MAGISTERARBEIT

Requirement Classification of Dependable Real-Time Systems

unter der Leitung von

Ao. Univ.-Prof. Dr. Peter Puschner
Institut für technische Informatik 182/1

und als verantwortlich mitwirkenden Universitätsassistenten

Univ.Ass. Dr. Raimund Kirner
Institut für technische Informatik 182/1

eingereicht an der
Technischen Universität Wien,
Fakultät für Informatik

durch

Josef M. Trojer, Bakk.techn.
Matr.-Nr. 9908365
1070 Wien, Neustiftgasse 53/2/37

Wien, im Jänner 2007 .....

Abstract

Implementing safety in real-time systems today requires time-consuming and cost-intensive certification procedures recommended by domain-specific safety standards. Additionally, meeting tight time-to-market deadlines in implementing increasing system complexity forces industry to find comprehensive ways to reduce the effort expended on developing system artifacts.

This master's thesis introduces a strategy for the development of dependable real-time systems that encapsulates the reuse potential of system behavior on the basis of requirements specifications. To this end, the platform-oriented concept of the model-driven architecture (MDA) is used to separate dependable system behavior on the basis of a platform-specific and platform-independent viewpoint. When it comes to integrating safety, a systematic development of requirements according to the RTCA/DO-178B guidelines helps to specify safety-critical behavioral aspects of a real-time system by using high-level and low-level requirements. Combining the viewpoint principles of the MDA with the systematic two-level requirements development according to RTCA/DO-178B is the basic strategy for the reuse of requirements, with a requirements classification pattern (RCP) dividing the dependable system behavior into four requirements classification windows (RCW).

The assignment of requirements to RCWs is based on a platform-oriented analysis of system properties. The requirements classified in this thesis are stated in natural language. Each system property belongs to a platform layer, with a platform layer providing a manageable reusable insight of the system platform behavior. Thus, a platform layer, with its container-related characteristics, limits the scope of the RCP within the same technological domain.

In order to demonstrate the practical relevance of the reuse strategy mentioned, an RCP developed for this thesis was applied to a set of sample software requirements of TTP-OS, a safety-critical, fault-tolerant real-time operating system developed according to RTCA/DO-178B as a certifiable software product for the aerospace industry. As the use of natural language for the specification of requirements often is ambiguous, so-called informal guidelines are used to support a platform-oriented property assignment decision. A reusable distribution reflects a majority of assigned requirements in the platform-independent RCWs. Changing requirements from being platform-specific to being platform-independent shows a reuse-driven approach to developing requirements for dependable real-time systems.

Danksagung

Einleitend möchte ich mich bei Prof. Peter Puschner bedanken, welcher es mir ermöglichte, meine Diplomarbeit am Institut für technische Informatik zu verfassen. Vor allem jedoch zolle ich großen Dank Dr. Raimund Kirner für dessen gewissenhafte und exzellente fachliche Betreuung, die wesentlich zur Fertigstellung meiner Diplomarbeit beigetragen hat.

Karl Salasch und Martin Schlager danke ich für wertvolles und essentielles Feedback in sämtlichen Diskussionsrunden und natürlich für deren fundiertes Korrekturlesen.

Ebenso möchte ich der Firma TTTech Computertechnik AG für deren technischen Unterstützung hinsichtlich Unterlagen zu TTP-OS, sowie den hervorragenden arbeitstechnischen Bedingungen speziell bei der Fertigstellung meiner Diplomarbeit danken.

Diese Arbeit widme ich meinen Eltern Josef und Cäcilia, sowie meinen beiden Brüdern Thomas und Reinhard.

Contents

Chapter 1

Introduction

The safety-critical domain considers a software failure as hazardous system situation that can cause a potentially serious harm to its environment. Thus, the determination of safety requirements, which specify the *safe* behavior of an entire system to be developed, helps to prevent errors from occurring in that system.

Software safety is based on systematic software development. The term *software safety* is already applied to an early stage of software development, when software requirements are being specified. At this stage, safety can be proved by following domain-specific safety standards defining guidelines and recommendations for the engineering of requirements, depending on the respective safety degree desired.

Software development has to undergo certification when it is necessary to prove that a requirements specification complies with the relevant safety standards to a sufficient degree. Depending on the safety approach applied, a slight modification of requirements can result in an unavoidable re-certification of parts of the artifacts developed if not of the entire system itself, which can cause an enormous increase in cost-intensive and time-consuming certification activities.

One way to overcome this problem is to reuse software. Software reuse is aimed at developing software on the basis of existing domain-specific system artefacts and development of know-how rather than creating an entire system from scratch [Kru92]. An existing requirement specification constitutes an adequate system artifact and can therefore act as core instrument in finding an appropriate solution to a certain problem.

This thesis introduces a generic requirement classification pat-

tern (RCP) that can be used to specify requirements for safety-critical software. Before requirements can be classified, a platform layer defining classification and domain boundaries has to be specified. Then RCP can be used to separate natural language (NL) requirements into four requirement classification windows (RCW), depending on the semantic information those NL requirements contain. The four RCWs are the result of a conceptual combination of high-level and low-level requirements specified according to the guidelines laid down in RTCA/DO-178B [Rad92] and the reuse-oriented, platform-independent and platform-specific viewpoint of the model-driven architecture (MDA) [MM03]. The RCP makes it possible to give reasons for reusing requirements specifications. The RCP also shows how to divide a software behavior in terms of so-called platform layers in order to enable a platform-oriented approach to the engineering of requirements for safety-critical software

For this master's thesis, the RCP has been applied to the requirements specification documents of an existing certifiable software product, TTP-OS [TG94], which is a fault-tolerant, distributed real-time operating system already used in safety-critical domains (for examples, aerospace).

1.1 Scope

This master's thesis covers the following topics:

- **Fundamentals** (Chapter 2 on page 4), outlining the essential concepts required for the better understanding of this thesis and discussing safety-critical aspects (including dependability), system safety and software safety. This section, moreover, classifies real-time systems, explains its distributed aspect and introduces formal methods as a potential way to verify and validate requirements for safety-critical software

- **Related Work** (Chapter 3 on page 22), listing related work done in the field of software reuse and classification by identifying approaches to information structures, by creating safety-critical software components, by presenting text-processing strategies reused for natural language (NL) requirements and by describing a formal pattern for real-time systems. Classification schemes contribute to the concept of safety-critical applications, listing two derived standards (IEC 61508 and

RTCA/DO-178B) and a related guideline concept for implementing *safe* programs.

- **Requirements and Safety** (Chapter 4 on page 33), describing the relationship between safety and requirements by providing a solution to the safety problem and its impact on the development of safety requirements. This section also answers the question of how system and software requirements can be developed when system safety is of particular importance. Finally this section describes the techniques that are used to uncover hazards, the root of unsafeness.

- **Application-Oriented Classification of Software Safety Requirements** (Chapter 5 on page 49), introducing the generic requirement classification pattern (RCP), which first determines a system platform layer and then splits up conceptual information into a platform-specific and platform-independent content. The determination of a system platform layer starts with a platform description of the respective real-time system, providing detailed information on corresponding systems, extending the platform-oriented view of dependable real-time systems and concluding its definition by discussing the impact of software safety on platform layers. The requirement classification resulting from that determination divides a set of requirements due to its system properties, determines rules for the intrinsic classification of those properties and summarizes the results as axioms of requirements development.

- **Evaluating Requirements Classifications for Safety-Critical Software** (Chapter 6 on page 64), applying the concept of the RCP to a safety-critical time-triggered software product. The applicability of the RCP is discussed with regard to system behavior modeling, its reuse potential and requirements development.

Chapter 2

Fundamentals

This chapter covers the fundamental concepts of this thesis. A distributed solution to real-time systems, for which we propose fundamental requirements, extends the criteria and classifications defined for real-time systems. The concept of dependability serves as a basis for describing safety, with that description being then put into the context of system and software development. Finally, we integrate the requirements specification into the system development process by discussing validation and verification issues for safety-critical system.

2.1 Real-Time Systems

2.1.1 The Concept of Real Time

"A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced [. . .] If a real-time computer system is distributed, it consists of a set of (computer) nodes interconnected by a real-time communication network [Kop97]."

An entire real-time system can be grouped into three components (subsystems):

1. the *operator*,
2. the *controlled object*, and
3. the *real-time computer system*.

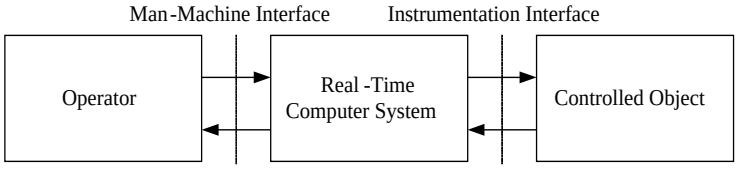


Figure 2.1: Conceptual Model of a Real-Time System [Kop97]

These subsystems are also called *clusters*. The operator cluster and the controlled object cluster constitute the *environment* of the real-time computer system. A real-time system is additionally characterized by two interfaces:

1. The *man-machine interface*, located between operator and real-time computer system has input and output devices.
2. The *instrumentation interface*, located between real-time computer system and controlled object, contains actuators and sensors for transforming physical signals into digital ones and vice versa.

Data processing within clusters and between interfaces, as shown in Figure 2.1 on the current page, is done by methods for scheduling and resource management. All kind of resulting *work* is computed and communicated within the real-time computer system domain. The resulting sequential execution of a program within a real-time system is called a *task*. A task not comprising an internal state at its point of invocation is a *stateless* task, otherwise a *task with state* [Kop97]. A set of n tasks is referred as *job* [HHCB06]. With respect to the time domain, the timely execution of tasks is fundamental aspect of any RT system.

"A real-time computer system must react to stimuli from the operator within time intervals dictated by its environment. The instant at which a result must be produced is called a deadline. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If a catastrophe could result if a firm deadline is missed, the deadline is called hard [...] A real-time computer system that must meet at least one hard deadline is called hard real-time computer system or a safety-critical real-time computer system. If no hard real-time deadline exists, then the system is called a soft real-time computer system [Kop97]."

Based on the temporal characteristic of a task, a deadline violation may imply a hazardous situation for the system. Thus con-

siderations on the correct execution of tasks affect the whole RT system.

A variety of examples for real-time systems can be found in the domain of embedded systems.

2.1.2 Classification of Real-time Systems

In addition to the fundamental real-time concepts of hard, firm and soft timing constraints, this section describes ways how to classify real-time systems [Kop97].

