8 UML for Software Safety and Certification

Model-Based Development of Safety-Critical Software-Intensive Systems

Michaela Huhn[1] and Hardi Hungar[2]

[1] Institute for Software Systems Engineering, Technische Universität Braunschweig, Mühlenpfordtstr. 23, 38106 Braunschweig, Germany
m.huhn@tu-bs.de
http://www.cs.tu-bs.de/sse
[2] OFFIS eV, Escherweg 2, 26121 Oldenburg, Germany
hungar@offis.de
http://ses.informatik.uni-oldenburg.de

Abstract. With the proliferation of UML in the development of embedded real-time systems, the interest in methods and techniques integrating safety aspects into a UML-based software and system development process has increased. This chapter provides a survey on relevant UML profiles and dialects as well as on design and verification methods and process issues supporting a safety assessment. These subjects are discussed in the light of norms and standards on software development for safety-critical systems.

8.1 Introduction

Nowadays, software has become an integral part of safety-critical systems in nearly all technical domains, from aeronautics or power generation, to traffic control or medical devices. Due to advances in mechatronics and communication the role which software plays is expected even to grow in future. In addition, the complexity of control to be implemented increases permanently. The adaptation of the well established model-based software engineering paradigm to the specific needs of safety engineering is an obvious and frequently proposed approach to systematically cope with the challenges of developing software components in safety-critical systems.

As stated by N. Leveson [1] and others, safety is an issue to be solved on the system and not the component level. Since software is immaterial, it differs from physical entities: Software by itself will not harm persons, property or the environment. But as an integral part controlling the behavior of physical components, its correct functioning contributes to safe operation or hazardous situations [2], as any other component of a safety-critical system. Software failures are mostly considered as systematic, having their cause in the safety analysis or software development process, whereas physical components may also fail at random.

It is the purpose of the discipline of *software safety engineering* to prevent software failures to occur. According to [3], software safety engineering has three major sub-processes: (1) *Software safety analysis* extends system safety analysis

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 201–237, 2010.

to software components in that hazards particularly relevant for software and software/hardware interaction are identified. The software safety analysis sub-process results in software safety requirements and safety design strategies aiming at elimination or mitigation of the identified hazards. (2) In *software safety design*, the software is designed and implemented according to the requirements and safety strategies. Safety design activities take the needs of safety assurance for traceability, documentation and safety argumentation into account. (3) *Software safety assurance* is concerned with all activities that provide evidence that the software meets its safety objectives. Verification and validation (V&V) activities are essential constituents of this sub-process. They are by themselves not sufficient, but their results have to be incorporated into an overall safety argumentation that integrates them into the system safety process.

In this paper, we focus on UML-based approaches to the sub-processes of software safety design and assurance. These two are also considered together in standards like the CENELEC standards for railway applications [4], the RTCA-DO-178B for airworthy software [5], or the IEC 61508-3 [6] on software requirements for the functional safety of electrical / electronic / programmable electronic controlled systems. Model-based techniques for the preceding system and software safety analysis like failure modes, effects and criticality analysis (FMECA), hazard and operability analysis (HAZOP), or event tree analysis (ETA) [7] are beyond the scope of this paper. They are partially addressed in Chapter 10 "Model-based Analysis and Development of Dependable Systems". An approach to a further aspect not covered here, namely UML-based dependability analysis, can be found in [8, 9].

As a consequence, we assume the software safety requirements to be provided. Another important input, coming from safety analysis, is the criticality level that classifies the software's contribution to system safety. The criticality level determines a so-called *software integrity level* (SIL) in the IEC 61508-3 [6], and the CENELEC standard [4][1]. Each SIL is equipped with requirements and recommendations on processes, activities and roles, and on software engineering design and V&V techniques. For the higher SILs, formal models are highly recommended for requirements analysis, design and verification. However, the relation of formalisms, which are mentioned in present standards, to artifacts of the development process remains vague. Thus, model-based software development and in particular the integration of model-based design and V&V techniques is a lively research field and major challenge in safety-critical systems engineering [10].

We start our presentation in Section 8.2 by briefly recapitulating the essentials of software development for safety-critical systems conforming to existing norms and standards, and deriving from that a categorization of usages of software models in software safety processes. In Section 8.3 we survey safety-related extensions of UML and classify them according to their purposes and usages. Section 8.4 sketches a seamless certification-oriented process based on UML. The perspectives of model-based V&V techniques and tool support are discussed in Section 8.5. Section 8.6 concludes.

[1] A similar concept in RTCA-DO-178B [5] are *development assurance levels* (DAL).

8.2 Development of Certifiable Software

The standards [4, 5, 6] do not prescribe a specific process model, but they require clearly distinguished development phases or activities with predefined input and output documents. The classical V-model (see Fig. 8.1) is well-accepted for software development for high assurance systems, in particular in the context of the CENELEC standards [4] and IEC 61508 [6] that both refer to it. The standards regulate key objectives to be addressed by the activities and in the documents. The development has to assure quality criteria like conciseness, completeness, traceability or testability. Safety-related requirements and constraints have to be distinguished and traced throughout the development phases. They are the major subject of the recommended verification and validation techniques. Moreover, for achieving safety, programming strategies and mechanisms like defensive programming or cyclic self-tests [4] are to be applied.

As stated in the introduction, we concentrate on UML-based approaches employed in software safety design and assurance of critical systems. Safety analysis, which we mention briefly in Sec. 8.3.4, is a mandatory preceding sub-process in a safety-critical system's life cycle. For the phases of safety design and assurance, two results of the safety analysis are of major interest: (1) The product-specific requirements for functional safety, i.e., goals to be achieved constructively in order to eliminate or mitigate the identified hazards. (2) The association of a SIL classifying the risk resulting from a failure of a software component. A SIL 0[2] classification means that the component is not related to system safety functions, whereas a SIL 3 and SIL 4 classification is assigned if a component failure may cause a severe or even catastrophic accident.

The standards associate with each SIL a set of process requirements or objectives to be met, concepts to be employed and techniques to be applied in order to achieve an acceptable level of confidence that systematic flaws in software development are eliminated. For software developed under SIL 3 or 4, specific formal and semi-formal model-based techniques are highly recommended for software specification, for software verification, and to complement software validation (see table A.2, A.5, and A.8 in [4]). However, today's standards [4, 6], do not state clearly which software engineering techniques should or may be used to achieve the required software quality characteristics.

An advanced view is taken by the Committee Draft for Voting (CDV) of IEC 61508-3 [11]: It suggests an explicit semi-quantitative *quality model* to relate particular usages of software engineering techniques to detailed quality characteristics of development artifacts. This relation is expressed in terms of a degree of rigour by which a certain software engineering technique can achieve a quality characteristic.

Another issue that hampers the proliferation of model-based development methodologies in software safety design is the fact that traditional programming is assumed for module design and implementation in the standards (see for

[2] In RTCA-DO-178B, the corresponding classification ranges from A to E in reversed order.

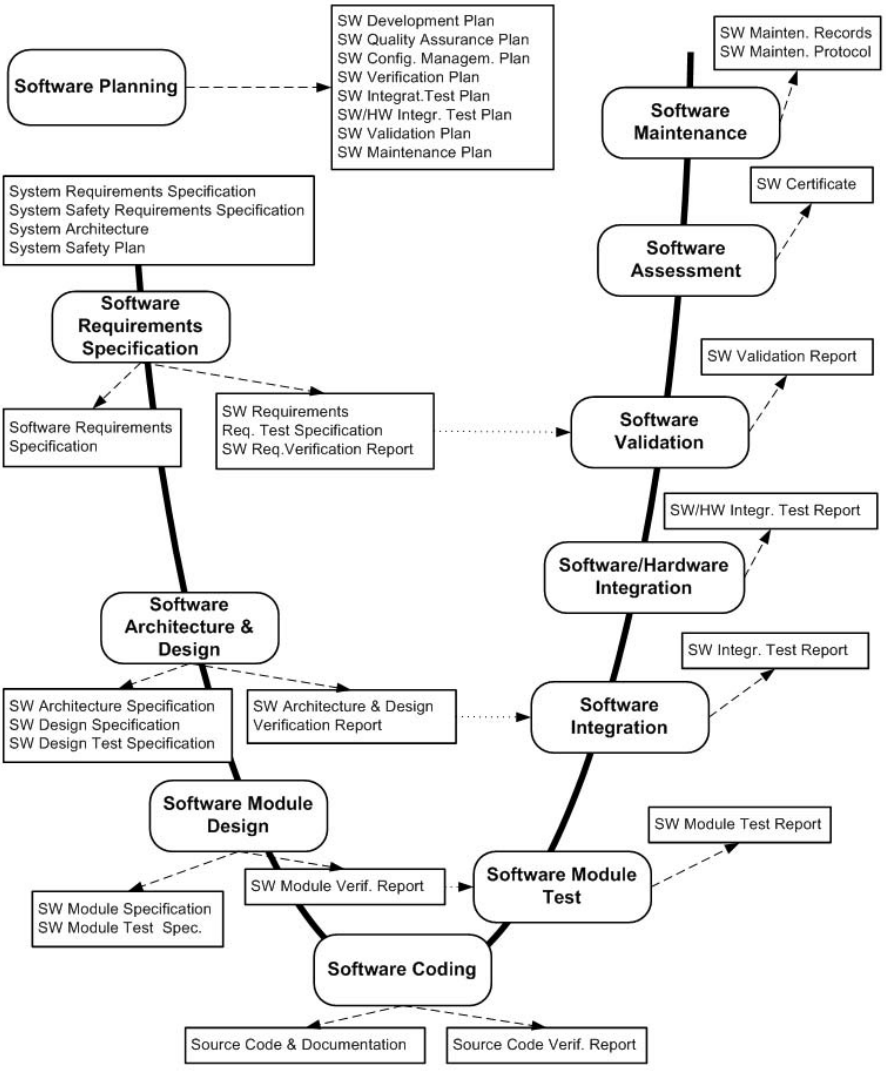

Fig. 8.1. V-Model according to EN 50128

instance the V-model according to EN 50128 [4] in Fig. 8.1). Restrictions to a safe subset of programming languages and approved compilers according to the SIL are highly recommended. How to establish a corresponding notion of safe model-based programming and code generation is discussed in Sec. 8.4.

All tools which are employed within the development process of safety-critical software have to be qualified. In general, tools that facilitate design automation – as in particular model-driven approaches incorporating code generation - are requested to be qualified with the same rigour as the safety-critical software

itself. Whereas tools that are intended for use in the safety assurance process – i.e. that support testing, validation or verification - can be qualified by a more light-weight assessment process. An example for a certified code generator is the SCADE Software Factory [12].

