg (2008)

[5] Ahluwalia, J., Krüger, I., Meisinger, M., Phillips, W.: Model-Based Run-Time monitoring of End-to-End deadlines. In: Proc. of the Conference on Embedded Systems Software, EMSOFT (2005)

[6] Krüger, I., Nelson, E.C., Prasad, V.: Service-based software development for automotive applications. In: CONVERGENCE 2004 (2004)

[7] Pretschner, A., Broy, M., Kruger, I.H., Stauner, T.: Software engineering for automotive systems: A roadmap. In: 2007 Future of Software Engineering, pp. 55–71. IEEE Computer Society, Los Alamitos (2007)

[8] Sharp, H., Finkelstein, A., Galal, G.: Stakeholder identification in the requirements engineering process. In: Proc. Tenth Intl. Workshop on Database and Expert Systems Applications, pp. 387–391 (1999)

[9] Easterbrook, S.M., School of Cognitive, Computing Sciences, University of Sussex: Domain Modelling with Hierarchies of Alternative Viewpoints. University of Sussex, School of Cognitive and Computing Sciences (1992)

[10] Anderson, J., Fleak, F., Garrity, K., Drake, F.: Integrating usability techniques into software development. IEEE Software 18, 46–53 (2001)

[11] Bevan, N.: Usability is quality of use. Advances in Human Factors Ergonomics 20, 349 (1995)

[12] Mayhew, D.J.: The usability engineering lifecycle. In: Conference on Human Factors in Computing Systems, pp. 147–148. ACM, New York (1999)

[13] Bennett, J.L.: Managing to meet usability requirements: Establishing and meeting software development goals. Visual Display Terminals: Usability Issues and Health Concerns, 161–184 (1984)

[14] Chung, L.: Non-Functional Requirements in Software Engineering. Springer, Heidelberg (2000)

[15] Robertson, S., Robertson, J.: Mastering the requirements process. ACM Press/Addison-Wesley Publishing Co. (1999)

[16] Nixon, B.A.: Representing and using performance requirements during the development of information systems. LNCS, p. 187. Springer, Heidelberg (1994)

[17] Guinan, P.J., Cooprider, J.G., Faraj, S.: Enabling software development team performance during requirements definition: a behavioral versus technical approach. Information Systems Research 9(2), 101–125 (1998)

[18] Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based performance prediction in software development: A survey. IEEE Transactions on Software Engineering, 295–310 (2004)

[19] Gobbo, D.D., Napolitano, M., Callahan, J., Cukic, B.: Experience in developing system requirements specification for a sensor failure detection and identification scheme. In: High-Assurance Systems Engineering Symposium, Proc. Third IEEE Intl., pp. 209–212 (1998)

[20] Smidts, C., Stutzke, M., Stoddard, R.W.: Software reliability modeling: an approach to early reliability prediction. IEEE Transactions on Reliability 47(3), 268–278 (1998)

[21] Mooney, J.D.: Issues in the specification and measurement of software portability. In: Poster Session at the 15th Intl. Conference on Software Engineering (May 1993)

[22] Lauesen, S.: Software Requirements: Styles and Techniques. Forlaget Samfundslitteratur (1999)

[23] Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Reading (2004)

[24] Barker, S.D.P., Eason, K.D., Dobson, J.E.: The change and evolution of requirements as a challenge to the practice of software engineering. In: Proc. of the IEEE Intl. Symposium on Requirements Engineering, San Diego, California, January 4-6. IEEE Computer Society Press, Los Alamitos (1993)

[25] Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos (1996)

[26] Strens, M.R., Sugden, R.C.: Change analysis: A step towards meeting the challenge of changing requirements. In: Proc. of the IEEE Symposium and Workshop on Engineering of Computer Based Systems, p. 278. IEEE Computer Society, Washington (1996)

[27] Bohner, S.A., Arnold, R.S.: Software Change Impact Analysis. Wiley-IEEE Computer Society Pr (1996)

[28] Heninger, K.: Specifying software requirements for complex systems: New techniques and their application. IEEE Transactions on Software Engineering 6(1), 2–13 (1980)

[29] Carlshamre, P., Regnell, B.: Requirements lifecycle management and release planning inmarket-driven requirements engineering processes. In: Proc. 11th Intl. Workshop on Database and Expert Systems Applications, pp. 961–965 (2000)

[30] Al-Rawas, A., Easterbrook, S.M., National Aeronautics, Space Administration, United States: Communication Problems in Requirements Engineering: A Field Study. National Aeronautics and Space Administration; National Technical Information Service, distributor (1996)