Event-Triggered versus Timed-Triggered

An *event* is an occurrence of an happening on the time-line. It connects the past with the future. The occurrence of an event starts some action, for example, the execution of a task. This occurrence is also considered as a *trigger* of an action. Communication and processing activities depend on the trigger mechanism used by the real-time computer system. Dealing with triggers in the real-time domain results in two completely different design approaches.

In the *event-triggered* approach, system activities depend on the events occurring in the computer system or environment. Dynamic scheduling strategies used in the real-time computer system handle the interrupt mechanism reflecting the signaling of significant events. In the *time-triggered* approach, all system activities are triggered by the progression of the global time and started periodically at predetermined instants [Kop93]. Static scheduling strategies offer a synchronous interrupt mechanism delivering signals at predefined points of the global time.

Fail-Safe versus Fail-Operational

The execution of applications on real-time systems often includes the concept of state machines in hardware and software. The occurrence of malicious events have a significant impact on the correct behavior of the system. If the system immediately enters a safe state, after malicious states have been detected, the system is called *fail-safe*. A real-time system holding a safe state does not have a hazardous effect on the environment. Thus, a fail-safe system does not provide the environment with any additional functionality. Nevertheless, some systems have to provide a minimal level of

service even if a hazardous situation or failure occurs. These systems are called *fail-operational* (for example, flight-control systems in an aircraft).

Guaranteed Response versus Best Effort

Designing a real-time system faces engineers with two implications of integrating the fault and load hypothesis, namely *guaranteed response* and *best effort*. A real-time system is characterized by *guaranteed response* when the design of that system neglects probabilistic arguments, even in the case of peak load and fault scenarios. Careful planning and analyzing is indispensable for creating a guaranteed-response system. If the system design does not have guaranteed response, this implies a *best-effort* design method when the system has been implemented. Engineers design the system using the statement *"best possible effort taken"* as their fundamental motto.

Resource-Adequate versus Resource-Inadequate

The concept of *resource adequacy*, which nowadays is integrated in many designs of safety-critical systems, enables a correct handling of rare-event scenarios by providing enough system resources. Non-safety-critical systems do not have to provide adequate resources. They also do not have to satisfy extraordinary fault and load scenarios in a sufficient way. Thus, this kind of real-time systems reflects *resource inadequacy*.

2.2 Distributed Real-Time Systems

To overcome the increasing complexity of real-time systems, the entire system functionality is divided into modular subsystems. The safety-critical domain makes use of the decomposition principle and its effect on system development. This design consideration is sustained by the fact that using a centered real-time system never achieve a overall system failure rate of 10^{-9}, which reflects a safety-critical behavior [Sur94]. The design of distributed real-time systems has to show compliance with fundamental communication requirements and real-time system properties.

2.2.1 Real-Time Entity and Real-Time Image

A *real-time (RT) entity* per definition is a *"state variable of relevance for the given purpose [Kop97]"* and plays an important role in the efficient description of real-time system behavior. RT entities reside within a computer system or its environment and have a logical or physical interpretation. RT entities are characterized by static and dynamic attributes, for example, the name for an entity is considered to be static, while dynamic attributes change their values at a certain point in time. Each RT entity comprises an internal state and has a close relationship to its *real-time image*.

An RT image per definition is the *"current picture of an RT entity [Kop97]."* For the behavior of a system it is necessary to know in which (internal) state the distributed real-time system resides. Therefore an RT entity is observed at a certain point in time and evaluated by

$$Observation = <Name, t_{obs}, Value>$$

The *atomic data structure* of an observation reflects the moment of observation t_{obs} with respect to time and the resulting $Value$. An arbitrary time-based observation can only be done with a continuous RT entity. An RT entity or RT image is stored in a *real-time object* on a node within the distributed real-time system network. *Temporal accuracy* defines a temporal relationship between an RT entity and its RT image of an RT Object within a corresponding real-time application.

"A RT-image is temporally accurate if the time interval between the moment "now" and point in time when the current value of the real-time image was the value of the corresponding RT entity is smaller than an application specific bound"

2.2.2 Real-Time Communication Requirements

The entire functionality of a distributed system is determined by a set of nodes interacting with each other according to specific rules. A computational node located within a distributed real-time system typically comprises a host processor and a communication controller. The entire system behavior is based on a variety of communication flows taking place between these nodes. These communication flows are controlled by systematic and well-defined rules, which are called *protocols*. Internal system information or envi-

ronmental information is transmitted via *state messages* or *data messages*.

A safe real-time communication has to comply with the following five essential behavioral aspects, valid for the development of distributed real-time systems [Kop97].

Bounded Protocol Latency Information transmission is characterized by protocol latency, which is the interval between the start of message transmission on the communication network interface (CNI) on sending node A and the reception of that message on receiving node B.

Supporting Composability A composable communication allows expanding the communication range to other nodes without any further problems of recalibrating existing configurations or implementations. The variety of existing types for real-time systems is another requirement for communication.

Flexibility Flexibility in this context means that different types of systems with different configurations and properties are adapted to the existing real-time network without any major changes to the system.

Error detection Dependable and predictable services should determine the probabilistic presence of communication errors. Those services also determine a correction of erroneous transmission or inform all nodes participating in the network about a hazardous situation. Communication errors are also caused by operations taking place on the node itself (for example, malicious interactions with actuators). Communication has to take care about the end-to-end acknowledgment of control messages and its reaction, which implies a separate monitoring mechanism for such systems.

Physical structure The physical structure is decisive for the requested behavior of the distributed real-time system. In most cases, decisions depend on economic and technical reasons limited by the stakeholders.

2.2.3 Time-Triggered Architecture

Developed over the last two decades at the Vienna University of Technology[1], the *Time-Triggered Architecture (TTA)* provides a com-

[1]Vienna University of Technology, Institute of Computer Engineering, Real-Time Systems Group (http://www.vmars.tuwien.ac.at)

prehensive design statement to industrial demands listed in Section 2.2.2 on page 8 before.

The TTA specifies a computing framework designated for large distributed real-time systems. The entire system intended is divided into nearly autonomous components. The components are clusters or nodes. Every node has a global, fault-tolerant time base of a priori known precision. Communication and agreement protocols are simplified by this global time, which allows a transparent specification of interfaces among nodes. A global time base also enables a prompt error detection and guarantees the timeliness of a real-time application. The design of time-triggered systems is based on the so-called two-level design approach. In the architecture design phase, interaction among the components is specified by defining interfaces in the temporal and value domain. In the component design phase, components are built according to the constraints specified in the preceding architecture design phase [KB03].

The TTA solves communication issues by using the *time-triggered protocol (TTP)*. TTP is an integrated communication protocol enabling fault-tolerant communication among distributed real-time systems. Based on a set of specific services, TTP contributes to a safe behavior of the distributed system. Those service are *"predictable message transmission, message acknowledgment in group communication, clock synchronization, membership, rapid mode changes, and redundancy management[KG94]."* Two classes of protocol types have been derived from TTP. *TTP SAE Class A (TTP/A)* is a low-cost field-bus protocol, whereas *TTP SAE Class C (TTP/C)* is intended as a fault-tolerant communication solution in the safety-critical domain (for example, aerospace industry) [Sch01] [PK98].

2.3 Safety-Critical Aspects

When we specify safety requirements for a computer-based, distributed real-time system, the fundamental principles to be linked with the concept of safety have a deep impact on an appropriate and sufficient safety-related design and the implementation corresponding to that design.

The *safety-critical aspects* presented in the following sections are essential concepts that are frequently used in this domain when elaborating approaches to software requirements for safety-critical systems, in particular when developing software for distributed, time-triggered real-time systems.

2.3.1 Dependability

The safety of a software system property is integrated into the concept of *dependability*.

Dependability itself is the capability of a computer-based system to *"deliver service that can justifiably be trusted. The service delivered by a system is its behavior as it is perceived by its user(s); a user is another system (physical, human) that interacts with the former at the service interface. The function of a system is what a system is intended to do, and is described by the functional specification. Correct service is delivered when the service implements the system function. A system failure is an event that occurs when the delivered service deviates from correct service. A failure is thus a transition from correct service to incorrect service, i.e., to not implementing the system function. The delivery of incorrect service is a system outage. A transition from incorrect service to correct service is service restoration [ALR01]."*

Software requirements are the integral part of any functional specification. Thus, the lexical and semantic content of a software requirement contributes to the dependability of a computer-based system. Apart from the impact functional specifications can have on safety-critical systems the notion of time constitutes the conceptual basis for a failure.

Dependability can be located within three categories – *impairments* to dependability, *means* for dependability and *attributes* of dependability –, from which a number of attributes extend. Figure 2.2 on the next page shows these categories in a dependability tree [ALR01].

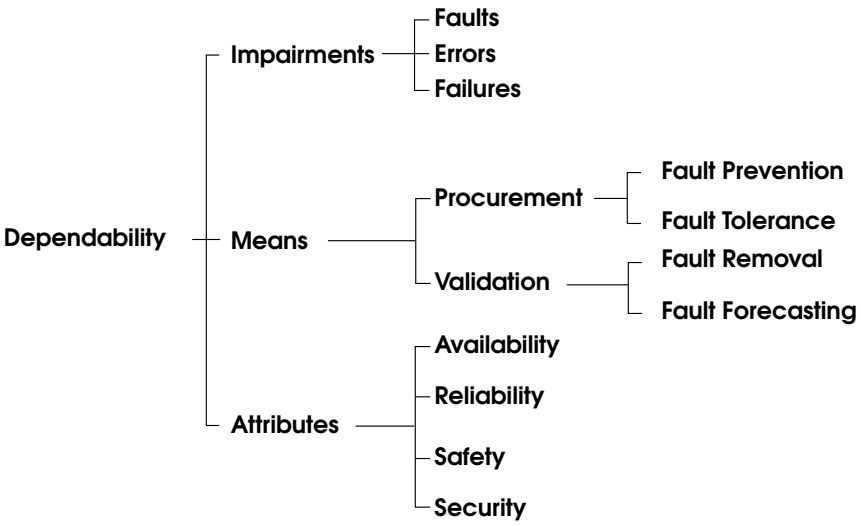


Figure 2.2: The dependability tree [ALR01]

The structural background of the first category, *impairments* to dependability (see Figure 2.2 on this page), is based on the attributes *error*, *fault* and *failure*. Whenever a sequential time-line is followed, a fault constitutes the adjudged or hypothesized cause of an error. An error is an unintended state within the behavior of a computer-based system. Based on an error, the resulting deviation of actual service from intended service is the corresponding failure [Kop97].