We summarize this elaboration in the following observation: Any scientific approach to customize UML as a modeling notation for the use with safety-critical systems has not only to fulfil the intentions but also the practical certification-oriented requirements set by the standards. Only then, it will come into operation in industrial safety-critical system development. Practical certification-oriented requirements are in particular: (1) A conclusive argument for the usage of a UML-based method for specific design or V&V activities has to be provided to prove the method's adequacy for the quality characteristics required for a certain artifact. This part will benefit from progress in standards: E.g., the upcoming CDV of IEC 61508-3 is less concrete with respect to the referenced modeling formalisms than [4] and [6], and more explicit with respect to the quality characteristics to be achieved by a technique. (2) As explained in the previous paragraph, a tool supporting a UML-based method has to be qualified according to its usage in the development process. Even if an approach does not strive for *design automation* (see Sec. 8.5), the standards' requirements are satisfied only by very few UML tools today.

In the view of our recapitulation of the standards, we identify the following six categories where software models can be used in the development and certification of safety critical systems:

Usage 1 - Precise specification of safety and software requirements: As a starting point for software safety design, the domain and safety engineers identify a set of safety requirements and constraints that have to hold on the system under development or evolution. Subsets of these are allocated to software components in the architectural steps of decomposition and partitioning. Requirements models are used to assist a common and detailed understanding of all safety-related issues between the software engineers and the safety and system experts. This usage scenario aims at enhancing the communication processes on safety-related requirements at the interface between system level and software component view.

Usage 2 - Software design and evolution: The next step after requirements specification is software design. In a model-based approach, the software architects and engineers describe the design or evolution task in terms of models representing various views. In case of embedded safety-critical software, hardware-dependent runtime properties like real-time behaviour, power consumption or resource utilization are an integral part of the functional safety requirements, and thus these properties should be addressed in the models. Additionally, a number of specific software safety strategies and techniques (e.g., defensive programming or multiple version dissimilar software, watchdogs or voters) are recommended for architectural design. Hence, this usage scenario describes the process of designing and thereafter implementing software components of safety-critical systems.

Usage 3 - (Partial) code generation: In a model-driven approach, automated model transformations and code generation are employed to obtain target-specific, executable models from design models. This scenario extends the model-based design scenario described before: Code generation moves efforts from manual implementation and extensive testing from the code level to model analysis. But a prerequisite is a qualified development environment that assures that the semantics of the models within the modeling environment corresponds to that of the generated code as it is executed by the runtime environment on a specific target. In addition, from the safety engineering viewpoint this usage of models faces a number of difficulties as discussed in Sec. 8.5.

Usage 4 - Verification and testing: At all stages of software development, the software engineers have to show that the outcomes meet the specifications and constraints induced by the previous stage. Verification and testing are part of the safety assurance process. Formal modeling of the software and system behaviour and its specification plays a prominent role here: The standards, e.g. [4], recommend a number of formal modeling notations that were considered potentially useful at the time the standards were published. However, the standards remain unspecific in which technique should be employed for which kind of safety requirement: Software safety requirements that are derived from well established safety analysis techniques like SHARD (Software Hazard Analysis and Resolution in Design) [13] cover a broad spectrum of software failures like omitted or untimely reactions, or unexpected or missing parameters. In difference to that, the referenced techniques like HOL [14] or CCS [15] focus on subsets like functional correctness and correct interaction behaviour. Moreover, questions of model validation, i.e. showing evidence that the formal model truly represents the relevant behaviours of the real system, are not addressed explicitly though they are of course highly important.

In practice, this phase is dominated by testing applied either to code or to executable models. In research, the usage of formal models for different verification techniques is considered at least as relevant as model-based testing.

Usage 5 - Software validation: Validation is the process of establishing conclusive, documented evidence that a system satisfies its requirements. In early phases, validation can be supported by animating, resp., simulating executable models. In later phases, software models may be part of the informations available to the validator, in particular for systems developed under SIL 3 or 4. For these, the standards require that the validation and design tasks have to be performed by independent teams. To transfer this principle to model-based approaches, independence between the models that are used for design and those to derive tests from has to be guaranteed.

Model validation is again, as already mentioned in Usage 4, an inevitable prerequisite for accepting results from model analyses as evidence for requirements compliance of the software.

Usage 6 - Software certification: In the certification process, assessors from a certification authority will examine whether the system will operate

adequately safe. Therefore, the manufacturer delivers a so-called safety case. In the safety case, the safety claims for the system in its operating environment are identified and linked by structured, sufficient and comprehensible arguments. These address the documentation on the development process, design artifacts, and verification and test results that provide evidence that the claims are valid. Traceability of the requirements through the whole process as they are realised step by step in the design, and an underlying rationale are major prerequisites for certification.
In practice, maturity of processes, techniques and tools is also mandatory. In addition, particular software engineering techniques that are well accepted in safety engineering, like the restriction of programming constructs to a safe subset in the implementation phase or Modified Condition Decision Coverage (MC/DC) as a testing technique oriented towards code coverage, are an integral part of the documented evidence of conformity to the standards. Formal models may be part of the design documentation (usage 1 or 2) or the basis of analyses that support the evidence of the safety arguments.

It has to be pointed out that UML does not belong to the notations explicitly referenced in safety standards. Hence, employing UML models in the software safety design and assurance process requires conclusive safety case arguments on several aspects:

(1) The use of UML models in activities and through the development process has to be clarified as for any other artifact. It has to be demonstrated how and to which confidence level the requested safety objectives and quality characteristics can be achieved by UML-based techniques.
(2) The standards recommend a rich portfolio of safety strategies ranging from defensive programming, design diversity or restriction of programming languages to elements that are statically verifiable to formal V&V techniques and testing. Those strategies are widely accepted in safety engineering and they should be supported by a modeling approach.

8.3 Safety-Related Extensions of UML

In this section we survey UML profiles and dialects dedicated to safety-critical software development. From the numerous works, we have selected a subset of approaches aiming at a seamless and tool-supported model-based software safety and assurance process. As the software safety process is complex and multi-faceted, the approaches differ significantly in their aims and methodologies:

UML Profile for Developing Airworthiness-Compliant Safety-Critical Software [16] aims at a tight linkage of a UML-based software design with the safety argumentation in the context of RTCA DO-178B (see Sec. 8.3.1).

rtUML and the OMEGA-RT Profile [17] focusses on seamless integration of UML design models and a rich collection of formally founded, tool-supported V&V techniques (see Sec. 8.3.2).

Safe-UML [18] tailors UML for certifiable software safety design in the railway domain. Besides a formal foundation, also best practices to achieve quality characteristics for the design and safety-directed issues in model-based programming are considered. (see Sec. 8.3.3).

UML Profile for Modeling and Analysis of Real-Time Embedded Systems [19] explicitly addresses resource allocation and SW/HW integration. It supports the specification and analysis of real-time and performance properties (see Sec. 8.3.4).

Railway Control System Domain Profile [20] targets seamless support for formally founded design, code generation and verification of interlocking functionality in the railway domain (see Sec. 8.3.5).

A concise comparison of the particular strengths of these UML profiles is given in Table 8.1.

SysML [21] is not discussed here, because it has a more general objective of extending UML from software to system development and safety is not addressed by particular modeling elements.

EAST-ADL [22] is an architecture description language for the automotive domain defined upon UML 2.0. EAST-ADL offers notational elements for the Goal Structuring Notation [23] to model arguments of a safety case in context of the upcoming automotive standard ISO 26262 [24]. However, its support for safety remains rudimentary compared to other UML profiles.

8.3.1 The UML Profile for Developing Airworthiness-Compliant (RTCA DO-178B) Safety-Critical Software

In [16], Zoughbi, Briand, and Labiche address the explicit representation of safety information within UML models that constitute the requirements, the design, the deployment, or the finally installed configuration of a software system. The authors aim at a better understanding of safety issues during development and certification. They want to improve the communication between safety engineers, software developers, and assessors from the certification authorities (usage 1, 2 and 6).

The authors identified 65 safety-related concepts in the airworthiness standard [5] that are relevant for software models. The concepts are grouped into eight categories: *safety, reliability, integrity, concurrency, performance, certification, design, and configuration.* However, all concepts contribute (at least indirectly) to software safety. The relationship between the concepts is formalized in a conceptual meta-model. The meta-model is the basis for the definition of stereotypes, tagged values and constraints of the UML profile, and it reflects the key idea of integrating the safety argument into the UML models for software design.

A central concept is `Safety Critical` which is used to stereotype entities with direct impact on system safety. By the tagged values `Criticality Level` and `Confidence Level` the developer may declare the criticality level of a

safety critical component determined in a safety analysis[3] and his/her confidence that the requested criticality level will be reached. Thus, a direct link is established between the design elements in the UML model and the safety argumentation according to a standard. The link between the safety terms from the standard and the UML model is strengthened by two major groups of concepts in the meta-model that are connected via the `Safety Critical` concept. The first group supports argumentations on design by offering concepts like (safety) `Requirements`, `Rationale`, `Strategy`, or `Deviation`. They describe design decisions, architecture rationale and modifications of approved plans that the developers make when they transform the original requirements into a design. The second group enables to explicitly represent technical safety engineering expertise in the UML model. For instance, design elements to detect and handle any kind of safety-related event are uniformly structured and classified by the stereotypes `Monitor`, `Handler`, `Event`, or `Reaction`. In a similar way, a group of concepts related to `Replication Group` allows to characterize the safety strategy of a replicated group of components whose elements are connected to a voter. Among others, concepts like `Style` are provided to describe the kind of a selected solution in common software safety terminology on the level of detailed design and implementation.

Since the proposed UML extension is defined as a UML profile, integration in existing UML modeling tools is possible. The authors propose an integration into frameworks like Rhapsody by IBM [25] or the Eclipse Modeling Framework (EMF)[26]. Thereby, the designer and certifier are supported in searching UML designs for occurrences of specific stereotypes or tagged values either by using a proprietary API or the Object Constraint Language (OCL). Thus, certain information - like listings of all COTS components used or all hardware-software interfaces - that are required for certification in the context of RTCA DO 178B [5] can be generated automatically. Additionally, traceability can be achieved if the model is fully elaborated according to the methodology suggested by the authors. Therefore, not only the UML model must contain different views on the software architecture and the design. The requirements linked to the design rationale and safety considerations leading to that design have to be represented in the model, too. Then the designer may traverse the model guided by the stereotypes provided by airworthiness profile to comprehend the safety argumentation.

To summarize, the airworthiness profile by Zoughbi, Briand, and Labiche is tailored for UML-based development and certification of safety-critical software according to the RTCA DO-178B. Safety information supporting the communication, and reasoning for safety cases is integrated into UML design models. With this focus on incorporating the safety argumentation *into* software design models, the airworthiness profile can be understood as a standard-specific alternative to approaches that provide the safety argumentations *externally* like the *Goal Structuring Notation* by Kelly [23] or *Assurance Based Development* by Knight et al. [27].

[3] e.g. "A" to "E" if the component is developed according to RTCA DO-178B.