[31] Easterbrook, S.M.: Handling Conflict Between Domain Descriptions with Computer-Supported Negotiation. University of Sussex, School of Cognitive and Computing Sciences (1991)

[32] Easterbrook, S.: Resolving requirements conflicts with Computer-Supported negotiation. In: Requirements Engineering: Social and Technical Issues, pp. 41–65 (1994)
[33] Boehm, B., Bose, P., Horowitz, E., Lee, M.J.: Software requirements negotiation and renegotiation aids. In: Proc. of the 17th Intl. Conference on Software Engineering, pp. 243–253. ACM, New York (1995)
[34] Crowston, K., Kammerer, E.E.: Coordination and collective mind in software requirements development. IBM Systems Journal 37(2), 227–246 (1998)
[35] van Lamsweerde, A.: Elaborating security requirements by construction of intentional Anti-Models. In: Intl. Conference on Software Engineering: Proc. of the 26 th Intl. Conference on Software Engineering, vol. 23, pp. 148–157 (2004)
[36] Potts, C.: Requirements models in context. In: 3rd Intl. Symposium on Requirements Engineering (RE 1997), pp. 6–10 (1997)
[37] Heeks, R., Krishna, S., Nicholson, B., Sahay, S.: Synching or sinking: Global software outsourcing relationships. IEEE Software, 54–60 (2001)
[38] Lala, J.H., Harper, R.E.: Architectural principles for safety-critical real-time applications. Proc. of the IEEE 82(1), 25–40 (1994)
[39] Lutz, R.R., Helmer, G.G., Moseman, M.M., Statezni, D.E., Tockey, S.R.: Safety analysis of requirements for a product family. In: Proc. 1998 Third Intl. Conference on Requirements Engineering, pp. 24–31 (1998)
[40] Xu, J., Randell, B., Romanovsky, R.J., Stroud, R.J., Zorzo, A.F., Canver, E., von Henke, F.: Rigorous development of a safety-critical system based on-coordinated atomic actions. In: Twenty-Ninth Annual Intl. Symposium on Fault-Tolerant Computing, Digest of Papers, pp. 68–75 (1999)
[41] Leveson, N.G., Stolzy, J.L.: Safety analysis using petri nets. In: The Fifteenth Intl. Symposium on Fault-Tolerant Computing. IEEE, Los Alamitos (1985)
[42] Leveson, N.G.: Software safety in embedded computer systems. Communications of the ACM 34(2), 34–46 (1991)
[43] Lutz, R.R.: Targeting safety-related errors during software requirements analysis. ACM SIGSOFT Software Engineering Notes 18(5), 99–106 (1993)
[44] de Lemos, R., Saeed, A., Anderson, T.: Analyzing safety requirements for process-control systems. IEEE Software 12(3), 42–53 (1995)
[45] Modugno, F., Leveson, N.G., Reese, J.D., Partridge, K., Sandys, S.D.: Integrated safety analysis of requirements specifications. Requirements Engineering 2(2), 65–78 (1997)
[46] Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Safety-Critical Systems Symposium, Birmingham, UK (February 1998)
[47] Hansen, K.M., Ravn, A.P., Stavridou, V.: From safety analysis to software requirements. IEEE Tran. on Software Engineering 24(7), 573–584 (1998)
[48] Napolitano, M.R., An, Y., Seanor, B.A.: A fault tolerant flight control system for sensor and actuator failures using neural networks. Aircraft Design 3(2), 103–128 (2000)
[49] United States Military Procedure: Procedure for performing a failure mode effect and criticality analysis, MIL-P-1629 (November 1949)
[50] Barlow, R.E., Chatterjee, P.: Introduction to Fault Tree Analysis (December 1973)
[51] Chung, L.: Dealing with security requirements during the development of information systems. In: Rolland, C., Bodart, F., Cauvet, C. (eds.) Proc. 5th Int. Conf. Advanced Information Systems Engineering, CAiSE, pp. 234–251. Springer, Heidelberg (1993)