Safety is considered as one of six basic attributes a dependable system has to comprise. Dependability requirements describe to which extent and with which effort a dependability goal, as defined by a specific dependability attribute, has to be achieved within a system. In [ALR01] the authors extend their concept of dependability by so called *secondary* attributes. In the context of software requirements, the secondary dependability attribute *robustness* plays a major role in specifying dependable real-time software systems. Robustness deals with dependability in the presence of unavoidable external faults which denote a specific class of faults with its own classification of system reaction. Strictly satisfying only the basic set of dependability attributes is not a deterministic approach to a dependable system, because different kinds of dependable systems imply a variety and/or combination of dependability attributes in order to satisfy a specific safety-critical user requirement.

The third dependability category, *means* of dependability, relates to methods and techniques enabling the establishing of dependable computer-based systems. Two attributes extending these means,

fault prevention and *fault tolerance*, can be seen as dependability *procurement* constructing a dependable system, whereas the attributes *fault removal* and *fault forecasting* can be considered as dependability *validation* ensuring a valid employment of procurement methods [Hil98].

Fault management deals with the adequate employment of these four means during the development of dependable systems. A description of these means and the proper techniques used to implement them is given below.

Fault prevention: Fault prevention is aimed at preventing the introduction or occurrence of a fault within the behavioral sphere of a system, which is done by quality control during the software design and software implementation. Techniques used to implement this means of dependability are modularization, structured programming and abstraction techniques supported by systematic approaches to development and training.

Fault tolerance To preserve the intended behavior of a system even in the presence of an active fault, fault tolerance is carried out mainly by error detection and system recovery. Whenever the system reaches an erroneous state, error detection within the system uncovers the situation by assigning error signals to the corresponding unintended state (for example, "exception handling" for software). The effectiveness of the error treatment methods is crucial to the valid and correct execution of any fault-tolerant system. In a next step, the system recovery returns the erroneous state into a valid and correct execution state mode by using error and fault handling. Examples for software fault tolerance techniques are *N-Version Programming*, *Recovery Blocks* or *Diversity Programming* [Hil98].

Fault removal: Fault removal techniques must be applied to system development or the operational life of a system to reduce the presence (number, seriousness) of faults. *Validation* and *verification* (see Section 2.4 on page 17 for a more detailed description) are applied to the development process to uncover faults in the software design specification and subsequent software implementation. As far as the operational life of a system is concerned, faults can be reduced by *corrective* and *preventive* maintenance.

Fault forecasting: Fault forecasting estimates the future occurrence and consequences of faults and determines a fault model for the software system by using qualitative (ordinal) or quantitative (probabilistic) evaluation. In the course of modeling software requirements, which are stated in functional system specifications, techniques and methods are used to prevent the system from hazardous states. Such software hazards may contribute to an error (or accident) occurring within the system. Examples of hazard treatment techniques are *Fault Tree Analysis (FTA)* or *Hazard and Operability Analysis (HAZOP)* (see Section 4.3 on page 45 for a more detailed description).

2.3.2 System and Software Safety

The term *safety* for computer-based, safety-critical systems has a large number of definitions in the scientific and industrial literature. The term *safety* can be defined in the following way:

> *"Safety is a property of the system that will not endanger human life or the environment [Sto96]."*

Here is a more precise definition:

> *"The conservation of human life and its effectiveness, and the prevention of damage to items as per mission requirements [B.S03]."*

The operational field of real-time systems offers adequate functionality in order to cope with the challenge of providing safe system behavior to the environment. Real-time systems constitute a co-design of software and hardware components and cover, owing to their wide operational field in different industrial domains, a broad range of safety-critical applications.

This conceptual point of view makes it necessary to distinguish between *system* and *software* when talking about safety.

System Safety

In this work we assume a safety-critical system as a computer-based system comprising hardware and software components. When developing a safety-critical system it should be made sure to prevent the risk of an unforeseen accident that might be based

on the occurrence of a hazard or to reduced such a risk to a tolerable level. Therefore it is necessary to identify and analyze hazards at system level. If a system hazard is identified, it must be determined how severe or critical are the consequences of that hazard and how likely it is to occur. After the design decision has been made, potential hazard sources are assigned to the specific hardware or software components. Unfortunately hazards have the nature to appear in every stage of the development process. Thus system safety activities have to be done at every stage of a system life-cycle in order to handle the unavoidable presence of system hazards in an appropriate and safe way. A system safety engineer has to manage hazards by using analysis, control and management procedures to identify, evaluate, eliminate and control those hazards [Lev03].

Based on the following seven principles, this safety approach to creating computer-based systems offers a systematic way for preventing foreseeable accidents [Lev95] :

"System safety . . . "

- *". . . emphasizes building in safety, not adding it on to a completed design."*
- *". . . deals with systems as a whole rather than with subsystems or components."*
- *". . . takes a larger view of hazards than just failures."*
- *". . . emphasizes analysis rather than past experience and standards."*
- *". . . emphasizes qualitative rather than quantitative approaches."*
- *". . . recognizes the importance of trade-offs and conflicts in system design."*
- *". . . is more than just system engineering."*

Whenever a hazard is identified, the system design or environment have to be modified to provide a way to eliminate this potentially malicious system state. If such a system design or environment modification cannot avoid an unacceptable system hazard, system safety requirements have to be specified.

Software Safety

> *"A safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state [Lev95]."*

In other words, the relationship between software and system [2] is based on the assignment of specific system hazard aspects to the corresponding software parts. For this reason, software safety is a system safety problem. *Software safety*, which is also referred to as *software system safety* to emphasize the system nature of this specific issue, is considered as *"the freedom from software-related hazards [B.S03]."* A *software hazard* is a software condition that is a malicious prerequisite for an accident of the system itself. The embedded behavior of software within the system context, but also the hazard characteristics of software itself can affect system safety in two different ways [Lev95] :

- software output values or timing can contribute to a hazardous situation,
- software is not able to control or handle hardware failures which propagate through the system design in order to transfer the system to a hazardous state.

The complexity and diversity in software development entails difficulties when it comes to determine qualitative statements about the dependability of a software component. Whereas in hardware development a component failure rate is a significant state about the systems reliability, in software no according means can be applied and therefore does not constitute an effective means for software risk assessment within the domain of safety-critical (software) systems [Lev91]. However, different approaches employed to achieve software reliability can show that software is not *safe*. For this reason, software safety has to focus on the safe creation and implementation of software design issues preventing hazardous situations from occurring or reducing the risk of hazard occurrence.

To elaborate safety requirements for software it is necessary to have an elementary understanding of how software can cause hazards. Table 2.1 on the next page gives an overview of the potential causes leading to the occurrence of software hazards [B.S03].

The entire system development process uses the *preliminary hazard analysis (PHA)* at a late stage of requirements engineering and

[2] In this context a system is an abbreviation for safety-critical system.

A failure in recognizing a hazardous situation requiring a corrective measure
A failure to perform a required function
Poor response to a contingency
Incorrect solution to a problem
Performing a function which is not required
Performing a function which is out of sequence
Poor timing of response for an adverse situation

Table 2.1: Potential Causes for Software Hazards [B.S03]

at an early phase of the design process, that is, comparably early in the entire life-cycle. The PHA is aimed at identifying safety-critical domains, doing a first assessment of hazards, and defining an adequate hazard control and appropriate additional activities. Hazards identified in the course of a PHA are assigned to specific software components, after a design decision has been made. A *subsystem hazard analysis (SSHA)* carried out in connection with a PHA is aimed at identifying the potential causes for software hazards listed in Table 2.1 on this page. Those hazards can occur at the interface level of software components, between software components, or within the software design itself. Software safety, furthermore, is aimed at finding out how software components have contributed to a hazardous situation, which is done by

1. identifying software hazards,
2. assessing an appropriate risk level for the potential occurrence of a software hazard and
3. ensuring a sufficient effort to design devices that eliminate or control the hazards identified [Lev91].

Software safety standard documents describe appropriate techniques and measures used to conduct a software development process in a proper way (see Section 3.2.2 on page 27 for details).

2.4 Software Validation and Verification

Every project starts with the specification of requirements developed by stakeholders involved in order to provide a transparent and consistent view of the intended system. Requirement specifications constitute a legal and commercial basis on the basis of which the

system is developed. The system development process, however, has to integrate several phases phases providing a continual statement about the correct implementation of the intended system behavior.

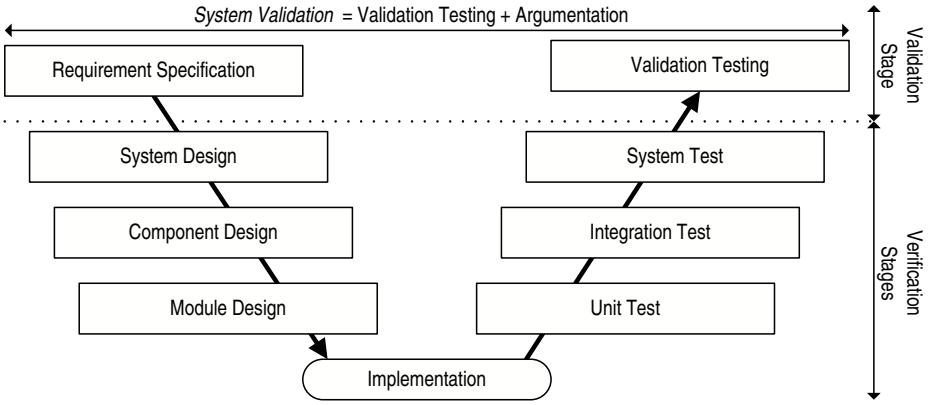


Figure 2.3: The V-Model

Figure 2.3 shows the *V-Model*, which is a development process model contrasting every development stage displayed on the left side of the figure with the specific *verification* and *validation* stage displayed on the right side. Thus, verification or validation, in every stage of system development, provides an up-to-date statement about the implementation of the respective system behavior.

2.4.1 Verification, Validation and Argumentation

Definition 2.1 (Verification) *"Confirmation by examination and provisions of objective evidence that the particular requirements for a* ***specific intended use*** *are fulfilled [IEE98]."*

Development activities, which relate to the verification stages, have to determine measures and techniques of how to construct a system architecture that provides the intended system behavior. The requirement specification itself is a means to decide how to design and implement components and their relationship between each other. Thus, design and implementation have a definition of their own particular requirements. A summary of these particular requirements specified for each system component defines the entire system behavior. Simultaneously the *specific intended use* is defined by implementing a specific solution to the existing problem.

Verification methods deal with the correct development of the design and the implementation corresponding to that design. *Testing* offers a comprehensive and broad, diversified verification activity. Different kinds of testing techniques and measures used to design and implement a software system provide meaningful statements about the correctness of that system. Classifications of testing techniques and measures reflect different perceptions of the system and its development.