8.3.2 *rtUML* and the OMEGA-RT Profile

rtUML and the OMEGA-RT Profile were defined in the context of the EU funded project *Correct Development of Real-Time Embedded Systems* OMEGA[4] as an extension of UML 1.4. The OMEGA approach integrates functional views and extra-functional properties, mainly timing, into functional views on specification, architecture and detailed design. A main goal is a formal foundation enabling tool-supported formal verification and validation techniques [17] (usages 1,2, 4, and 5).

rtUML comprises those functional concepts from UML that are considered most relevant in the embedded domain: For a structural design view, object-oriented concepts like polymorphism, inheritance, aggregation as well as various kinds of associations can be used in class diagrams. Active, passive and reactive objects are distinguished. To model the behaviour of a class or object resp., hierarchical state machines with a rich action language are included in *rtUML*. Interaction between so-called activity groups can be modelled as synchronous or asynchronous inter-object communication. *rtUML* can be pre-compiled to a kernel language *krtUML* containing only basic concepts from class diagrams and flat state machines. An operational, discrete time semantics in terms of *Symbolic Transition Systems* (STS) for *krtUML* was defined by Damm, Josko, Pnueli and Votintseva in [28].

rtUML is extended for requirement specification, architectural descriptions and in particular for the specification and verification of real-time aspects: Requirements can be specified scenario-based as Live Sequence Charts (LSCs) [29] or by temporal logic formulae. On the architectural level, a component-connector view comprises required and provided interfaces, protocol state machines, and OCL constraints. The OMEGA-RT Profile distinguishes different kinds of internal events like the send and accept of signals or state enter and exit events. Additionally, matching clauses and filters can be used to specify constraints on the duration between two event occurrences. The model can be extended by classes stereotyped as observers to express more involved timing requirements. Observers emulate timed automata in the UML modeling setting.

A rich portfolio of verification and validation techniques and tools supports software development with *rtUML* and the OMEGA-RT Profile [17]: Live Sequence Chart specifications can be animated with the Play Engine tool for requirements validation. Formal verification on finite state design models can be performed in two ways: Either a model checker specifically optimized for *rtUML* models can be used to prove specifications in terms of LSCs or temporal logic formulae. Alternatively, a model transformation to the IF framework [30] can be applied, enabling discrete and continuous time verification. Automated time and data abstraction mechanisms are offered for state space reduction as a preparatory step for model checking. To enable formal verification for infinite state models, a model transformation from *rtUML* to PVS (Prototype Verification System) [31] is provided. Using the interactive theorem prover PVS, infinite

[4] www-omega.imag.fr

value domains or unbounded message queues can be handled. As the transformation includes type information and OCL constraints on the model, these can be checked in PVS as well.

8.3.3 Restricting UML for Specification and Programming in a Certification Context

Motivated by the wish to be able to use UML in a way compatible with the railway norms (mainly EN 50128), Safe-UML has been designed as a restriction of (a part of) general UML. It is intended to address the functional viewpoint, expressed in class diagrams and statecharts [18]. To adequately cover an application range from documentation over specification (artifacts in the early phases of the design process) to actually UML-based programming, the definition has been organized in two levels:

Safe-UML (S): (*S* for *Superstructure*) applies to the OMG standard [32] for superstructures. It takes the definitions of state machines and class diagrams of UML and eliminates all semantical ambiguities, sources of underspecification, unclarity and unboundedness of system resources. In particular, it considers the parallelism (and its potentially sequentialized implementation).

Safe-UML (P): (*P* for *Programming*) applies to IBM's Rhapsody in Cpp as an instance of a UML implementation which enables programming in UML via the Cpp code generation. Safe-UML (P) gives directions on how to achieve conformance of the generated code with coding guidelines. Together with the rules from Safe-UML (S) it defines a set of restrictions which turn UML with Cpp annotations into a programming language suitable for the development of safety-critical systems.

Though originally designed for the rail domain – for instance, Part 42730 of the Mü 8004 [33] was taken for the definition for an admissable subset of Cpp – it is applicable also in other domains, particularly if the IEC 61508 is the source for the standard to be adhered to.

Safe-UML (S) and the Principles Guiding its Definition
In the following, we will give a short overview of essential features of Safe-UML, grouped as instances of four main principles which guided the definition of the language. The cross-cutting issue of parallelism and communication is treated separately.

Unambiguity: Every construct used must have a clearly (unambiguously) defined semantics. General UML, for instance, explicitly includes "semantic variation points" such as the handling of incoming events. In such cases, Safe-UML restricts to a particular interpretation, such as a bounded FIFO queue.

Determinacy: Usually, UML behavior specifications are nondeterministic. This is, for instance, the case if there are conflicting transitions leaving the same state,

or if behavior is executed in orthogonal regions of a state machine. Safe-UML (S) tackles these problems by adding constraints to (a) prevent these situations to occur (e.g. guards of conflicting transitions must be exclusive, if they are triggered by the same event), or, if this is not possible, (b) ensure that the outcome is the same for each possible execution order, so that the internally nondeterministic behavior cannot be observed externally.

Clarity: Clarity addresses the question of accessibility and understandability of a specification or program. As an example, the state machines may be influenced severely by the context in which they are used (e.g. a transition triggered by an event may never fire, because the event is deferred in an enclosing state machine). Such effects are targeted by adding constraints which try to reduce context influence to a minimum (e.g. a constraint that events should not be deferred).

Boundedness: Consumption of time and space are particularly important aspects of a safety-critical system. I.e., system reactions shall come in time, the system must never deadlock nor run out of memory, etc. So, among other things, unbounded multiplicities are forbidden in class diagrams, and transition loops are ruled out in state machines.

Multiple Threads and Communication: To capture this major source of problems, Safe-UML requires a conservative system structure which is closely related to the one assumed by *rtUML* – in fact it also bases on [28] and its semantic definition. A major feature is the requirement of a finite, static structure where all active objects are organized in *active groups*, each featuring one active object with a common set of queues for events, timers, calls and completion events (one set for the active group). Problems related to sequentialization within queues, potential queue overflow, deadlocks due to multiple calls are in general hard to avoid. Safe-UML forbids some constructs and, for the rest, refers the developer to proven patterns of communication and scheduling, resp., to methods establishing correctness (like an abstraction to a decidable Petri-net property).

Safe-UML (P) — Safely Programming in UML

The objective in the definition of Safe-UML (P) is to turn state machines with Cpp annotations and class diagrams into a graphical programming language which by itself adheres to principles underlying the definition of coding guidelines ([33, 34]), and, taking the code generator from Rhapsody, translates into a fragment of Cpp meeting these restrictions.

First, of course one must restrict the Cpp annotations to the UML constructs accordingly. Part of the remaining answer is given by importing the Safe-UML (S) restrictions, which essentially restrict the (mostly) graphical UML constructs in a way one would restrict a programming language for safety – see the four principles exemplified above. And last but not least, the implementation dependent (Rhapsody-specific) code generation has to be considered.

The generator translates the UML constructs to a Cpp program using a library, the so-called framework, which essentially provides all necessary objects and methods to execute them, i.e., the equivalent of a runtime system. The code generator, if parametrized properly [35], produces rather well-structured code, so that only minor issues do arise. This analysis has been performed on a large set of examples systematically covering the graphical constructs and annotations.[5] The framework, which is part of the resulting Cpp, is itself not programmed according to strict safety guidelines. It can freely be modified by the Rhapsody user, so that one can remedy the defects identified in [36]. Framework modifications may also be employed to complement the restrictions on the graphical UML level by adding safety features to, e.g., event communication. Such an approach has already be used successfully in a signaling application which has been certified by the German Railway Authority.

Summarizing, Safe-UML defines a way to rigorously specify and safely program using UML in the rail domain and similarly regulated contexts (usage 2,3 and 6). It is, however, not yet integrated into design environments, and its (P) level is geared towards a particular implementation.

8.3.4 The UML Profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE)

In 2008, the OMG published the Beta Specification for a UML Profile for "Modeling and Analysis of Real-Time Embedded Systems" (MARTE) [19] that shall replace the existing UML Profile for "Schedulability, Performance and Time" (SPT Profile) [37]. As stated in the title, the primary concern of the MARTE profile is real-time in embedded (RTE) systems, and not safety. However, the correct timing is part of functional correctness. With its modeling extensions, the MARTE profile supports detailed design and verification of safety-critical RTE systems (usages 2 and 4). Since MARTE is already realized as a plug-in of Papyrus for UML [38], tool support is available.

The MARTE foundations offer elements for modeling logical and physical time, resources and the spatial and temporal allocation of functional application entities onto them. The MARTE design model contains a so-called "RTE Model of Computation and Communication" to characterize the concurrency and synchronization behavior. A generalized, UML-conformant description of standardized APIs of real-time operating system like POSIX, QNX, or OSEK is supported. The extensions of the MARTE analysis model aim at the integration of state-of-the-art techniques for schedulability and performance analysis at the level of detailed design. Techniques like SymTA/S [39] or Modular Performance Analysis [40] offer tool supported analyses for various, common scheduling strategies and communication protocols. Their use is twofold - either predictive or verifying: For *predictive* use, a design model is enriched with estimated values on execution times and communication loads and with a specification of the planned scheduling situation. The analysis result is predictive and can (only) increase the

[5] This approach parallels a widely used practice of compiler validation.

confidence that the system design will fulfil its requirements on response times (worst, average or best case) or path latencies. Moreover, the analysis can be used to optimize the real-time dimensioning of a design [41]. In the case that values from the implementation are available, the analysis formally *verifies* whether real-time requirements are met in all possible situations. To pave the way for this kind of real-time analysis in the MARTE context, the concepts from the real-time analysis models are included in the MARTE profile. Thus, the definition of model transformations into the analysis framework is straightforward.

Additionally, the MARTE profile provides a package for the declaration of *non-functional properties* and an associated value specification language. Thereby the developer may annotate the model with further information relevant for safety-critical systems. In particular, reliability and availability issues addressed in the UML profile for "Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [42] can be integrated seamlessly by these means [43].

The MARTE profile has been extended towards dependability [44] analysis by several authors: Pataricza [45] introduced the concept of error propagation[6] to the General Resource Model of the MARTE predecessor, the SPT profile [37], to enable efficient system level diagnosis based on partial diagnostic information. He used quality of service parameters to characterize errors and the error behavior is modeled explicitly. Recently, Bernardi, Merseguer and Petriu [8] proposed a more general extension of the MARTE profile for analyzing and modeling dependability. Their "Dependability Analysis Model" addresses reliability, availability, maintainability, and safety, so-called RAMS properties, as major attributes of dependability. Among others, a "Threat Model" is introduced to describe either errors and failures when reasoning on reliability and availability or hazards that are relevant for safety. In that, the Dependability Analysis Model in [8] mainly focuses on the analysis of RAMS properties which is a usage of UML that precedes the software design and assurance process.