[52] Landwehr, C., Heitmeyer, C., McLean, J.: A security model for military message systems: retrospective. In: Proc. 17th Annual Computer Security Applications Conference, ACSAC 2001, pp. 174–190 (2001)
[53] Frankel, D.S.: Model Driven Architecture. Wiley, New York (2003)
[54] IBM Rational DOORS (formerly Telelogic): DOORS (2009), `http://www.telelogic.com/`
[55] IBM: Rational RequisitePro. (2009)
[56] 3SL Cumbria, England: Cradle Requirements Management v6.0 (July 2009), `http://www.threesl.com/`
[57] Wiegers, K.E.: Automating requirements management. Software Development 7(7), 1–5 (1999)
[58] Jackson, M., Zave, P.: Domain descriptions. In: Proc. of IEEE Intl. Symposium on Requirements Engineering, pp. 56–64 (1993)
[59] Zave, P.: Classification of research efforts in requirements engineering. In: Proc. of the Second IEEE Intl. Symposium on Requirements Engineering, pp. 214–216 (1995)
[60] Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6(1), 1–30 (1997)
[61] Dardenne, A., Fickas, S., van Lamsweerde, A.: Goal-directed concept acquisition in requirements elicitation. In: Intl. Workshop on Software Specifications & Design: Proc. of the 6 th Intl. workshop on Software specification and design, vol. 25, pp. 14–21 (1991)
[62] Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. In: Selected Papers of the Sixth Intl. Workshop on Software Specification and Design, pp. 3–50. Elsevier Science Publishers B.V., Amsterdam (1993)
[63] Heninger, K.L., Kallander, J.W., Parnas, D.L., Shore, J.: Software requirements for the a-7 e aircraft. Memorandum Report 3876, Naval Research Lab., Washington D.C. (November 1978)
[64] Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
[65] Yu, E.S.: Modelling strategic relationships for process reengineering. PhD thesis, University of Toronto (1995)
[66] Yu, E.S.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proc. of the Third IEEE Intl. Symposium on Requirements Engineering, pp. 226–235 (1997)
[67] Yue, K.: What does it mean to say that a specification is complete? In: Proc. IWSSD-4, Fourth Intl. Workshop on Software Specification and Design, Monterey (1987)
[68] Darimont, R., van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration. In: Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering, San Francisco, California, United States, pp. 179–190. ACM, New York (1996)
[69] van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering 24(11), 908–926 (1998)
[70] Greenspan, S., Feblowitz, M.: Requirements engineering using the SOS paradigm. In: Proc. of IEEE Intl. Symposium on Requirements Engineering, pp. 260–263 (1993)
[71] Warmer, J., Kleppe, A.: The object constraint language: precise modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)

[72] Gill, A.: Introduction to the Theory of Finite-state Machines. McGraw-Hill, New York (1962)
[73] Hennie, F.C.: Finite-state Models for Logical Machines. Wiley, Chichester (1968)
[74] DeMarco, T.: Structured analysis and system specification, pp. 409–424. Yourdon Press, New York (1979)
[75] Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proc. of the First Intl. Conference on Requirements Engineering, pp. 94–101 (1994)
[76] Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. In: IEEE Transactions on Software Engineering, 58–93 (2001)
[77] Ross, D.T., Schoman, J.K.E.: Structured analysis for requirements definition, pp. 363–386. Yourdon Press, New York (1979)
[78] Gane, C.P., Sarson, T.: Structured Systems Analysis: Tools and Techniques. Prentice Hall Professional Technical Reference (1979)
[79] Orr, K.: Structured requirements definition. K. Orr, Topeka, Kan. (1981)
[80] Burns, A., Wellings, A.: Real-time Systems and Programming Languages, 3rd edn. Addison Wesley, London (2001)
[81] Boehm, B.W.: Software Engineering Economics. Prentice Hall PTR, Englewood Cliffs (1981)
[82] Fairley, R.: Software engineering concepts. McGraw-Hill, Inc., New York (1985)
[83] Berry, G., Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Institut National de Recherche en, Informatique et en Automatique (1992)
[84] Barnes, B.H.: Decision Table Languages and Systems. In: Metzner, J.R. (ed.) Academic Press, Inc., London (1977)
[85] Parnas, D.L., Madey, J.: Functional Documentation for Computer Systems Engineering. Queen's University at Kingston, Dept. of Computing & Information Science (1990)
[86] Schouwen, J.V.: The A-7 requirements model: re-examination for real-time systems and an application to monitoring systems. National Library of Canada (1991)
[87] van Schouwen, A., Parnas, D., Madey, J.: Documentation of requirements for computer systems. In: Proc. of IEEE Intl. Symposium on Requirements Engineering, pp. 198–207 (1993)
[88] Faulk, S.R.: State determination in hard-embedded systems. PhD thesis, The University of North Carolina at Chapel Hill (1989)
[89] Heitmeyer, C., Labaw, B., Kiskis, D.: Consistency checking of SCR-style requirements specifications. In: Proc. of the Second IEEE Intl. Symposium on Requirements Engineering, pp. 56–63 (1995)
[90] Heitmeyer, C., Mandrioli, D.: Formal Methods for Real-Time Computing. John Wiley & Son Ltd., Chichester (1996)
[91] Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: SCR*: a toolset for specifying and analyzing requirements. In: Reggio, G., Astesiano, E., Tarlecki, A. (eds.) Abstract Data Types 1994 and COMPASS 1994. LNCS, vol. 906, pp. 109–122. Springer, Heidelberg (1995)
[92] Heitmeyer, C., Kirby, J., Labaw, B.: The SCR method for formally specifying, verifying, and validating requirements: Tool support. In: Proc. of the 1997 (19th) Intl. Conference on Software Engineering, pp. 610–611 (1997)
[93] Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. 5(3), 231–261 (1996)