In [Rus91] testing activities are grouped into *dynamic testing*, *static testing* and the *testing of specifications and requirements*.

Dynamic Testing monitors the system and provides information about whether the system is executed correctly or not. Dynamic test case generation covers a broad range of input data for the system in order to prevent unexpected behavior. Examples for dynamic testing are *random testing* or *border value testing*.

Static Testing reveals inconsistencies and omissions within the source code implementation. Static testing also proves the correct transformation of software requirements into source code. Static testing examples are *structured walk-throughs* or *anomaly detection*.

Testing of requirements and specifications proves the correct and valid specification of the system behavior at a very early development stage.

In this context, testing procedures depend on how information about the intended system is specified in the documentation. If a specific formalism is provided, adequate formal techniques and methods can be applied to detect anomalies within the requirement model or design specification. Formal methods are described in Section 2.4.2 on the following page.

Definition 2.2 (Validation) *"Confirmation by examination and provisions of objective evidence that* ***specified requirements*** *have been fulfilled [IEE98]."*

Based on testing activities, a verification statement informs about the correct design or implementation of the system. When requirements are validated, a summary of verification statements provided by different development stages comprehensively evidences the correct and safe system behavior. The summary of verification activities is therefore called *validation testing*.

In the case of safety-critical systems, where requirements constitute specified statements about system safety, *argumentation* arranges information coming from validation testing. Argumentation is aimed at providing evidence of sufficient safety being implemented in the system.

For this reason, a correct validation of a safety-critical software product consists of validation testing done in combination with argumentation.

2.4.2 Formal Methods

Using *natural language (NL)* for the specification of software requirements often leaves room for interpretation, as far as the subsequent development stages are concerned. Ambiguity or inconsistency present in the requirements document may lead to malicious misinterpretation of the specified requirements. Thus NL may constitute a potential source of hazard occurrence.

As described in Section 2.4.1 on page 18, the use of a precise formalism in the requirements specification process supports argumentation about system safety. Formal methods use discrete mathematics to describe the discrete behavior of a system.

"The word 'formal' in formal methods derives from formal logic and means 'to do with form'. The idea in formal logic is to avoid reliance on human intuition and judgment in evaluating arguments by requiring that all assumptions and all reasoning steps be made explicit, and further requiring that each reasoning step be an instance of a very small number of allowed rules of inference. Assumptions, theorems, and proofs are written in a restricted language with very precise rules about what constitutes an acceptable statement or a valid proof. In their pure, mathematical form, these languages and their associated rules of manipulation are called logic. In formal methods for computer science, the languages are enriched with some of the ideas from programming languages and are called specification languages, but their underlying interpretation is usually based on a standard logic[3]*".*

In terms of requirements specifications, formal specification languages provide a mathematical model description of the intended system behavior. Requirements are additionally transformed into formal properties of the system model. A proof using a requirement as a formal property on the formal model validates the correct im-

[3]Nowadays, *temporal logic* is used for *standard logic* [Rus93].

plementation of the system behavior.

Modeling and proving separates formal methods into *formal verification* and *formal specification*.

"Formal Specification is the use of notations derived from formal logic to define (1) the requirements that the system is to achieve, (2) a design to accomplish those requirements, and (3) the assumptions about the world in which a system will operate. The requirements explicitly define the functionality required from the system as well as enumerating any specific behaviors that the system must meet, such as safety properties [...] Formal Verification is the use of proof methods from formal logic to (1) analyze a specification for certain forms of consistency and completeness, (2) prove that the design will satisfy the requirements, given the assumptions, and (3) prove that a more detailed design implements a more abstract one [Spi00]."

Despite the cost-effective benefits of validation done in the early stages of the development process, a possible introduction of formal methods into the development process encounters resistance, which is partly based on myths described in [Hal90] and [BH95].

Examples of the use of formal methods in industry are the formal specification notation Z or *model checking* and *theorem proving* for the verification of hardware design logic [CW96].

Chapter 3

Related Work

A reuse approach is basically aimed at creating an information structure which then is classified in order to provide specific reusable system artifacts. This chapter outlines related work done in the field of software requirements reuse and proceeds with classification schemes established for the safety-critical domain.

3.1 Requirements Reuse

As software used in real-time systems is playing a more and more independent role, the need for software reuse even in the domain of safety-critical systems (for example, in the automotive industry) [HKK04] increases. Reusing software offers a cost-effective strategy in improving quality and increasing productivity for the intended system. In the software domain, the idea of patterns has lead to effective re-usability artifacts [GHJV02] [App97]. Reuse in general is characterized by four dimensions: *abstraction*, *selection*, *specialization* and *integration* [Kru92]. This thesis focuses on potential reuse sources for the requirements capture phase.

"Knowledge-based requirements representation techniques are frequently matched with intelligent identification of and search for software requirements [Cyb98]."

3.1.1 Domain Analysis

If system development is forced to prepare the system design due to a modification of requirements, existing domain-specific knowledge minimizes the effort connected with that task. In software engineering, domain analysis deals with the extraction of generic informa-

tion structures suitable for a specific domain. In this context, the term *domain* refers to an *application area*.

Furthermore, *"domain analysis can be seen as process where information used in developing systems is identified, captured, structured and organized for further reuse. More specifically, domain analysis deals with the development and evolution of an information structure to support reuse [PD90]."*

In order to establish such information structures, the authors of [PD90] introduce a domain analysis process called *structured analysis and design technique (SADT)*. The process model is used to conduct a conditional development process. The SADT uses domain-related inputs, outputs, controls and mechanisms to determine a *domain model* for every stage of the development process. In addition to domain models, the SADT elaborates standard documents and reusable components.

The *feature-oriented domain analysis (FODA)* [KCH+90] is one of the most well-known domain analysis methods. FODA contributes to the model-based approach and focuses on the *features* of similar software applications. From a user perspective, a feature is a capability of the system application. FODA is aimed at creating a domain model that covers common and variable features of related software systems. Thus, a set of features is taken as a description for the domain in which the system is located. A feature itself can be *mandatory*, *optional*, or *alternative*. These three attributes are used for defining the domain. Requirements engineering disciplines participate in the creation of a domain model corresponding to those attributes.

The *JIAWG Object-Oriented Domain Analysis (JODA)*[1] method combines domain analysis issues with the concept of *object-oriented analysis* [CY91]. JODA basically divides the domain analysis process into three stages:

1. **Domain preparation:** The object-oriented paradigm identifies objects, attributes and services, and their relationship.
2. **Domain definition:** The definition of the domain provides scenarios that are defined and simulated.
3. **Domain modeling:** Finally, the resulting objects are abstracted and grouped together for their potential reuse.

The behavior classification described in this thesis is oriented toward the concept of domain analysis. Apart from the identification

[1] "JIAWG" stands for "Joint Integrated Avionics Working Group".

of commonalities, the thesis focuses on specifying a precise separation of system behavior. The methods mentioned above deal with a potential reuse of requirements, but do not cover any issue with regard to their semantic behavioral information.

3.1.2 AC 20-148 Reusable Software Components (RSC)

Safety-critical software systems are certified in combination with the hardware underlying those systems in order to provide system safety. Changing the design of the system software or its hardware units results in an unavoidable re-certification of the entire system. Thus, a software upgrade entails cost-effective certification efforts on the contractor's side. On the other hand, certification authorities are faced with an administrative certification overload based on such minor changes (for example, on software systems for different CPU models of a controller family). For this reason, the *AC 20-148 (AC)*[2] [Fed04] provides guidelines supporting engineers in receiving FAA acceptance for a reusable software component (RSC) The AC leaves room for system-specific interpretations. It does not provide detailed information on how to establish a reusable set of specified requirements. In terms of the AC, a system intending to use an RSC has to meet all applicable RTCA/DO-178B objectives. Thus any reusable software component has to be developed following the guidelines in RTCA/DO-178B. In addition to considerations of AC the component is reusable within the certification domain. Section 3.2.2 on page 29 gives a detailed description of RTCA/DO-178B. The AC guidelines provide information on the development and use of RSC. Furthermore the AC documents how to handle a design change of the RSC. The AC also addresses life-cycle data of the RSC across company or division boundaries.

A commercial example for considerations laid down in the AC is *LynxOS®*, a real-time operating system that complies with all RTCA/DO-178B guidelines. Being an active operating software platform, it constitutes an RSC[3].

3.1.3 Natural Language Processing

In the *Advanced Integrated Requirement Engineering System (AIRES)* [4] project, requirements are processed as natural language texts.

[2] *AC* stands for *Advisory Circular*.
[3] http://www.lynx.com/
[4] http://ite.gmu.edu/~cset/aires.htm

In [Par93] the author introduces a technique that clusters requirement sentences specified in natural language. The motivation behind is to reduce the effort-intense specification process for software requirements and to prevent a situation where wrong requirements implicate wrong systems. The focus of the reuse concept is the requirements text specified in natural language. For this reason, the concept discussed in [Par93] describes a two-phase process aimed at developing reusable *requirement components*. In this context, requirement components are objects, functions and quality goals with regard to the system. In the first phase, the *component extraction phase*, a text analysis and extraction process is used to identify the requirement components. The text analysis is based on a *lexical analysis* and *syntactic pattern analysis*, whereas the intrinsic extraction is based on requirement clustering due to component identification. In the next phase, the *component classification phase*, the components are classified with regard to their lexical affinities.

This thesis examines natural language requirements using the model-driven approach, as described in [MM03]. Requirements are also treated as whole sentences. The proposed behavior classification may profit from the concept of requirement components and lexical-related identification techniques.

3.1.4 Real-Time Requirement Pattern

The use of formal methods in the field of requirements engineering has been mentioned in Section 2.4.2 on page 20. The authors of [PGK97] introduce a generic approach to the specification of system requirements. This specification is based on a set of requirement patterns aimed at reducing the formal system requirements specification. A requirement pattern relates to design patterns used in the field of software engineering [GHJV02]. In [GKP98] they extend this approach by using a *real-time requirement pattern* which allows transforming natural language requirements for real-time systems into a precise and concise formal specification, with a tailored, temporal real-time logic being used for specification. A class of domain-specific software requirements are specified on the basis of the requirement pattern concept. The description of a requirement pattern follows a *requirements pattern description template*. Starting with a pool of requirements patterns, the reuse of requirements undergoes a three-step process. First, one requirement pattern is *selected* from the *requirements pool* and evolves in the course of time. Secondly, the patterns are *adapted* to specific, suitable instantiations. Finally a requirements specification is

developed by *composing* these specific patterns together. Unfortunately, if there is no pattern available, a time-consuming process in creating such a pattern has to be started during development. In [PGK97] and [GKP98], the authors in each case "discover" a requirement reuse pattern for real-time systems.