As shown by Thomas, Delatour, Terrier, and Gérard [46], the rich set of concepts for resource allocation provided by MARTE permits to clearly separate the model of the application from an explicit model of the real-time execution platform. The explicit platform model is taken as input to govern the model transformations to different target platforms in an MDA approach. In that, the Software Resource Modeling sub-profile has the potential to support deployment and code generation (usage 3) that goes beyond existing approaches that address the RTE characteristics only implicitly.

An alternative approach was chosen in the OMEGA-RT profile by Graf, Ober, and Ober [47] where a specific RTE platform model is explicitly addressed in the formal semantics that fosters automatic, correct code generation.

[6] According to [44], we call an event, at which a violation of the specified behavior becomes observable at the system boundary, a *failure*. An *error* describes the occurrence of a deviation from the intended behavior that may be internally compensated. If the error is propagated to the system's interfaces, a failure occurs.

8.3.5 The Railway Control System Domain Profile (RCSD)

Berkenkötter and Hannemann [20] conservatively extend UML 2.0 by a domain specific profile for railway and tram control systems. The RCSD profile supports the precise specification of railway networks with the aim to automatically generate code for a specific interlocking functionality. It shall foster the unambiguous communication between railway experts and embedded software designers and lay a foundation for the automated generation of verified controller software (usages 2, 3 and 4 for a specific rail application).

The RCSD profile offers basic entities to model railway tracks, namely track segments, points, and crossings. Additionally, there are elements for signals indicating the driving instruction for the following track segment, specific sensors for detecting whether a track element is occupied by a train, and automatic train runnings, which enforce braking if a train does not obey the signaling. The states of these elements are described by attributes for which specific datatypes are introduced. The topology of a railway network is modeled through the neighboring relation given by specific associations of sensors to track elements. Additionally, a set of top level constraints is included to ensure global consistency and completeness.

A class diagram models a restricted pattern or sub-problem of the RCSD domain, employing for instance further constraints on some entities. An object diagram can then be used to describe the track layout of a concrete network as an instance of the sub-problem of the corresponding class diagram. On both modeling levels, the static semantics is precisely defined by an elaborated set of OCL constraints that can be evaluated automatically on class and object diagrams annotated according to the RCSD profile [48].

The dynamic semantics is defined as a *Timed State Transition System*. Timed transitions are defined locally for the RCSD elements. The behavior of a network is the parallel composition of its component behaviors. To ensure safe train passage through the network, a *controller* realizing the interlocking functionality has to be added to the network model. Haxthausen, Peleska et al. [49] have shown how to generate the controller automatically from sets of generic transitions patterns that are instantiated according to the concrete network and the set of pre-defined routes when synthesizing the controller.

In addition, formal verification is supported on the level of the configured network by employing bounded model checking and inductive reasoning. A set of generic functional safety requirements is provided that covers the specific interlocking problem addressed by RCSD. Thereby all those states of the network are characterized - in terms of train locations, moving directions, and point positions - that are considered hazardous. The safety predicates on the configured network to be enforced by the controller are derived automatically by instantiation. The formal system description is transformed into SystemC that serves for both, verification input and executable code. To validate software/hardware integration the authors propose automated hardware-in-the-loop testing.

The RCSD profile provides proprietary prototype tool support for the development of a specific class of controllers in the railway domain on a non-standard

compliant, formal semantics. The profile's application domain is clearly restricted for the benefit of rigorous formalization of the specification and design and an intertwined set of verification techniques covering several process phases.

8.4 Using UML in Certification-Oriented Processes

8.4.1 Questions to Be Addressed by a Certification-Oriented Process

The central idea of model-based software development is to employ models as key design elements, expressing design aspects in a tangible way. For safety criticality or even certification, it is desirable to extend the role the models play: We want to integrate them into the documentation of the development entering the safety case. Thus, not only the detailed usage of UML models within each activity or process phase has to be explained, but also quality characteristics requested for artifacts have to be substantiated, and a way to achieve this quality has to be delineated. Common quality characteristics are completeness and correctness wrt. the requirement specification, traceability, simplicity and understandability, behavioral determinacy, testable and (statically) verifiable design, fault tolerance, and last but not least, linkage to the safety analysis sub-process in both parts, design and assurance. If UML models are employed in more than one activity, their relation and consistency becomes an issue, too. Wrt. achieving the quality of models, one may note that models which document the finished design usually are not finalized in an early phase, but have to change over the course of the development. This is an issue to be reflected in the process definition. In this section, we sketch a process framework which emphasizes the aspect of iterative model development in a way compatible with standard requirements. It is a framework of a process in that it will have to be instantiated to the concrete development context and to the project requirements.

8.4.2 Purpose and Scope of the Proposed Process

The process outlined here has been defined to be compliant with the EN 50128 for developing safety critical software for the railway domain. The sketch is based on results of the OpRail[7] project [50], and we will call it the OpRail process in the following. Its primary goal is to delineate a way to harmonize the use of UML for expressing design artifacts and tools related to UML development with the strict requirements of the EN 50128. As this norm has been derived from the more widely applicable IEC 61508, the sketched process outline is useful beyond the railway domain.

The main motivation for the definition of the process was to introduce expressive modeling features from UML into the development, to enhance precision and

[7] This project has been funded by the German Ministry of Education and Research (BMBF) under grant No. 01|SC26A. The process has been mostly developed by the project partner Berner& Mattner.

Table 8.1. Safety-related UML profiles and their usages

	Usage of UML Models						**Tool Support**
	1 **Specification**	2 **Design**	3 **Code Generation**	4 **Verification & Testing**	5 **Validation**	6 **Certification**	
Airworthin UML (aerospace)	annotations for SW safety requirements	built-in SW safety design patterns & strategies				traceability of RTCA DO-178B safety concepts through UML models	(extendable UML tools) + prop. support for search & review
***rtUML* & OMEGA-RT**	use cases, scenarios, constraints, observers	components & interfaces: structure, behavior, timing		model checking & theorem prov., refinem. & RT verification	LSC animation, consistency checks, property deduction		tool coupling: IF 2.0 toolbox, Play Engine, PVS
Safe-UML (railway)	use cases, scenarios, temp. logic formulae	component structure & behavior, module behavior	restricted Cpp code generation from classes & state machines	model checking, test generation: sequ. diagr., coverage criteria (MC/DC)		model-based process definit., adaptation of UML due to EN 50128	tool integration: Rhapsody + ATG + prop. extensions
MARTE	extra-functional properties, resource modeling, timing	SW&HW comp., allocation, RT constraints, dependability extensions	PIM → PSM: operating system & protocol configuration	schedulability & performance analysis			Papyrus for UML + interfaces to sched. analysis tools
RCSD (railway sub-domain)	constraints, generic temporal logic safety properties	domain model for track topology & railway control	controller synthesis & SystemC code generation	model checking, induct. reason-ing, autom. HIL testing	consistency checks		coupling of prop. tools

communicability of design artifacts, to explicitly represent the iterative nature of development activities – for instance the way in which early prototyping is employed – and to profit from the wide offering of tool support available for UML. It is mainly the intention to permit iterations and early prototyping which made it necessary to deviate from the V shape as depicted in Fig. 8.1. Despite the V model being quasi mandatory (as we mentioned above), norm compliance can be achieved by mapping the components of the V process to the new one.

The OpRail process covers the software development only, with interfaces to other development activities, including the integration of legacy code. Also sketched is the role tools could play and the requirements they would have to satisfy. In this short presentation, we only hint at these latter aspects.

8.4.3 Terms and Definitions

Process. A process defines who is doing what, when and how. A sketch of a process model like the one given in Fig. 8.1 is an illustration with a focus on the temporal aspect. A process may be composed of a set of sub-processes and is divided into different phases.

Sub-Process. A sub-process is a part of a process with is either focused on a particular aspect (like *Safety Management*) or the collection of actions to perform a logical step like the components in Fig. 8.1. A sub-process may span several phases of the process.

Phase. A phase is the period in a project begun and ended by major project milestones. A phase may encompass several sub-phases that may be repeated multiple times (iterations). Within a phase, a well-defined set of objectives is to be met and certain artifacts are to be produced.

Step. Within a phase, a (sub-)process can be divided into a number of more elementary steps.

Milestone. A milestone is an important event (completion of specified products) during the course of a project which can be scheduled and monitored and may be used for evaluating the progress of the project. The decision to move a project to the next phase is taken at a milestone. If the decision is negative, the milestone must be rescheduled and repeated.

Artifact. An artifact is an outcome of a sub-process or phase. It may be a required result of the process or some other piece of information that facilitates and supports the process. Artifacts may be grouped to artifact sets that are assigned to different sub-processes. For example, an artifact set can be composed of documentation, models, software modules etc. Artifacts shall be clearly specified by a given version number.

Activity. An activity is the execution of a step of the process.

Iteration. An iteration is a repetition of an activity, with the purpose of improving the end result, usually until a certain condition is met. Iterations are introduced to capture explicitly that many activities are often performed in this way, and to reflect agile development styles.

8.4.4 Phases and Sub-processes

Since we do not consider maintenance activities in the definition of our process, there are eleven sub-processes (of the twelve from Fig. 8.1) to be mapped.

Software Planning, Software Requirements Specification, Software Architecture & Design, Software Module Design, Software Coding, Software Module Test, Software Integration, Software/Hardware Integration, Software Validation and Software Assessment.

The OpRail process organizes these into (only) four phases, where these phases are not executed sequentially but overlap each other. Accordingly, there are five milestones, `M0` to `M4`: `M0` starts the first phase of the project, `M1` to `M3` mark transitions between phases, and `M4` ends the project.

The artifacts which are tied to the sub-processes are transferred to the phases. If a sub-process starts within a phase, this phase produces versions of the artifacts which are defined as an outcome of the phase, continuing a sub-process means revising the artifact, and finishing a sub-process finalizes the artifact. Accordingly, an artifact may be produced in stages *draft*, *revised* and *final*, where of course the version *final* corresponds to the artifact as defined in the standard. That is, the OpRail process produces a documentation of the development of the system as if it was carried out according to the V process, thus presenting a familiar view suitable for certification.

In short, the phases are defined as follows.

Concept Phase (`M0` to `M1`). Typically the concept phase consists of one to two iterations. In practice, the first iteration can be interpreted as the offer phase. Firstly, the main focus is to analyze and understand the problem. All input documents shall be reviewed. All SW related requirements, architectures and plans shall be proposed in draft.

Definition Phase (`M1` to `M2`). In this phase the SW requirements, architecture and detailed design shall be fixed and reviewed. The design shall be simulated and tested early to figure out cost-intensive design flaws. This phase shall ascertain that the proposed system can be realized as specified. Afterwards the requirements set can be approved by e.g. the EBA (Eisenbahn-Bundesamt) or assessed by an ISA (Independent Safety Assessor). After approval, it is not allowed to change the requirements. Further changes have to be realized as changes within the change management workflow.