[94] Landwehr, C.E., Heitmeyer, C.L., McLean, J.: A security model for military message systems. ACM Trans. Comput. Syst. 2(3), 198–222 (1984)
[95] Faulk, S., Brackett, J., Ward, P., Kirby, J.: The core method for real-time requirements. IEEE Software 9(5), 22–33 (1992)
[96] Faulk, S., Finneran, L., Kirby, J., Shah, S., Sutton, J.: Experience applying the CoRE method to the lockheed C-130J software requirements. In: Reggio, G., Astesiano, E., Tarlecki, A. (eds.) Abstract Data Types 1994 and COMPASS 1994. LNCS, vol. 906, pp. 3–8. Springer, Heidelberg (1995)
[97] Miller, S.P.: Specifying the mode logic of a flight guidance system in CoRE and SCR. In: Proc. of the second workshop on Formal methods in software practice, Clearwater Beach, Florida, United States, pp. 44–53. ACM, New York (1998)
[98] Jaffe, M.S., Leveson, N.G., Heimdahl, M.P.E., Melhart, B.E.: Software requirements analysis for real-time process-control systems. IEEE Transactions on Software Engineering 17(3), 241–258 (1991)
[99] Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Transactions on Software Engineering 20(9), 684–707 (1994)
[100] Heimdahl, M., Leveson, N.: Completeness and consistency in hierarchical state-based requirements. IEEE Transactions on Software Engineering 22(6), 363–377 (1996)
[101] Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
[102] Parnas, D.L., Wang, Y.: The trace assertion method of module interface specification. Queen's University, Dept. of Computing & Information Science, Kingston, Ont., Canada (1989)
[103] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16(4), 403–414 (1990)
[104] Hatley, D.J., Pirbhai, I.A.: Strategies for real-time system specification. Dorset House Publishing Co., Inc., New York (1987)
[105] Ward, P.T., Mellor, S.J.: Structured Development for Real-Time Systems. Prentice Hall Professional Technical Reference (1991)
[106] Ravn, A.P., Rischel, H.: Requirements capture for embedded real-time systems. Proc. of IMACS-MCTS 91, 1147–1152 (1991)
[107] Douglass, B.P.: Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
[108] Chen, R., Sgroi, M., Lavagno, L., Martin, G., Sangiovanni-Vincentelli, A., Rabaey, J.: UML and platform-based design, pp. 107–126. Kluwer Academic Publishers, Dordrecht (2003)
[109] Object Management Group: SysML Specification Version 1.0 (2006-05-03) (August 2006), `http://www.omg.org/docs/ptc/06-05-04.pdf`
[110] Rioux, L., Saunier, T., Gerard, S., Radermacher, A., de Simone, R., Gautier, T., Sorel, Y., Forget, J., Dekeyser, J.L., Cuccuru, A.: MARTE: a new profile RFP for the modeling and analysis of real-time embedded systems. In: UML for SoC Design Workshop at DAC 2005, UML-SoC 2005 (2005)
[111] Object Management Group: UML profile for schedulability, performance, and time (September 2003)
[112] Axelsson, J.: Real-world modeling in UML. In: Proc. 13th Intl. Conference on Software and Systems Engineering and their Applications (2000)