3.2 Classification Schemes

Classification schemes can reflect different views of the nature of a requirement in the safety-critical context. Safety integrity levels make the effort to be required for the development of a *safe* system measurable. Safety standards offer a requirement mechanism to achieve the intended safe-system behavior during development. A corresponding implementation profits from guidelines reflecting the requirements for man-machine interaction.

3.2.1 Safety Integrity Levels

Safety Integrity Levels (SIL) divide the need for system safety into five categories, referred to as SIL0, SIL1, SIL2, SIL3, and SIL4. SIL 0 is mostly neglected because systems with that safety integrity level do not have any malicious impact on their environment. SIL 4 systems, by way of contrast, are characterized by the highest safety integrity level suiting the context on which the system operates. The principle behind the SIL concept is that the risk of hazardous malfunctions in a *safe* system decreases, as the effort expended to develop it increases.

For this reason, the effort expended affects each stage of a system's life-cycle. Each stage results in a varying amount of requirements for the development and design of the system. Thus each SIL requires certain techniques and measures to provide evidence of safety, as specified by the SIL-related requirements. The SIL concept furthermore assumes that fatal and hazardous events resulting from system malfunctions derive from design errors rather than from an unexpected input value domain.

Table 3.1 on page 28 gives an overview of the five different SILs, describes the significance of the individual categories for the development of safety-critical systems and assigns *probability values of a dangerous failure per hour (PFH)*[5]. Table 3.1 on page 28 moreover

[5]PFH of a safety-related system encompasses a high demand or continuous mode of operation following [Int98a].

explains the role of the system with respect to its impact on the environment when the SIL increases.

Developing safe software as a part of a safety-critical system considers human activity in the specification, design and implementation stage as key issue when following the SIL principle. An adequate set of measures and techniques employed to achieve and validate a specifically required SIL is introduced into the life-cycle of software system.

3.2.2 Safety Standards

Characterizing safety by five SILs (see Section 3.2.1 on the preceding page) implies a set of requirements for the intended "safe" system. Safety standards classify those requirements, propose requirement development strategies and offer a specific approach to satisfy the need for safety evidence. IEC 61508 and RTCA/DO-178B have different interpretations of SILs and the treatment of requirements in the system development.

IEC 61508 – Functional Safety of E/E/PE Safety-Related Systems

IEC 61508 is a stand-alone standard for safety-related systems. This standard document can be adapted to different kinds of safety-related industrial domains (for example, CENELEC[6] for railway systems). IEC 61508 focuses on *functional safety*, which is described as *"part of the overall safety that depends on a system or equipment operation correctly in response to its outputs."* [Int02].

IEC 61508 covers safety-related developing issues for hardware and software by arrogating the satisfactory validation of so-called *safety requirements* for design, implementation, operation and maintenance. Unlike Section 4.1.1 on page 33, safety requirements in this context refer to system development. Thus they indirectly contribute to a safe behavior.[7] The IEC 61508 standard is divided into seven parts. For the development of safety-related software, IEC 61508-3 [Int98b] – in addition to IEC 61508-1 [Int98a] – defines a framework of safety requirements for software developing. It breaks down the entire system life-cycle into a specific safety software life cycle [Int98a].

[6] http://www.cenelec.org

[7] In IEC 61508, the safety function requirement equals to our safety requirement, described in Section 4.1.1 on page 33.

SIL	Description of SIL [HR99]	PFH
SIL 0	The system does not affect safety. No assumptions are made about its safety and the corresponding safety requirements.	*none*
SIL 1	The system was developed for normal commercial use with regard to quality and technology. SIL 1 systems are not expected to be the direct cause of fatal accidents, but they could be a minor contributing factor.	$\geq 10^{-6}$ to $< 10^{-5}$
SIL 2	High quality commercial systems. With careful system design, very good-quality COTS products should be able to be justified for use in SIL 2 systems. SIL 2 systems are not expected to be the sole cause of a fatal accident but they could provide a significant contribution.	$\geq 10^{-7}$ to $< 10^{-6}$
SIL 3	The system was specifically developed to meet safety requirements. The development should use specialist methods and techniques and the highest levels of quality assurance, modeling, analysis and testing. COTS products are most unlikely to be able to be used for the implementation of SIL 3 functions. Although a failure of a SIL 3 system should not be the sole cause of a fatal accident, it may create a situation that challenges the safety protection systems.	$\geq 10^{-8}$ to $< 10^{-7}$
SIL 4	The highest safety level of a system of proven quality utilizing multiple hardware redundancy. Software that is able to meet this level of integrity demands the highest possible project competence, the best available technology, the closest attention to the correctness of the requirements, design and code, and analysis of the code to demonstrate that it matches the requirements, in addition to all other elements of good practice. SIL 4 systems are considered as *safety-critical*.	$\geq 10^{-9}$ to $< 10^{-8}$

Table 3.1: Safety Integrity Levels

A specific SIL is assigned to every software system. To show compliance with a specific SIL, safety requirements, as defined in the IEC 61508 safety standard, are specified by a SIL-related classification. Such classifications propose (highly) recommended validation and verification techniques for every safety requirement stated. References to IEC 61508-7 citeiec61508:7 provide significant information on how to validate all the safety requirements proposed.

According to IEC 61508, a safety requirement is based on a *safety function requirement* and a *safety integrity requirement*. Safety function requirements describe the intended functionality and behavior of the system, whereas safety integrity requirements ensure that the system reflects a certain likelihood of the derived safety function to perform correctly. The safety function, however, is only implemented when safety is needed. Thus the entire system is divided into a functional and safety-related domain. When developing safe software, the safety-related domain usually is minimized because of the costs involved. This assumptions refer to system requirements demanding SIL 3.

RTCA/DO-178B – Software Considerations in Airborne Systems and Equipment Certification

The integration of safety into software is an essential requirement for systems developed for the aerospace domain. For a conventional aircraft, a system failure may have a catastrophic impact on the environment and on human lives. RTCA/DO-178B [Rad92] has been established by the *Radio Technical Commission for Aeronautics (RTCA) Special Committee 167* to provide strict guidelines for the development and certification of safety-critical avionics software. Unlike *safety-related* systems based on IEC 61508 (see Section 3.2.2 on page 27, RTCA/DO-178B focuses on *safety-critical* systems corresponding to IEC 61508 SIL 4.

In RTCA/DO-178B, the safety integrity levels finds their interpretation in assigning *failure conditions* to system functions. On the basis of these failure conditions, the integration of safety into software development and certification follows a separation into five different software *design assurance levels (DAL)*.

Table 3.2 on the following page compares design assurance levels with the IEC 61508 SIL concept. Both concepts are derived from the basic SIL description given in Section 3.2.1 on page 26. RTCA/DO-178B corresponds to IEC 61508-3[Int98b], because both address only software development issues. RTCA/DO-

RTCA/DO-178B	IEC 61508
Catastrophic (Level A)	SIL 4
Hazardous/Severe-Major (Level B)	SIL 3
Major (Level C)	SIL 2
Minor (Level D)	SIL 1
No Effect (Level E)	SIL 0

Table 3.2: Comparison of DALs and SIL

178B recommends a two-level development approach (the so-called *what?/how?* principle) for specifying a safe software behavior. Requirements for hardware are specified in RTCA/DO-154 [Rad00], which addresses hardware-related issues. The entire system safety concept, which corresponds to IEC 61508-1 [Int98a], is determined by ARP 4754 [Soc96a] and ARP 4761 [Soc96b].

The elicitation and specification of safety requirements for safety-critical systems within the software life-cycle itself is not described in RTCA/DO-178B. The *System Safety Assessment Process (SSAP)*, stated in ARP 4754 [Soc96a] and, in a more detailed way, in ARP 4761 [Soc96b], comprises three phases.

1. The *Failure Hazard Analysis (FHA)* specifies the safety requirements the system has to be meet and answers the question of how much safety is required to achieve a tolerable level of risk. The result of the FHA are *Safety Objects*
2. In the next phase, the *Preliminary System Assessment (PSA)*, these *safety Objects* are allocated to the safety requirements the system has to meet.
3. Finally, the *System Safety Assessment (SSA)* validates the specified safety requirements through software implementation.

The verification and validation of software requirements based on RTCA/DO-178B is ensured by *structural coverage* and *requirement coverage*. In literature, structural coverage is also referred as *code coverage*. RTCA/DO-178B recommends *Modified Condition/Decision Coverage (MC/DC)* as structural coverage [HV01]. In order to do a satisfactory test coverage, it is necessary to define test cases corresponding to the requirements. For traceability reasons, these *High -Level* and *Low-Level Test Cases* are directly derived from software requirements. A boundary value analysis adds specific *Robustness Test Cases (RTC)* to the existing low-level test cases,

which significantly extends the range of the existing test coverage. Requirement coverage adopts the results of the code coverage. Furthermore, requirement coverage provides information on software requirements validation. According to RTCA/DO-178B, the design assurance level A requires a one-hundred-percent code coverage.

3.2.3 Implementing Safe Software with MISRA C

The classification of requirements can also address a safer implementation of software systems. The *Motor Industry Software Reliability Association (MISRA) - Guidelines For The Use Of the C Language in Vehicle Based Software* [The98] constitutes such a contribution. In MISRA, a requirement is interpreted as a *Rule*. Rules determine the use of the C programming language for any kind of system implementation aspect. The motivation behind MISRA is the mitigation of programming language insecurities (for example, mistakes, misunderstandings of the programming language, or unexpected output based on erroneous compilers). Thus the MISRA guidelines focus on a safe interaction of human intention with the resulting system implementation.

The scope of the MISRA rules determines a subset of the C programming language, which is considered as safe for concrete implementation aspects. Apart from that, MISRA involves the concept of SIL, enforcing its adaption starting when SIL 2 or 3 is required. Software metrics and style issues are subjective items in every software development process and thus not in the sphere of MISRA. The MISRA standard document does not recommend any specific tools or vendors.

1	Environment	10	Conversions
2	Character Sets	11	Expressions
3	Comments	12	Control Flow
4	Identifiers	13	Functions
5	Types	14	Preprocessing Directives
6	Constants	15	Pointers and Arrays
7	Declarations and Definitions	16	Structures and Unions
8	Initialization	17	Standard Libraries
9	Operators		

Table 3.3: MISRA Rules

In Table 3.3 on the preceding page MISRA issues the corresponding classifications of proposed rules for a *safe* implementation using the C programming language. Not every rule[8] listed in Table 3.3 on the preceding page has to be applied to the implementation process. MISRA distinguishes between *required* and *advisory* rules. Whereas a required rule is mandatory, a programmer should follow advisory rules when using the MISRA C subset for implementation.