Realization Phase (`M2` to `M3`). The main focus of this phase is to realize the solution and construct a product. First of all, this means implementation of the functionality fixed in the System and SW Requirement Specifications. The implementation is accompanied by unit and SW integration and SW/HW integration tests. This phase ends with approved SW and SW/HW integration testing.

Qualification Phase (`M3` to `M4`). This phase includes validation tests. Furthermore it includes tasks for the assessment and certification of the system. This phase should end with extensive field tests which include the approval of the customer.

The full description (resp., a full instantiation) of the process contains a mapping of the stages of all design artifacts to the phases. Fig. 8.2 illustrates such a mapping.

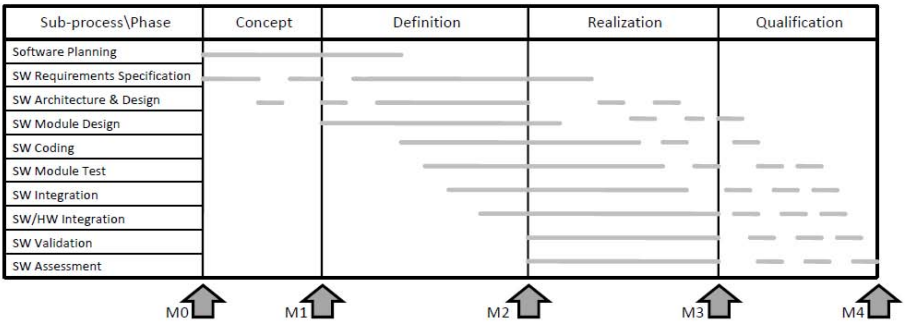

Fig. 8.2. Subprocesses and Phases in the OpRail-Process

8.4.5 The Use of UML in the Process

In general, UML diagrams can be used as parts of design artifacts, or even to replace some textual artifact completely. We indicate an elaboration which employs the most common diagram types to the various purposes.

Requirements. For the formulation of requirements (System Requirements Specification, SW requirements Specification etc.), the most suitable UML diagrams are *use case diagrams* to provide an overview and *sequence diagrams* to visualize behaviors. These can complement textual use cases (see e.g. [51]), as well as other, more traditional representation formats.

Architecture. An architecture may be visualized in a *structure diagram*, e.g. as a *component diagram* in the SW Architecture Specification, or as a *class diagram* on the level of module specifications. Depending on the nature of the design object and the level of abstraction, *sequence diagrams* illustrating the interaction of components may be useful.

Design/Module Specification. Detailed behavioral aspects can be specified with *state machines* or *activity diagrams*, while *class diagrams* define the software structure.

Test Specifications. Tests can be specified in *sequence* or *timing diagrams*.

The mapping above does not yet reflect the specific nature of a safety-critical design process. To be in line with the requirements coming from the safety criticality, one must observe that the diagrams are

- equipped with an unambiguous semantics, like for instance as given for SafeUML, see Sec. 8.3.3,
- embedded in a context completing the usually partial specification given by the annotated graphics,

– adequate in their level of detail to the development stage. For instance, it is difficult to formulate a global requirement in a state machine without referring already to a component of a particular implementation.

What makes the process specifically suitable for UML is its flexible phase structure which permits gradually refining models, switching between specification and prototypically implementing components or aspects for explorative purposes, and thus gaining a much clearer picture of characteristics of the remaining design space than with most other development approaches. In short, it gives a well-defined, norm-compliant elaboration for an agile software development style, taking full advantage of the expressiveness of models for rich specifications at different levels of abstraction, in particular in early stages.

8.4.6 Realization

There are several additional aspects to be taken into account when implementing the OpRail employing UML and UML tools.

Models as Documents. Traceability of requirements and accessibility of artifacts require specific measures to be taken to integrate diagrams into the development process. Often, model-based development environments come equipped with model management mechanisms. But these are, usually, not sufficient, so that integration with other tools (e.g., for requirement management) and further measures become necessary which may go to the point where all models used in the design are printed on paper.

Models as Specifications. While for ordinary development projects it makes sense to have different views of one and the same object at the same or at different stages of the development, which even need not be consistent, this is not acceptable in a safety-critical process. All inconsistencies and semantic ambiguities have to be ruled out. As already indicated, using a restriction defined for such purposes like Safe-UML from Sec. 8.3.3 is one possible approach for functional and communication aspects, while, for instance, Omega-RT from Sec. 8.3.2 is useful to address the real-time aspect.

Models and Code. If code is generated from models, this too has nontrivial ramifications in the context of safety-criticality. As one aspect, the relation between model and code has to be clarified. If the semantics of the model itself is given by such a translation, all arguments relying on the model must be rooted in the code it represents, which may be awkward. Otherwise, formal relations between the model and the code semantics must be established. One facet of this problem can be addressed by certifying the code generator as it has been done for SCADE. Another facet comes with the execution environment, where the development environment (e.g., a Windows PC) and the target system (e.g., a real-time operating system running on a small processor in a simple architecture) may differ considerably. And if the generated code is modified later, be it for reasons of efficiency or platform compatibility, this must be reflected in the model (e.g., via roundtrip engineering) or

addressed in the respective artifacts, the SW Module Verification Report, to name an example.

Models and Formal Verification. Complete models with an operational semantics permit the application of formal verification tools like theorem provers and model checkers. These, with their promise of assuring complete coverage of the model behavior seem attractive for high SILs. Indeed formal verification in the long run may further increase the usefulness of model-based design. However, currently the tools and methods are rarely mature enough for using them in industrial practice. Computational complexity and tool qualification are common obstacles. Nevertheless, formal verification techniques may already today help the designer in exploring models at abstract design stages, or specific components with great scrutiny, without making use of the result in safety cases. A more complete treatment of these issues and the following point can be found in Sec. 8.5.

Models and Tests. Test generation from models or models as components in test scenarios constitute another possible benefit of employing the model-based design paradigm. Mature techniques are available which can be put to use in the OpRail process. Generating tests covering models can partly automate the construction of the Requirement Test Specification or the Software Module Test Specification.

Taking into account all these issues, we conclude that it is possible to move from traditional design processes to one which profits from the use of models at different steps and levels of abstraction. The scheme provided by the OpRail process has been favorably evaluated by the TÜV Süd Rail as being suitable for an instantiation to a real-life process apt for the development of safety-critical rail applications.

8.5 Verification and Validation Techniques

Almost all safety-related UML profiles come along with a number of formally founded V&V techniques. It is far beyond the scope of this paper to present their technical background to a satisfactory level of detail. Instead, we discuss issues on the integration of formal verification techniques in the software safety assurance process for certifiable systems. As it is the case for the use of models in design, scope and objectives to be achieved by employing models in V&V activities have to be made explicit for safety assessment in certification processes. A verifiable, mathematical proof of a theorem on a formal model can serve as a piece of evidence for a safety claim only, if conclusive arguments are provided how the mathematical statement relates to the real-world system.

8.5.1 General Remarks on Verification and Validation Techniques in Model-Based Development of Certifiable Software

In accordance with the standards, we perceive a technique as *verification method* if it is adequate to evaluate whether or not the system or a component complies

to a specification or constraints imposed at a preceding development phase. A *formal analysis* technique is based on a mathematical model of the system and the requirements and uses mathematical deduction for reasoning. In many cases the reasoning can be mechanized. In case, the primary and immediate goal of the technique is to prove that the (sub-)system satisfies the specification, we call it a *formal verification technique* in the narrow sense; if it directly aims at disproving the conformance of the system or component to the requirements it is called an *error detection technique*. In this article, we use *testing* for V&V techniques that execute the (sub-)system with selected inputs and compare its outcomes with the expected ones. A *validation* technique increases our confidence that the system accomplishes its intended requirements.

Following the terminology of Dwyer [52], we call a technique *sound* if a positive result of the evaluation constitutes conclusive evidence that the stated claim holds. Thus, a sound method does not generate false positives[8]. In this sense, exhaustive state exploration is a sound formal verification technique on finite state system models and testing usually is sound for error detection, but not for verification because exhaustive testing is impossible in most cases. Pure bounded model checking is sound for error detection only, as the state space is explored to a limited depth. But enhanced with inductive reasoning it may be extended to a verification technique. In particular in the realm of safety-critical systems, the limitations of a verification technique have to be clarified carefully as the evidence provided by the reported analysis results can only have relevance in a safety argumentation if the technique is applied in a sound manner.

Furthermore, software verification and validation – whether model-based or not – do not prove that software will not contribute to serious hazards under any circumstances. The best what can be achieved is to demonstrate that the software accomplishes its functional and safety requirements that have been derived from the aggregated knowledge on the system, its environment, and the foreseen hazards.

With models as design artifacts new V&V techniques can be applied. If design models are executable, simulation of the functionality provides additional validation facilities already in design phases. In case models are formally founded, model checking, abstract interpretation and theorem proving offer a powerful formal verification tool kit that can be further enhanced with various abstraction heuristics or compositional reasoning. If safety-related, extra-functional characteristics like reliability or the error behaviour, real-time or performance are explicitly represented in the model, then these can be subject of the analysis, too. However, reliable data for extra-functional runtime characteristics are most often only available when software/hardware integration is finished. Hence, analyses performed at earlier design stages on the basis of estimated values have to be repeated to approve the results.

Additional V&V techniques are not only a possibility, but are also a necessity in a model-based development process for several reasons: First of all, manual

[8] While false positives principally compromise the value of a verification technique, false negatives may cause additional effort, but do not put the technique in question.

review and inspection as traditionally performed on text documents have to be significantly adjusted for models. Without denying the well-known deficiencies of textual documents like incompleteness, inconsistency, poor structure, and the lack of traceability, these problems are at most disarranged but not solved without effort in a model-based integrated development environment:

Comprehensibility of a model-based design can be negatively affected by aspects of the method, the modeling language and the tooling: The design is usually scattered over several views and kinds of diagrams. Moreover, UML is a rich notation that often offers a set of alternative modelings to express the same issue. The developer may not oversee all semantic interdependencies between complex issues like object creation or deletion, event handling, transition selection, or run-to-completion-steps, even if the semantics is precisely defined. Tools often hide the details in the top view on a diagram, like, e.g., attribute or method declaration in class diagrams or inner hierarchy levels in statecharts. Moreover, specific settings severely influencing the semantics are often accessible only via nested preference lists or attribute tables within internal model browsers. Another open issue is the accessibility of different versions of a model stemming from earlier design phases or abstraction levels within one model repository.