[113] Berkenkötter, K., Bisanz, S., Hannemann, U., Peleska, J.: The HybridUML profile for UML 2.0. Intl. Journal on Software Tools for Technology Transfer (STTT) 8(2), 167–176 (2006)
[114] Bichler, L., Radermacher, A., Schürr, A.: Integrating data flow equations with UML/Realtime. Real-Time Syst. 26(1), 107–125 (2004)
[115] Kirsch, C.: Principles of real-time programming. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 61–75. Springer, Heidelberg (2002)
[116] Berry, G.: The foundations of Esterel. In: Stirling, C., Plotkin, G., Tofte, M. (eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge (2000)
[117] Maraninchi, F.: The Argos language: Graphical representation of automata and description of reactive systems. In: IEEE Workshop on Visual Languages, Kobe, Japan (October 1991)
[118] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language Lustre. Proc. of the IEEE 79(9), 1305–1320 (1991)
[119] Camus, J.L., Dion, B.: Efficient Development of Airborne Software with Scade Suite. Esterel Technologies (2003)
[120] Caspi, P., Raymond, P.: From control system design to embedded code: the synchronous data-flow approach. In: 40th IEEE Conference on Decision and Control, CDC 2001 (December 2001)
[121] Kopetz, H., Bauer, G.: The Time Triggered Architecture. In: Proc. of the IEEE Special Issue on Modeling and Design of Embedded Software (2002)
[122] Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: A time-triggered language for embedded programming. Proc. of the IEEE 91(1), 84–99 (2003)
[123] Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A.: Event-driven programming with logical execution times. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 357–371. Springer, Heidelberg (2004)
[124] Templ, J.: TDL Specification and Report. Technical report, Department of Computer Science, University of Salzburg, Austria (March 2004)
[125] Farcas, C.: Towards Portable Real-Time Software Components. PhD thesis, University of Salzburg (2006)
[126] Farcas, E.: Scheduling Multi-Mode Real-Time Distributed Components. PhD thesis, University of Salzburg (2006)
[127] Farcas, E., Pree, W., Templ, J.: Bus scheduling for TDL components. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 71–83. Springer, Heidelberg (2006)
[128] Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent distribution of real-time components based on logical execution time. ACM Press, Chicago (2005)
[129] Wiegers, K.E.: Software Requirements: Practical Techniques for Gathering and Managing Requirements Throughout the Product Development Cycle. Microsoft Press, Redmond (2003)
[130] Boehm, B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Professional, Reading (August 2003)
[131] Holbrook, I.H.: A scenario-based methodology for conducting requirements elicitation. SIGSOFT Software Engineering Notes 15(1), 95–104 (1990)
[132] Gruber, T.: A translation approach to portable ontology specifications. Knowledge Acquisition 5, 199 (1993)
[133] Damm, W., Harel, D.: Lscs: Breathing life into message sequence charts. In: Formal Methods in System Design, pp. 293–312. Kluwer Academic Publishers, Dordrecht (1998)

[134] Konrad, S., Cheng, B.: Requirements patterns for embedded systems. In: Proc. IEEE Joint Intl. Conference on Requirements Engineering, pp. 127–136 (2002)
[135] Withall, S.: Software Requirement Patterns. Microsoft Press, Redmond (2007)
[136] OSGi: OSGi Alliance Specifications (2007), `http://www.osgi.org/`
[137] Automotive Multimedia Interface Collaboration: AMI-C Software API Specifications – Core API (2003), `http://www.ami-c.org/`
[138] Krüger, I.H., Ahluwalia, J., Gupta, D., Mathew, R., Moorthy, P., Phillips, W., Rittmann, S.: Towards a process and Tool-Chain for Service-Oriented automotive software engineering. In: Proc. of the ICSE 2004 Workshop on Software Engineering for Automotive Systems, SEAS (2004)
[139] ITU-T Geneva: ITU-T Recommendation Z.120 – Message Sequence Chart (MSC 1996) (1996)
[140] Krüger, I.H.: Distributed System Design with Message Sequence Charts. PhD thesis, Technische Universität München (2000)
[141] Object Management Group (UML 2.0), `http://www.omg.org/uml/`
[142] Krüger, I.H.: Capturing overlapping, triggered, and preemptive collaborations using MSCs. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 387–402. Springer, Heidelberg (2003)
[143] Munich University of Technology: AutoFocus (1996-2002), `http://autofocus.informatik.tu-muenchen.de/index-e.html`
[144] Leveson, N.G.: Safeware: system safety and computers. ACM Press, New York (1995)
[145] Ermagan, V., Krüger, I., Menarini, M., Mizutani, J.I., Oguchi, K., Weir, D.: Towards Model-Based Failure-Management for Automotive Software. In: Proc. of the ICSE 2007 Workshop on Software Engineering for Automotive Systems, SEAS (2007)
[146] Arora, A., Kulkarni, S.S.: Component based design of multitolerant systems. IEEE Transactions on Software Engineering 24, 63–78 (1998)
[147] Object Management Group: UML 2.1.1 Superstructure Specification (2007)
[148] Ermagan, V., Farcas, C., Farcas, E., Krüger, I.H., Menarini, M.: A service-oriented approach to failure management. In: Proc. of the Dagstuhl Workshop on Model-Based Development of Embedded Systems, MBEES (April 2008)
[149] Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison Wesley, Reading (2003)