[8]MISRA presents 127 rules illustrated within the 17 classifications listed in Table 3.3 on the previous page.

Chapter 4

Requirements and Safety

It is essential to understand the context of safety and requirements for a system development process. This process is targeted on safety by elaborating the system and software requirements a system has to meet. A safety-related system also undergoes a hazard analysis. The systematic use of hazard analysis techniques allows the assignment of safety requirements to unexpected but identifiable hazards that might occur in the system to be developed.

4.1 Safety Requirements Engineering

Requirements Engineering is a discipline that deals with the understanding and documenting of software requirements. Safety requirements constitute a solution to a specific safety-critical problem. Once the problem is defined, it is possible to elaborate requirements and determine a set of safety requirements stated in a requirements specification.

4.1.1 Safety Problem, Hazards and Requirements

A system development process addresses issues that have to do with the *problem domain* and *solution domain*. The development of requirements creates a conception of how to bring these two domains together.

Definition 4.1 (Problem Domain) *"The problem domain is the part of the world where the computer is to produce desired effects, together with the means available to produce them, directly or indirectly [Kov98]."*

With reference to Definition 4.1 on the previous page, the safety-critical domain constitutes the problem domain. In the problem domain, a system has to deal with the presence of hazards, which may lead to accidents and, therefore, to loss of human life. The specifically implemented system itself, however, resides in the so called *solution domain*. In the case of computer-based systems, the solution domain is also called *machine domain*.

Definition 4.2 (Well-Defined Problem) *"A well-defined problem is a set of criteria according to which proposed solutions either definitely solve the problem or definitely fail to solve it, along with any ancillary information, such as which materials are available to solve the problem [Kov98]."*

Note: Throughout this thesis, the term *well-defined problem* (see Definition 4.2 on this page) is briefly referred to as *problem*.

A safety problem addresses a situation residing in the safety problem domain. The set of criteria specified by a safety problem comprise structured information about situations that could potentially cause harm to human life or to the system environment. Basically, we can restrict the safety problem domain to a temporal domain, where unexpected situations may reside within a time-line that starts with the systems execution until the final result, an accident.

Definition 4.3 (Hazard) *"A hazard is a situation in which there is actual or potential danger to people or to the environment."*

Obviously, if there is a way to prevent a hazard, as explained in Definition 4.3 on the current page, the system remains safe even in extraordinary, unexpected system situations. Software hazards are based on faults. Hardware faults are random, whereas software faults are systematic. Software faults focus on software design and errors in the specification of software artifacts.

Definition 4.4 (Accident) *"An accident is an unintended event or sequence of events that causes death injury, environmental or material damage [Lev86]."*

If a safety problem is defined, an information structure has to bridge the gap between the problem and solution domains. Requirements address issues of the problem domain in order to provide a solution. They manifest themselves as part of the solution domain,

but obtain their information about the intended system from the problem domain. Based on Definition 4.3 on the preceding page, a safety requirement refers to the safety problem domain and provides details about hazardous situations (or hazards) in the systems. Requirements are furthermore statements that identify the need for a safe software system.

Software Hazard: The function `set_speed` returns the value 120 after calculation. **Safety Requirement** If the function `set_speed` returns a value < 100, the exception handler *shall* set the error flag to 1. *Rationale:* On the basis of data provided by two wheel sensors, the system function `set_speed` calculates the current speed value of a transport vehicle. The possible speed values are defined by an integer and can range from $I = \{0 \text{ to } 100\}$. The speed value calculated is sent to a braking system, which can adjust the system to a value level of not more than 110. Otherwise the braking system will fail.

Table 4.1: Example of a Safety Requirement

The software hazard given in Table 4.1 on the current page can lead to accident if the value is transmitted to the motor unit of the vehicle. A software requirement provides a solution to this problem by setting the error flag to 1, which will automatically switch the system to a mechanical mechanism. The safety requirement therefore provides a solution to a hazardous situation. A potentially fault-tolerant implementation would reside in the solution domain. The above example also shows that a requirement-based system behavior determines whether a requirement resides in the safety problem domain or not.

4.1.2 Safety Requirement Development

During the development of a safety-critical system, the unexpected occurrence of a hazard is omnipresent in every state of the development process. Development stages following the requirement development stage can profit from a well-defined and safe behavior description of the system. Even during requirements development, the different stages with their different techniques and measures are aimed at specifying correct, consistent and complete requirements intended to be valid over the entire software lifecycle [ZG02] [HED93].

For this reason, it is necessary to assess potential sources of an unexpected occurrence of a hazard and manage such sources by using a set of specific processes and techniques. In this context, the term *safety requirement development* defines activities needed for the development of safety requirements based on hazard identification.

Traditionally, the development of requirements comprises five core activities [NE00], that is the

1. *Eliciting*,
2. *Analyzing and Modeling*,
3. *Specifying*,
4. *Validating* and
5. *Managing* of requirements.

Note: As this thesis does not focus on the management of requirements and describes requirement validation in Section 2.4 on page 17, we confine ourselves to describing the following requirement development stages :

- *Eliciting*, focusing on reviewing, wording and understanding of what the user needs. In the case of software, the user's needs are also available as system requirements. A user is a human or a system. Besides, the elicitation process includes constraints on the software to be developed.
- *Analyzing and Modeling*, taking the established needs and starts defining them as requirements.
- *Specifying*, creating a document that states and specifies the software requirements defined.

These three fundamental requirement stages are combined with activities that are used to determine a set of safety requirements for a safety-critical system. The development of safety requirements is determined by this combination. Basically, the time-line of a safety requirement process can be divided into two phases, according to the occurrence of a hazard. Activities are required *before* or *after* a hazard is present in the system.

"Having identified the hazard associated with a system it is useful to classify them by their severity and their nature. The severity

of a hazard is related to the consequences of any accident that might occur as a result of that hazard. The nature of a hazard has considerable impact on the manner in which it may be controlled. [...] The importance of a hazard is related to both its severity and its frequency of occurrence. These two factors are combined within the concept of risk [Sto96].”

Appropriate safety-related activities take place in the context of conventional requirements development (see Figure 4.1 on the current page). Figure 4.1 on this page shows the presence of a hazard that has occurred at a very early development stage[1].

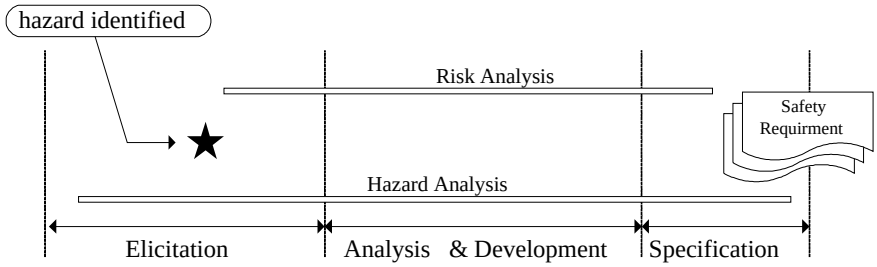


Figure 4.1: Safety Requirement Development

A *hazard analysis* is done even during the elicitation of requirements and aimed at identifying hazardous states in the software design. A *risk analysis* then uses information available about the hazards identified to determine – with reference to Definition 4.5 on the current page – the risk level for individuals or other systems.

Definition 4.5 (Risk) *“This is a measure of the probability and severity of a negative effect to environment, equipment/property, or the health [B.S03].”*

A risk analysis furthermore tries to propose a classification of hazards according to their *severity* and *frequency*, which reflects the quantitative probability of a specific risk. Qualitative measures are often used to describe the frequency or severity of a certain hazard where numerical values for risk classes are not appropriate. Quantitative and qualitative risk classes can be found in safety standards [Int98b] [Rad92].

If there is unacceptable risk for a design issue, *remedial design work* must be done to make the hazard less likely to occur or to

[1]The black asterisk stands for “hazard identified”.

mitigate the consequences. The results of such analysis are often referred to as *derived safety requirements* [McD02].

After hazards and risks have been identified, the safety requirement documentation ensures an interlock mechanism so that the system will be safe.

4.2 Requirements Capture

In the course of any system development, the requirements capture phase constitutes a key aspect in specifying the correct behavior of a system. As for safety-critical systems, requirements are split up into *system requirements*, *software requirements*, and *hardware requirements* for which additional safety requirements have to be specified if unavoidable hazards could occur.

4.2.1 System and Software Requirements

System Requirements

System Requirements specify the overall behavior of a system, which – in the case of real-time systems – constitutes a co-design between hardware and software. After the elaboration of system requirements, the respective specification allows taking specific *design decisions* on the ratio of components to be developed. An economic, legal and technical system requirements agreement made by the stakeholders involved is the basis of further software development activities.

In [Sof98] the number of stakeholders has been grouped into three main categories, with regard to system requirements:

Customer Usually the safety-critical domain favors product-driven systems instead of market-driven systems in which the user and his needs are well known. A user is a human or thing (for example, another subsystem). The system requirement specification is considered as a legal contract and interface between customer and developing company.

Raw requirements for the system are stated by the customer and describe the objectives, user needs and problems to be solved. Raw requirements describe the concrete idea of a safety-critical system and result in the determining of rough and superficial *system properties*.

Operational	***System***	***System Interface***
Properties, describing the general goals, objectives and desired capabilities of the system, but neglecting considerations on how to implement the system.	*Properties*, providing information about operations done within a system, about an approximate interface determination and about explicit external requirements for the system.	*Properties*, comprising a detailed description of external system interfaces.

Table 4.2: System Properties in terms of System Requirements

Technical Community Members of the technical community participate in every stage of the software life-cycle. Involving engineers in the development process of system requirements improves the quality of system requirements.

Environment In addition to the customer and analysts relationship, the environment of the system restricts the development of system requirements. Basically, safety standards and technological policies, with regard to industrial and governmental concerns, largely determine system requirements. The industrial background is to introduce similar structures, processes, techniques and measures in different industrial agencies, whereas governmental agencies have a demanding interest in ensuring that developing companies prove their software is as safe, as they claim it is. The IEC 61508 standard, for example, recommends test techniques and measures for every stage of the safety life-cycle of a specific IEC-based SIL compliance (see also Section 3.2.2 on page 27, IEC 61508). The *Federal Aviation Administration* (FAA) as an governmental agency commissions a Designated Engineering Representative (DER) to check the developing company's Plan for Software Aspects of Certification (PSAC) for its compliance with the Design Assurance Level (DAL)[2], as laid down in RTCA/DO-178B (for details on RTCA/DO-178B, refer to Section 3.2.2 on page 29, RTCA/DO-178B). In keeping a competitive and strong position, the real-time systems market is faced with a tremendous demand for safety.