"Collateral validation", as the implicit team validation has been called by Heimdahl [10], is lost, if model-based development comes along with large scale automation: Traditional development processes of safety-critical systems involve a plurality of experts whose expertise covers a broad field ranging from domain knowledge and software architecture to detailed questions of process and communication integration and hardware drivers. In the V&V phases, test experts and validators have a good chance to identify the tender points in a design due to their experience. Model design encompasses tasks from the whole field and is performed by fewer developers who may not always distinguish all consequences.

Also from a more technical perspective, several issues have to be considered to provide a conclusive safety argument for a model-based development approach.

Model paradigms: Software design and verification models are based on a model of communication and computation (MoCC) defining an abstraction from physical time, the granularity of steps, a concurrency paradigm etc. These abstractions may be adequate on a certain level of abstraction and in certain contexts. On the level of implementation, the safety-critical software applications are going to be executed in an environment of real-time operating systems (RTOS) and communication protocols like IMA [53] or time-triggered protocols [54]. Only if these support safe abstractions to analyzable MoCCs – which is not always the case – one may develop the software applications independently from the RTOS. For applications themselves, an answer to the model abstraction problem is given by approaches that establish a direct correspondence between the formal model and the code like Safe-UML(P) (see Sec. 8.3.3 [18] or SystemC models in RCSD [49]).

Model content: It has to be justified by thorough model-based validation that the formalized description of the requirements in a model-based specification and their implementation in a design model meet the informal, intended

requirements. Only then a formal modeling framework can benefit from the enormous pool of techniques on model-based verification. Supplementary vacuity checks can assure the specification in fact covers the relevant behavior of the model (see Heimdahl for an overview [10]). Another caveat is the impact of simplifications and omissions: For scalability reasons or due to an early design stage, sub-systems or parts of the functionality are modelled very coarsely or omitted at all. Obviously, verification results have to be proven robust against such simplifications.

Backend questions: The more behavioral abstraction a modeling notation provides, the more is added in a code generation step that can only roughly be configured by the designer. In particular, extra-functional run-time properties like execution times and storage consumption may heavily depend on a prudent choice and combination of modeling elements.
Software-intensive technical systems are mostly assembled with proprietary hardware and operating systems for good reasons. But code generation of commercial modeling tools is optimized and approved for usage on standard processors. Thus, the code generator and linker have to be customized, which is a delicate task for specialists with joint expertise on the tool and the target system.

Tool qualification: The fundamental soundness requirements on tools offering early simulation, code generation, or formal verification are the *coincidence of the simulation, the verification, and the execution semantics* and *sound reasoning mechanisms*. If the execution semantics diverges from any of the other, or the deduction mechanism is corrupted, V&V results achieved on the basis of the simulation or verification semantics become worthless.

In contrast to many papers advertising verification techniques, successful industrial applications of formal model-based techniques mainly address detailed component design, not only for scalability reasons, but also for the validation needs and the caveats mentioned above. To benefit from formal verification and early simulation, a model must be precise and detailed with respect to all aspects that are the subject of verification. This can usually be carried out in the detailed design phase at the earliest.

8.5.2 Testing

Testing is the predominant V&V method applied in practice. For safety-critical systems, the standards explicitly recommend testing. Major test purposes are (1) to explore the functional specification in appropriate detail, (2) to execute the code to a sufficient degree of completeness, and (3) to ensure that the software is running properly on the target hardware. Therefore, a number of testing techniques are listed in the standards like testing based on equivalence partitioning of inputs, boundary value analysis or structural coverage criteria referring to data and control flow. Additionally, prototyping and animation for design validation, stress testing and exploratory or risk based testing are advised.

Executable design models pave the way to integrate testing activities in design phases: Well accepted test-selection criteria and the derived test-case specifications making such test criteria operational can be easily adapted to generate test suites that are applicable on the level of executable models instead on the code level. For instance, coverage criteria like statement, decision or MC/DC coverage have been transferred to statechart models [55].

This way, development fully benefits from providing executable models early in the process: Relevant shortcomings in the requirements specification can be detected before detailed design and costly implementation efforts are started.

In the following, we shortly discuss three specific approaches to adapt testing to model-based development:

(1) Design models from a previous development stage build the specification from which test cases are constructed.
(2) Test models are built independently from the artifacts used in the development and serve as basis for test case generation.
(3) Models are built by (automated) abstraction from code.

Test Cases Generated from Specifications

In the first approach, the current model or implementation is tested with respect to its conformance to a specification from a previous phase. Generating test cases from a previous design model can be applied iteratively at each stage to uncover deviations of the behavior of the current model from that of preceding models or artifacts. Detected deviations may have several causes:

- A preceding design step is flawed, but the specification is correct with respect to the original requirements.
- The preceding model or requirements specification is ambiguous or incomplete. This may concern the function to be realized, the execution platform and its limited resources or the assumptions on the environment. At some point, such an aspect may come into focus because the latter, more detailed model requires to settle it.
- The current model integrates different views or parts of the system that have been developed separately so far[9]. In such a step, testing conformance to the preceding separated models may reveal inconsistencies and erroneous assumptions that have been introduced into one of the preceding models.

However, as the test cases are derived from the same source as the current design model, this approach may support verification, but no independent validation. In other words, this approach uncovers inconsistencies that are inherent in the requirements specification itself or introduced in functional refinement steps. Mismatches between the functional specification, the execution platform and the environment can be detected only if the integrated models address these issues. But if for any reason the issue is not faithfully reflected in the design models this approach will not reveal any hint to a problem [56].

[9] Examples are functional composition or resource allocation and deployment in a layered architecture.

Independent Test Models

The second approach is to construct dedicated test models independently from the line of design models. Independence of the test model from the design models opens up new views [57] for the obvious price of additional effort:

- The test model may represent the system under development (SUT), modeled from the testing perspective, but also the system operator or the environment. Between these positions, numerous combinations of SUT and environmental models are possible that may apply various abstraction principles. For instance, a SUT test model may be restricted to the most common usage scenarios or a functional kernel, an environmental model may consists of a stochastic profile on input values and loads and their admissible ranges.
- Modeling formalisms differ with respect to the handling of fundamental concepts like time, causality, determinism, etc. Additionally, they provide different views and follow different computation paradigms like functional, operational, probabilistic or data-flow-oriented models.

The key issue of an additional test model is to complement the knowledge on the SUT from an independent perspective. As the test model is not a step in the design, it may be optimised for validation and verification purposes with respect to the functional and extra-functional requirements addressed, but also in terms of the concepts, paradigms and notational elements used for modeling. Heimdahl [10] reported on experiences that complementing a specification by several alternative models is considered a major factor towards achieving completeness.

Often, test models directly support derivation of test cases; alternatively, more general test generation techniques via the definition of test selection criteria can be adopted. Thus, building independent test models is adequate for all verification purposes mentioned in the previous paragraph, and it adds a chance to uncover defects that are outside the focus of the design artifacts, and extends the scope towards validation.

Models as Code Abstractions

A third use of models for testing safety-critical systems is to deduce a formal behavioral model as an abstraction from code and - if needed - a machine model. An intermediate representation can be extracted from source code by standard parsing techniques. The intermediate representation is symbolically interpreted on an abstract machine model. Thereby, constraints on the variables are collected and simplified by various techniques from abstract and concrete interpretation. Solving the constraints enables the generation of input values for a test case that covers a particular run through the model. The method supports the efficient generation of test cases for structural coverage criteria and boundary value analysis, but also the precise construction of test cases for certain classes of runtime errors. This way, testing the software/hardware integration can be transferred partially to the model level when using a refined hardware model. An testing technology based on this kind of models has been proposed by Peleska [58].

All three approaches to integrate testing in model-based development provide new prospects of design verification at an earlier stage than code integration on

the target platform. They do not eliminate the need for final tests by executing the implementation on the target system and showing that module, integration, system, and acceptance tests are passed. But they can shorten these activities and iterations in the design by uncovering errors earlier.

8.5.3 (Formal) Verification

In contrast to testing, the promise of formal verification is a hundred percent guarantee for compliance of an artifact with a certain claim. Though this may sound highly attractive, formal verification is still, even after forty years of thorough research, only used very rarely in the development of safety-critical systems. Some of the main obstacles can be summarized as follows.

(1) Incompatibility with the established design process.
(2) Limited scope and immaturity of the techniques and tools.
(3) Lack of skilled personnel.

We addressed the first and second point partly in Sec. 8.4. Here, we will elaborate more on the fundamental weakness of formal verification in practice, namely, that it firstly offers a proof in the mathematical sense and usually not in the juridical sense. This means that mathematical proofs are in most cases not easily usable in certification processes. This is due to several reasons.

(1) The proofs, if produced explicitly at all, are very large so that they cannot be checked manually.
(2) Tools which produce the proofs need to be verified or at least qualified themselves, what they usually are not.
(3) The statements proved are accessible only to specialists, and they are often difficult to interpret correctly.

We will exemplify these reasons by studying two proof techniques, model checking and theorem proving, and suggest remedies to these obstacles.

Model checking: Model checking is the name for mostly automatic proof routines which check whether the set of behaviors of a program (its runs) are a *model* for a specification formula, i.e., whether the runs satisfy the specification. There is a multitude of different model-checking algorithms and implementations. *Explicit* model checkers enumerate systematically execution states of the program, *symbolic* and *SAT-based* model checkers operate on logical representations of states. Common to most model checkers is the requirement that the examined program has only finitely many states (maybe after abstraction), and model checking consists in cleverly covering all relevant cases. The problem with this approach is that model checking is not intended to produce a proof – if the system is correct, its output may simply be "yes".

This is of course of not much help in convincing a certification authority of the correctness of a particular statement. Since model checkers are complex programs and efficiency is a major issue in their design, they are themselves hard to verify. A very appealing way to overcome this hurdle is the following.

(1) Extend the model checker to produce a proof. Such a proof might be rather large, but will likely employ only simple constructs – codes for finite sets, boolean representations and so on.
(2) Design a tool to check those proofs. It is easier to check proofs then to construct them, and checking has only to be performed on the proofs relevant to the safety case – namely on the final versions of design artifacts. Therefore, such a tool need not be as efficient and elaborate as the prover itself, so it can be much simpler and will be easier validated.
(3) Apply proof generation and proof checking for the program version to be certified. Previous versions which have been produced during the development need not be treated as thoroughly. Though formal verification may be applied to them, the verification itself need not be checked.

Examples of how to extend different model checkers can be found in [59, 60], though we are not aware of any realization used in practice. The reader may consider the experiences with model checking Rhapsody in Sec. 8.5.4 to see reasons for this state of affairs.

Theorem proving: Theorem proving offers a flexible way for machine assisted proof of complex verification problems, see for instance the impressive achievements of the VeriSoft project [61, 62]. In principle, theorem provers construct proofs, but there are two caveats: First, the proofs are constructed on the fly, that means, they are not intended to be stored. Second, usually a theorem prover for software verification employs automatic subroutines to increase efficiency. As neither the theorem prover implementation nor its automatic subroutines (nor the computer it runs on) are themselves verified, they face similar difficulties as model checkers when certification is concerned. And approaches to overcome the difficulties rely on similar means: Coq (http://coq.inria.fr/) has a small "certification kernel" to check proofs, as does the Boyer-Moore theorem prover.