[2]Corresponds to the term "Safety Integrity Level (SIL)", as defined by IEC.

The development of system requirements for safety-critical real-time system goes hand in hand with the creation of a corresponding system architecture. According to [KB03], a system architecture of a computer-based system is a framework for the construction of a system that constrains an implementation in such a way that the resulting system is understandable, maintainable, extensible, and can be developed in a cost-effective way. A component of a system architecture is either a subsystem or an elementary component (for example, a program, timer, sensor, or an actuator) and resides within the software or hardware of the system. The desired behavior of the system described by the system requirements specification has an influence on the selection of adequate components. As for dependable real-time systems, a system requirements specification helps to decide how communication should take place within the system. The relationship between the specification of system requirements and the creation of a corresponding system architecture traditionally is bidirectional. Whenever a system architecture does not completely suit the behavior description that is based on the system requirements, it is inevitable to modify the specification of the system requirements defined.

The development process for system requirements, with its bidirectional relationship to system architecture creation, deals with different types of system requirements. Apart from *functional* requirements, *safety* requirements and *additional certification* requirements there are well-established certification practices [Soc96a] [Soc96b] used for system development which focus on so-called *derived system requirements*. Section 4.2.2 on page 43 gives a detailed description of safety requirements for software. Airworthiness regulations usually require additional certification requirements, which have to be determined and agreed with airworthiness regulations for compliance with certification issues. *Derived* system requirements result from the design process for a safety-critical system itself. In the development process, design decisions based on derived system requirements show a new point of view of a system behavior not sufficiently covered by system requirements. Therefore derived system requirements can have implications, as they may not uniquely relate to a higher-level requirement stage. Derived system requirements also can result from system architecture considerations when the choice for a specific architecture or architectural component has its implications.

Software Requirements

When the system requirements have been sufficiently specified, engineers decide about how the ratio between software and hardware can be used within a safety-critical real-time system. This system co-design principle implies a close relationship between hardware and software architecture and, thus, characterizes the software requirements development process. Apart from requirements elaboration, the development process for software requirements is constrained by

- the *system requirements* and *architecture*,
- the underlying *hardware* with its requirements specification and architecture,
- *external considerations* based on the software engineering techniques used for development,
- the corresponding *software architecture*.

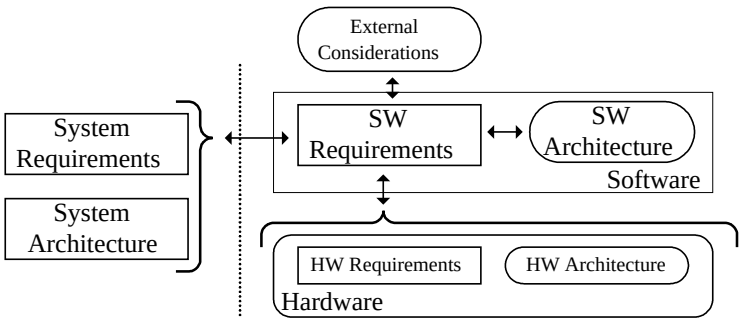


Figure 4.2: Constraints on Software Requirements

Figure 4.2 on the current page shows the constraints made on software requirements. The dashed line running between system, software and hardware requirements emphasizes the cognitive borderline when it comes to decision-making how the system should be composed.

A first analysis is made on system requirements in order to elicitate the specific software requirements for the software components to be developed. As for safety-critical systems, [Soc96a] describes the approach to filling this gap between system and software development. [Soc96a] describes a system that is decomposed into items

for which requirements are stated and then iteratively assigned to software or hardware components.

However, the flow of information from the system development process to the software development process has to focus on the correct, complete and consistent tracking of requirements to which the selected software components are allocated. In this context, requirements contributing to system safety are of particular importance. On the other hand, any modification of software requirements must not affect system safety, particularly when derived software requirements have to be introduced into the software system development.

Once the software requirements have been elicited, they constitute the key input for creating the corresponding software architecture. The design process for a software architecture, which has to be done in combination with the underlying hardware, results in modifications of the requirements specification. Furthermore, the proposed hardware implementation has a reverse impact on the elaboration of complete and consistent requirements [Lia00].

The development of safety-critical systems requires traceability to be established from the customer's idea to the final validation, with the latter constituting the end of a software development process model (for example, the V-model). The problem domain can be solved by putting the question *What?*, which is followed by *How?* in order to increase the understanding for a specific problem . This specific treatment of software requirements results in separating requirements into *high-level* and *low-level* requirements.

High-level software requirements (HLR) specify the behavior of the software system by considering the question *"What shall the system do ?"*.

Low-Level software requirements (LLR) introduce architectural considerations into the software requirements stage by answering the question *"How shall the system be implemented ?"*.

The two-level approach to specifying software requirements for a safety-critical system creates a relationship between HLRs and LLRs (that is, traceability), where each HLR can be traced to one or more LLRs and vice versa. Software implementation should directly start from the specification of a low-level software requirement [Rad92].

4.2.2 Software Safety Requirements

"Safety requirements derived through safety analysis will place integrity constraints on existing core functions [AK01]." With regard to the system development process, these safety requirements basically reside at the system level. The specification of the software component behavior has to include software-related aspects of the original safety requirements by providing traceability information. The relationship between safety requirements and software components is the starting point for determining software safety requirements.

Due to an existing knowledge structure in the safety-critical domain (for example, standards documents or national authorities), safety requirements, according to [Fed00], also exist in the form of *generic* or *specific* software safety requirements.

Generic software safety requirement (GSSR): A GSSR is an existing solution to a safety problem. The problem description addresses a situation taking place within a specific safety-critical domain. Furthermore, the knowledge base is aimed at assessing potential hazard sources in the system design process at an early stage.

Specific software safety requirement (SSSR) An SSSR results from hazard analysis activities done to find a particular solution. Thus an SSSR implies the implementation of a specific functional aspect ensuring a certain system behavior.

Table 4.3 on the following page lists examples of GSSRs. They address an abstractly safe behavior that can suit different kinds of systems for the same technical domain. The examples are extracted from a safety-critical knowledge base [Fed00].

Before giving examples of specific software safety requirements, we have to solve a dilemma that results from different safety approaches employed by the industry. In general, software requirements have only a part of software safety requirements, which depends on the safety approach used for the elaboration of system hazards. If a particular software solution is chosen to be implemented for a system, every derived software safety requirement reflects a SSSR. Safety approaches, however, have different meanings of how software safety requirements contribute to the software system behavior.

Generic Software Safety Requirements
GSSR 1 The failure of safety-critical software functions *shall* be detected, isolated, and recovered from such that catastrophic and critical hazardous events are prevented from occurring.
GSSR 2 Software *shall* process the necessary commands within the time-to-criticality of a hazardous event.
GSSR 3 Software *shall* provide error handling to support safety-critical functions.

Table 4.3: Generic Software Safety Requirements [Fed00]

For this reason, a comparison between two safety approaches is given below [Lev03] :

System Safety As mentioned in Section 2.3.2 on page 14, system safety determines its safety requirements at system level and propagates them to the software when a software component contributes to an identified system hazard. Software requirements therefore integrate a safe software system behavior.

Industrial Safety Whenever there is a hazardous situation, industrial safety focuses on reducing the likelihood of recurrence instead of reconsidering and changing the whole system. Thus safety requirements for industrial applications are specified for the potential operation mode of a system, therefore focusing on a specific part of that system.

An example of an SSSR following the industrial safety approach can be found when implementing a system that reflects *functional safety*. In this master's thesis the term "system safety" is referred to as "safety approach". System safety implies an integrated approach to software safety requirements. Furthermore, a software safety requirement is specific with regard to the overall design decision. The integrated approach allows shortening the term "software safety requirements" to "software requirements" and vice versa. Unfortunately, safety requirements on system level do not allow investigating a specific and isolated safety-critical part of the software system behavior.

4.3 Hazard Analysis Techniques

Safety requirements are derived from a hazard analysis. There is a large number of hazard analysis techniques, with two such techniques being described in this section. A fault-tree analysis (FTA) identifies a hazardous situation by decomposing functionality into sub-functionality and examining the potential hazardous relationship. A Hazard and Operability Study (HAZOP) provides an iterative and systematic process that reviews the design on the basis of a set of negative interrogatives.

4.3.1 Software Fault Tree Analysis

The *fault tree analysis (FTA)*, which is widely used in the safety-critical domain, examines an existing design if unexpected malicious events can occur. The Software FTA reflects a static analysis technique used to decompose a system or component into its elements, with all their relationships being shown. The graphical method is used in design and development. Whereas the HAZOP method (see Section 4.3.2 on page 47) deals with the process of identifying hazards, the FTA is aimed at determining the cause of hazards in the design.

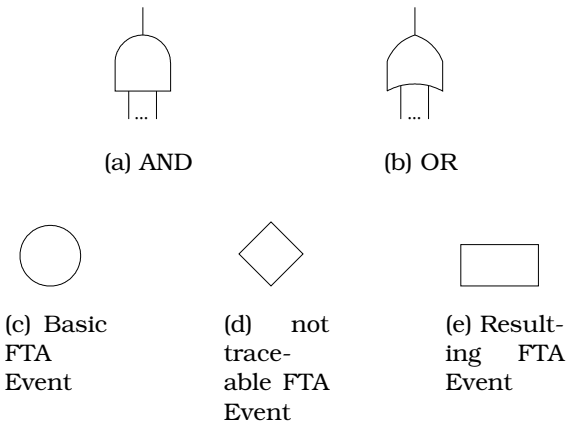


Figure 4.3: Fault Tree Analysis Elements

Figure 4.3 on the current page lists FTA elements. The elements AND and OR connect the events finally leading to a hazard. The final event, the so-called *top event*, is shown on top. The two logical operators can have one or more inputs, but only one output. FTA events can be grouped into three categories:

1. A circle represents a *basic event* (see Figure 4.3c on the preceding page). A basic event identifies a potential malicious situation within the system and can be connected by a logical AND or OR operator.

2. A rhombus represents a *fault event* (see Figure 4.3d on the previous page). A fault event cannot be traced back to its origin malicious source.

3. A rectangle represents a *resulting event* (see Figure 4.3e on the preceding page).

A typical FTA starts with a set of basic events on the bottom, with the relation between two events being expressed by a logical operator. But events can be connected even with no logical operator in between. Doing an FTA in the safety-critical software domain yields an hazardous output on top of the tree. Analyzing the progression of events has two objectives. An FTA is done in order *"to find paths through the code from particular inputs to these outputs or to demonstrate that such parts do not exist [LCS91]."* All event types shown in Figure 4.3 on the previous page are considered as inputs, whereas the top event is the output of the FTA. During the construction of a fault tree one successively asks the question *"How could this fault event occur ? [B.S03]"*.