Summarizing, while the confidence in a system's correctness may be greatly improved through formal verification, its practical value today is still limited: The effort involved is high, and the results, if achieved at all, are not readily usable for system certification. Considering the remarkable progress made in this field in the recent decade, we expect that these methods will gain importance in the future.

8.5.4 Tool Support

In this section we give two examples of tools offering V&V support for UML, the tool set ATG for automatic test generation [63] and the model checker RUVE (Rhapsody UML Verification Environment) [64]. They both are extensions to the "Rhapsody®[10] in Cpp" design environment. Rhapsody in Cpp has an elaborated `Code Generator` which produces Cpp code from models consisting of class diagrams (for structure) and state machines or activity diagrams (for behavior)

[10] Now IBM, formerly Telelogic, formerly i-Logix.

which carry conditions and statements formulated in Cpp. A `Simulator` permits to execute the resulting code with user input for external stimuli. There is no animation of models besides the one through Cpp, other for instance than in the case of Statemate®. Therefore, one may view the Cpp code as the semantics of a model.

We continue now with a presentation of the extensions and a discussion of their suitability for a safety-critical development.

Test Generation

The architecture of the test-generation extension (ATG for *Automatic Test Generation*)[11] to Rhapsody is depicted in Fig. 8.3. A simplified view of the Rhapsody environment is on the left of the figure, with `Model Constructor`, `Code Generator` and `Simulator` as its main components for our presentation. When an executable model has been constructed, the user can select a part of the model as a *System Under Test* (SUT) and provide test goals. These latter can be expressed on the level either of the Cpp code or of the model itself. The most important code-level testing goal is MC/DC, in model terms one may ask for covering all states and transitions of the state machines in the SUT. In our simplified description, we ignore details like code instrumentation (e.g. to observe coverage in terms of the model) and the issue of the environment of the SUT which is very important in practice. The goals are fed into the `Test Generator`. The generator outputs a set of test sequences driving the model according tothe specified goals, together with expected reactions of the SUT. It also reportson the degree of coverage achieved by the set, which is not always hundred percent.

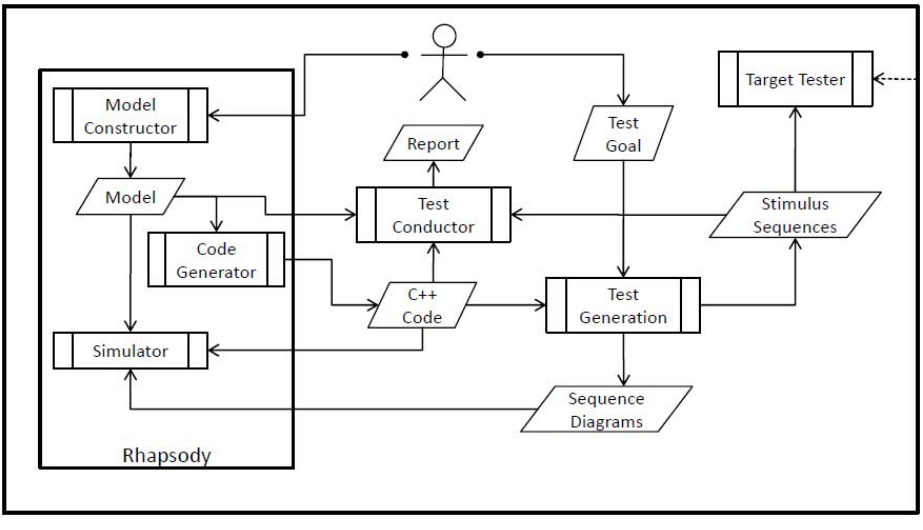

Fig. 8.3. Architecture of the Test-Generation Extension to Rhapsody

[11] Developed and marketed by BTC-ES.

The sequences may be exported in the form of sequence diagrams for visualization through the simulator. They can be applied to the model itself, or a different revision of the model, via the `Test Conductor` which, in the case of a regression test, will report any deviation from the expected behavior. Or they may be exported to other test-execution environments, e.g. for the purpose of testing compliance of target code or target system to the model. If the coverage is insufficient, the generated test suites may be completed by manual effort.

ATG can be applied in implementing the first two of the approaches to adapt testing to model-based development: Test cases generated from a model may be run on a model from a later stage, or even the target itself (first approach), or one may generate the test cases from specifically constructed model for covering certain aspects (second approach). With respect to standard requirements in safety-critical design, the generated test cases can be used for functional and black-box testing, or, with specific test models, also for non-functional aspects like timing.

Qualification is of course an issue. The test generator is the most advanced component of the ATG extension. It works by symbolically executing the Cpp code. Fortunately, the test generator itself does not have to be validated. Instead, one may independently – using a much simpler tool – validate the coverage achieved. The test-cases conductor is more critical. If one relies on automatic execution, the environment performing the tests has to be validated. Alternatively, one may add extensive reporting to provide evidence for correctness and completeness of the test execution.

Summarizing, the ATG extension to Rhapsody provides tool support to automate part of the testing activities which consume a substantial percentage of the development effort. Thus, it adds to the advantages of model-based design over traditional methods. Taking adequate provisions, ATG may even be applied when developing a safety-critical system.

Model Checking

The architecture of the model-checking extension is depicted in Fig. 8.4. Cpp code generated from Rhapsody models is translated into an automaton format and fed into a model-checking engine, whose other input may come from a number of different specification formats (graphical, pattern-based, temporal-logic formulae). The model-check engine is based on VIS, i.e. it is a symbolic model checker employing BDDs (Binary Decision Diagrams)[12]. Its output is either the message that the model satisfies the specified requirement (“`true`”) or an error path which may be animated on the model or visualized as a graphical specification.

Other than the ATG extension, RUVE is limited in the input it can process. Some of the limitations are inherent, others are founded in the experimental, not yet mature state of the tool set. Since semantically, the input to the model checker must be a finite automaton, dynamic object creation must be limited by static bounds. Also, the component and association structure cannot be changed

[12] A SAT-based engine may be used as an alternative.

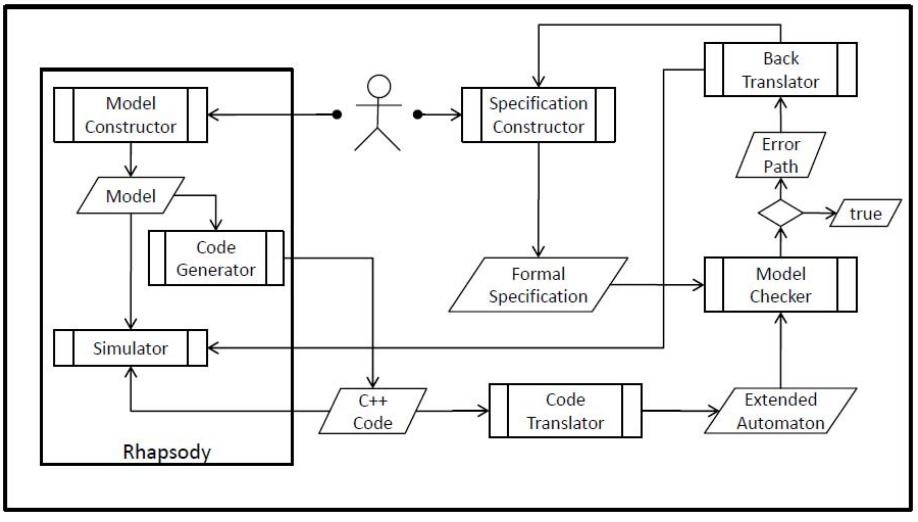

Fig. 8.4. Architecture of the Model-Checking Extension to Rhapsody

freely. Floating point numbers are not permitted for complexity reasons, and pointer operations are restricted. What remains is still a large and practically usable subset of the features of Rhapsody in Cpp.

The main usage of model checking is the verification of properties, which are, despite the different input formats available to the user, in their essence temporal-logic formulae. Model checking thus offers a way to verify properties which are predicative in nature. It is less suited to show refinements, e.g. that some model implements another, more abstract one. A second use case employs the error-path feature for design animation. For the specification "`Always Not In(S)`", the model checker will output a path leading to the state `S` if `S` is reachable. In that way, one may explore a model with automatic support. Further usages employ instrumented models.

If one wants to use the model checker for direct verification, the qualification question becomes important. There are two main issues.

First, the model checker does not work on the Cpp code (which, as already discussed, can be seen as the true semantics of the UML model), but on an automaton which has been generated by a nontrivial translation process. One way to ensure the correctness of this step is to employ methods from compiler validation. One could either validate the `Code Translator`, or independently verify that it worked correctly on those models appearing in the safety case.

Second, one will have to verify the operation of the `Model Checker` component. Since it seems impractical to certify this engine – the model checker works on advanced data structures and is geared towards efficiency – the best option is to verify that it worked correctly in the invocations relevant to the safety case. In Sec. 8.5.3 we explained that a promising way is to equip the model

checker with a feature that produces on request a proof for the answer "`true`". Though this has not been done for a BDD-based model checker as far as we know, there is no fundamental difficulty involved. A proof will be a large object, but a rather simple procedure would be able to check its correctness.

These obstacles prevent the practical application of RUVE in a safety-critical development today. In addition, the resource consumption of RUVE is high already for rather small UML models. One may say that UML state machines – at least in this Rhapsody implementation – are a rather difficult object for formal verification. As a result, RUVE can be applied successfully only in specific cases, e.g. to aid the designer in analyzing a small model thoroughly. This may either be an abstract model at an early design stage, or a model of a component like one implementing a protocol for which model checking is particularly suited. Other than test generation, model checking does not seem mature enough for industrial application in UML-based design. Design for verification (i.e., language restrictions), dedicated procedures and additions for certification seem to be called for to enable profitable use of model checking on a larger scale.

8.6 Conclusion

UML and its extensions offer modeling elements for most aspects of interest in the context of safety-critical software and system development. A major advantage of UML is its wide dissemination for general purpose software development, while e.g. domain-specific languages face communication barriers due to their limited user basis. It is also clear that by introducing UML into the safety-critical systems domain not any UML expert will automatically become a software safety engineer.

Model-based software development is not yet considered in the safety-critical system standards. But statutory obligations and best practices recommend design qualities like structuredness, simplicity, and preciseness that are among the key promises of model-based development. Considering the plurality and complexity of UML, UML profiles and variants providing a well defined set of views and a restricted set of notational elements with precise (formally defined) semantics seem to be well suited for safety-critical systems. Upon the notational basis, a consistent set of techniques supported by qualified tools has to be defined that facilitates integration of UML models as key artifacts in a software safety design process. Limited maturity of techniques and tools as well as lack of elaborated methods supporting specific safety strategies are still severe hurdles for the proliferation of UML in industrial safety-critical software design.

Verification and validation benefit a lot from executable models in early design phases. Formal verification and testing contribute with a new quality of rigour and completeness to verification efforts. Recent research in this field has achieved substantial progress towards real-world size models and integration into industrial design processes. By customizing verification techniques to particular UML-based modeling frameworks, an equivalence between the verification model and the generated code executed on an abstract machine has been established

for several tool environments. Such approaches are pathbreaking with regard to the rigour in establishing functional correctness. In addition, the adaptation of static analysis techniques to UML models enables assurance of most statically verifiable properties in principle. But in the whole, tool support is still fragmented, scalability is limited, and skilled personnel is rare.

Additionally, certification aspects have yet to be addressed adequately: A number of formally founded approaches still lack an explicit and conclusive argument on how mathematically proven facts relate to the properties of the real system, its software components, and the software design artifacts. Despite advances proposing solutions to particular aspects and sketches of integrating them with the software safety analysis sub-process, this still need to be improved for most UML-based techniques. This applies for design-centered methods as well as for V&V-centered ones.

Therefore, we may conclude that a lot of useful progress has been made. And while no mature, consistent methodology has been found yet, with prudent choice of techniques and tools, employing UML can improve the development of safety-critical systems in practice today.

References

[1] Leveson, N.: Safeware - System Safety and Computers. Addison-Wesley, Reading (1995)
[2] Lutz, R.: Software engineering for safety: A roadmap. In: FOSE 2000: Future of Software Engineering, Washington, DC, USA, pp. 137–152. IEEE Computer Society, Los Alamitos (2000)
[3] McDermid, J.A., Pumfrey, D.J.: Software safety: Why is there no consensus? In: 19th International System Safety Conference, System Safety Society (2001)
[4] European Committee for Electrotechnical Standardization: EN 50128: Railway applications - communications, signaling and processing systems - software for railway control and protection systems (2001)
[5] Radio Technical Commission for Aeronautics (RTCA): Software Considerations in Airborne Systems and Equipment Certification (December 1992)
[6] Intern. Electrotechnical Commission: IEC 61508: Functional safety of electrical / electronic / programmableelectronic safety-related systems (1998)
[7] Federal Aviation Administration: System Safety Handbook (2008)
[8] Bernardi, S., Merseguer, J., Petriu, D.C.: Adding dependability analysis capabilities to the MARTE profile. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 736–750. Springer, Heidelberg (2008)
[9] Bernardi, S., Merseguer, J.: A UML profile for dependability analysis of real-time embedded systems. In: Proceedings of the 6th International Workshop on Software and Performance (WOSP), pp. 115–124 (2007)
[10] Heimdahl, M.P.E.: Safety and software intensive systems: Challenges old and new. In: FOSE 2007: Future of Software Engineering, Washington, DC, USA, pp. 137–152. IEEE Computer Society, Los Alamitos (2007)
[11] Intern. Electrotechnical Commission: 65A/524/CDV: IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems part 3: Software requirements, Committee Draft for Voting (2008)

[12] Esterel Technologies: Scade 6.0 (2008)
[13] McDermid, J.A., Nicholson, M., Pumfrey, D.J., Fenelon, P.: Experience with the application of HAZOP to computer-based systems. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) Abstract Data Types 1995 and COMPASS 1995. LNCS, vol. 1130, pp. 37–48. Springer, Heidelberg (1996)
[14] Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
[15] Hennessy, M.: Algebraic Theory of Processes. MIT Press, Cambridge (1988)
[16] Zoughbi, G., Briand, L., Labiche, Y.: A UML profile for developing airworthiness-compliant (RTCA-DO-178B) safety-critical systems. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 574–588. Springer, Heidelberg (2007)
[17] Hooman, J., Kugler, H., Ober, I., Votintseva, A., Yushtein, Y.: Supporting UML-based development of embedded systems by formal techniques. Software and System Modeling 7(2), 131–155 (2008)
[18] Hungar, H., Robbe, O., Wirtz, B.: Safe-UML - Restricting UML for the development of safety-critical systems. In: Schnieder, E., Tarnai, G. (eds.) Proc. FORMS/FORMAT 2007, pp. 467–475 (2007)
[19] Object Management Group: UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), Beta 2 (2008)
[20] Berkenkötter, K., Hannemann, U.: Modeling the railway control domain rigorously with a UML 2.0 profile. In: Górski, J. (ed.) SAFECOMP 2006. LNCS, vol. 4166, pp. 398–411. Springer, Heidelberg (2006)
[21] Object Management Group: SysML Specification Version 1.1 (2008-11-02) (November 2008), http://www.omg.org/spec/SysML/1.1/
[22] ATESST2: EAST-ADL2 Profile Specification (January 2008)
[23] Kelly, T.: Arguing Safety – A Systemic Approach to Managing Safety Cases. PhD thesis, University of York (September 1998)
[24] ISO TC22/SC3/WG16: Road Vehicles – Functional Safety. Committee Draft (September 2008)
[25] Telelogic: Rhapsody (2008)
[26] Eclipse Modeling Framework Project, EMF (2008), http://www.eclipse.org/modeling/emf/
[27] Graydon, P.J., Knight, J.C., Strunk, E.A.: Assurance based development of critical systems. In: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 347–357. IEEE Computer Society, Los Alamitos (2007)
[28] Damm, W., Josko, B., Pnueli, A., Votintseva, A.: A discrete-time uml semantics for concurrency and communication in safety-critical applications. Sci. Comput. Program. 55(1-3), 81–115 (2005)
[29] Harel, D., Marelly, R.: Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2003)
[30] Bozga, M., Graf, S., Mounier, L.: If-2.0: A validation environment for component-based real-time systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 343–348. Springer, Heidelberg (2002)
[31] Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
[32] Object Management Group: UML2.0 superstructure specification (2005)
[33] Eisenbahn-Bundesamt: Technische Grundätze für die Zulassung von Sicherungsanlagen (1999)

[34] Guidelines for the use of the language C in critical systems (2004)
[35] Sanders, R.: Rhapsody 6.0 properties, Technical report, OSC-ES, Oldenburg, Germany (2006)
[36] Robbe, O.: Analysis of the Rhapsody C++-code and framework according to compliance with the EBA-guidelines 42720 and 42730. Technical report, OFFIS, Oldenburg, Germany (2005)
[37] Object Management Group: UML Profile for Schedulability, Performance, and Time (SPT), Version 1.1 (2005)
[38] Papyrus for UML (2009), `http://www.papyrusuml.org`
[39] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System level performance analysis - the SymTA/S approach. IEEE Proceedings Computers and Digital Techniques 152(2), 148–166 (2005)
[40] Thiele, L., Chakraborty, S., Naedele, M.: Real-time calculus for scheduling hard real-time systems. In: International Symposium on Circuits and Systems (ISCAS), vol. 4, pp. 101–104 (2000)
[41] Hagner, M., Huhn, M., Zechner, A.: Timing analysis using the MARTE profile in the design of rail automation systems. In: 4th European Congress on Embedded Realtime Software, ERTS 2008 (2008)
[42] Object Management Group: UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification, Version 1.1 (2008)
[43] Espinoza, H., Dubois, H., Gérard, S., Pasaje, J.L.M., Petriu, D.C., Woodside, C.M.: Annotating UML models with non-functional properties for quantitative analysis. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 79–90. Springer, Heidelberg (2006)
[44] Avizienis, A., Laprie, J.C., Randell, B.: Fundamental concepts of dependability. Technical Report LAAS Report no. 01-145, UCLA, LAAS-CNRS, Univ. of Newcastle upon Tyne (2001)
[45] Pataricza, A.: From the general ressource model to a general fault modeling paradigm? In: Jürjens, J., Cengarle, M.V., Fernandez, E.B., Rumpe, B., Sandner, R. (eds.) Critical Systems Development with UML – Proceedings of the UML 2002 workshop, TU München, Institut für Informatik, pp. 163–170 (2002)
[46] Thomas, F., Delatour, J., Terrier, F., Gérard, S.: Towards a framework for explicit platform-based transformations. In: 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), pp. 211–218. IEEE Computer Society, Los Alamitos (2008)
[47] Graf, S., Ober, I., Ober, I.: A real-time profile for UML. International Journal on Software Tools for Technology Transfer (STTT) 8(2), 113–127 (2006)
[48] Berkenkötter, K.: OCL-based validation of a railway domain profile. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 159–168. Springer, Heidelberg (2007)
[49] Haxthausen, A., Peleska, J., Große, D., Drechsler, R.: Automated verification of train control systems. In: Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT), pp. 252–265 (2004)
[50] Hungar, H., Bruhns, G., Plan, O., Lemke, O.: OPRAIL - Normenkonforme Entwicklung sicherheitsrelevanter Software unter Einsatz der UML. Signal + Draht 7 (2007)
[51] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Reading (2000)
[52] Dwyer, M.B., Hatcliff, J., Robby, P.C.S., Visser, W.: Formal software analysis emerging trends in software model checking. In: Briand, L.C., Wolf, A.L. (eds.) Workshop on the Future of Software Engineering (FOSE), pp. 120–136 (2007)

[53] Lewis, J., Rierson, L.: Certification concerns with integrated modular avionics (IMA) projects. In: Digital Avionics Systems Conference (DASC). IEEE, Los Alamitos (2003)
[54] Kopetz, H., Grünsteidl, G.: TTP - a protocol for fault-tolerant real-time systems. IEEE Computer 27(1), 14–23 (1994)
[55] Mücke, T., Huhn, M.: Minimizing test execution time during test generation. In: IFIP Working Conference on Software Engineering Techniques (SET 2006). Springer, Heidelberg (2006)
[56] Pretschner, A., Philipps, J.: Methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 281–291. Springer, Heidelberg (2005)
[57] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. Working Paper 04/2006, Department of Computer Science, The University of Waikato (2006)
[58] Peleska, J.: A unified approach to abstract interpretation, formal verification and testing of C/C++ modules. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigün, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 3–22. Springer, Heidelberg (2008)
[59] Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, pp. 10880–10885. IEEE, Los Alamitos (2003)
[60] Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
[61] Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N.W., Starostin, A.: The Verisoft approach to systems verification. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
[62] Beyer, S., Jacobi, C., Kroening, D., Leinenbach, D., Paul, W.: Putting it all together: Formal verification of the VAMP. International Journal on Software Tools for Technology Transfer 8(4-5), 411–430 (2006)
[63] Lettrari, M.: Using abstractions for heuristic state space exploration of reactive object-oriented systems. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 462–481. Springer, Heidelberg (2003)
[64] Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML Verification Environment. In: Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Bejing, China, pp. 174–183. IEEE, Los Alamitos (September 2004)