In the case of software, we use failure mode templates for a certain programming language to do an FTA. The events shown in those templates correspond to the semantic information provided by the respective programming language. Figure 4.4 on the following page, by way of example, shows a possible assignment template.

The hazardous output yielded by the FTA is defined in the *top event*. Situations that might lead to this particular output are listed below the top event. The logical OR states that the system is in a hazardous state even if only one of these basic or non-backward-traceable fault events occurs. If the FTA results in such a path, the design or code implementation of the software program has to be modified.

An FTA is also used in risk analysis [B.S03]. The idea behind software FTA is described in [LCS91]. In this context, [Sto96] and [Kop97] give a more detailed description of the system FTA.

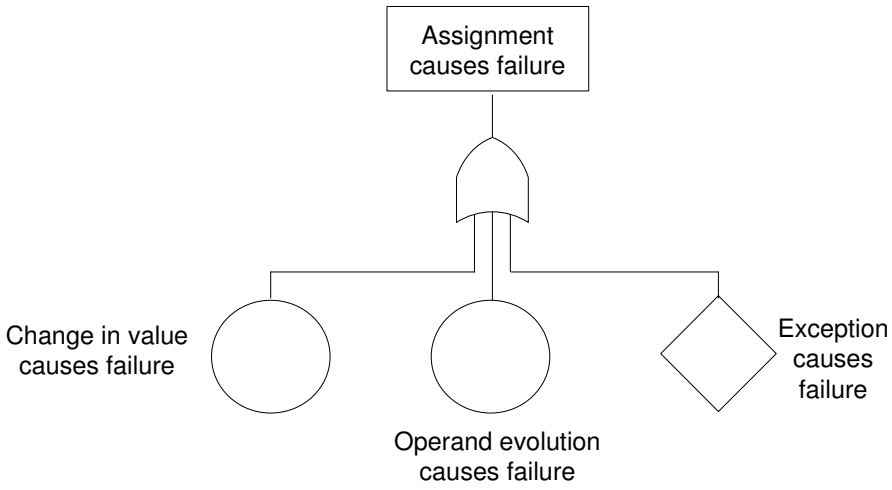


Figure 4.4: Assignment SFTA Template

4.3.2 Software Hazard and Operability Analysis

The *Hazard and Operability Analysis* (HAZOP) is usually done at the beginning of the system development process. In the case of software, hazards do not lead to a direct accident. Software hazards only contribute to an overall system hazard. Thus, also a software component within a safety-critical system can be likely to be responsible for an accident.

Basically, the HAZOP is done at system level. At this level, however, the results of the HAZOP have a significant influence on software design and software implementation. The HAZOP does not only identify a hazard present in the system, but it also determines the operations to be done in the event of such a hazard occurring. Based on the underlying accident model of the system, the HAZOP makes use of system theory and helps an engineer in discovering where the design and the intended operations deviate from each other.

The HAZOP analyzes every process unit and its relationship to other system entities. The objective of the HAZOP is *"to find all possible deviations from the design's expected operation and all hazards associated with these deviations [Lev95]."*. Thus, the HAZOP is suited to analyze any kind of hazardous situation that might occur in every part of the system. Specific questions have to be answered when creating an evidence of a hazard-free design solution. The

questions to be answered in the design review use specific guide words. Table 4.4 on the current page shows the *HAZOP guide words* and their meanings [Lev95].

Guide words	**Meaning**
NO, NOT, NONE	The intended result is not achieved, but nothing else happens.
MORE	More of a relevant physical property than there should be.
LESS	Less of a relevant physical property than there should be.
AS WELL AS	An activity occurs in addition to what was intended, or more components are present in the system than there should be.
PART OF	Only some of the design intentions are achieved.
REVERSE	The logical opposite of what was intended to occur.
OTHER THAN	No part of the intended result is achieved, and something completely different happens.

Table 4.4: HAZOP Guide words

Considering a concrete system design, we can apply the HAZOP guide words to any variable of interest (for example, to system component, a resulting temperature value or a relation between engine and sensor). Furthermore we investigate any existing relationship of the variable to other system entities by including the HAZOP guide words. If an unexplained situation has been discovered by applying the guide words listed in Table 4.4 on this page, the systematic review activity has found a potential source of a malicious situation. Each of the hazardous situations detected is logged by a so-called *HAZOP report*. This report contains information about the HAZOP guide words used, any deviation found in the system design, the possible causes for that deviation and the likely consequences if that deviation would occur in actual system operation.

A HAZOP can be done for every instance of the system development time-line. However, it is recommended to apply this hazard analysis technique to the earliest stages of the development process. An iterative HAZOP review process can also be used in terms of risk analysis [B.S03].

Chapter 5

Application-Oriented Requirements Classification

The system safety approach determines safety requirements at system level. After the design decision has been made, software component development inherits safety-critical behavior from system level when software requirements are specified. This means that safety-critical concerns are still within a classification of software requirements, which constitute the focus of this thesis. However, to understand the problem domain, we assume a dependable real-time system as a system platform. Then we have to assign a novel platform-related concept to the behavioral description of the real-time system. This novel core concept, which is inspired by the MDA [MM03], is the platform layer (PL). Requirements address the behavior of one or more platform layers. Furthermore, a requirement itself is interpreted as a textual arrangement of system properties in natural language. Based on these system properties, the requirement classification pattern (RCP) determines the according requirement classification. The RCP provides four different requirement classification windows (RCW), which result from a conceptual combination of MDA platform viewpoints [MM03] and the development of requirements according to RTCA/DO-178B [Rad92]. Classification axioms finally advise engineers on how to deal with requirements using the RCP.

5.1 Platform Behavior Specification

The *model-driven architecture (MDA)* [MM03], with its proposed concepts, relationships and objects provides ways to find adequate solutions to the reuse of safety-critical requirements specifications. In the MDA, the notion of *platform* constitutes the integral core component for any system to be developed. The platform concept is mapped to the real-time system domain and extended by software-related and distributed, time-triggered issues. To this viewpoint the RTCA/DO-178B, with its two-level requirement development strategy, is added and embodies the comprehensive safety development.

5.1.1 Real-Time System Platform Determination

Considerations made on the classification of requirements always start with determining the *platform* for the respective real-time system, which is a core concept of system development in [MM03]. A system that is a platform has *platform-specific* and *platform-independent* functional characteristics. Thus we assume that a system behavior is also classified into a platform-specific and platform-independent part.

Definition 5.1 (Platform (MDA)) *"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern the details of how the functionality provided by the platform is implemented [MM03]."*

Software is part of real-time system co-design. Co-design encompasses hardware design and software design. The platform concept in [MM03] focuses on the software domain. Thus relevant concepts of the MDA are introduced in this and the following Section 5.1.2 on page 52 via showing its relevance not only for software itself but also for behavioral aspects of real-time systems.

In contrast to conventional software (such as desktop software for conventional PCs) and its development domains, the real-time software domain is constrained by a close relationship of software to its underlying hardware target. For this reason, the real-time software platform inherits the underlying hardware set of subsystems and technologies. Specific services extend this inherited functionality through proper and well-defined interfaces at software platform level. In other words, the real-time software platform is the platform

application running on hardware. Furthermore, the co-design principle has the advantage that it helps to specifically overcome design problems at hardware level using the software platform stage and vice versa.

An example of the considerations made on extending the original platform is the implementation of a timer unit that might not exist on the hardware platform. In this case, the timer unit has to be provided by a service on the software platform side. As far as the distributed, time-triggered real-time domain is concerned, the system to be developed is embedded in a specific network, which provides the system environment with distributed services. Thus, the real-time system with its real-time software platform becomes a new platform used to provide the basis for distributed software applications. Figure 6.1 on page 67 show the abstraction levels of system platforms with regard to distributed time-triggered real-time software systems, with these abstraction levels being described as a case study in Section 6.2 on page 67.

In Figure 5.1 on the following page, a real-time system scenario illustrates the dependency between a hardware and software platform within a real-time system. A sensor measures the environment and provides the hardware platform with an analog value, with the hardware platform providing the overlying software platform with functionality. The sensor determines the correct actuator value and sends it through the hardware platform to the actuator. The platform principle ensures that the value determined is also sent to platform levels lying over the software platform shown in Figure 5.1 on the next page.

Once the hardware and software platforms for the real-time system to be developed have been set up, the system behavior is separated into *platform-specific* and *platform-independent* conceptual information. Conceptual information includes a summary of technological aspects of the system to be developed (for example, requirements, design or development procedures). Platform-independent information about the requirements specification allows transferring existing specific, behavioral knowledge from one platform to another within the same technological domain, which ensures different kinds of real-time system behaviors to be platform-independent, as long as the transfer of behavioral knowledge does not go beyond the domain boundary.

In case of real-time software, examples for an adequate software platform can be found in the field of real-time operating systems.

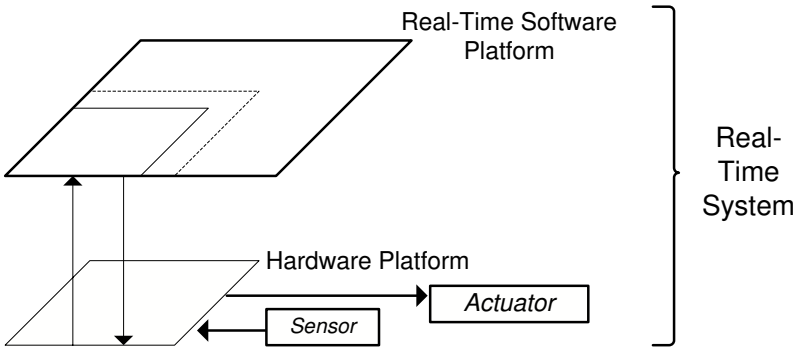


Figure 5.1: Real-Time System Platforms

5.1.2 Real-Time System Platform Viewpoints

Establishing a real-time system platform divides conceptual platform information into a platform-independent and platform-specific part. Following this conceptual separation, however, is not always suitable for the domain of requirements. Requirements describe the intended system behavior. The MDA is aimed at establishing a final model of the system, in this case a real-time system. A model is a specification of architectural considerations made on the system to be developed. If the specification process focuses on the system behavior, different *viewpoints* of the system behavior are required. A system behavior therefore is treated as blackbox, providing certain capabilities towards the environment and its user (or other systems). Knowledge about the insight or on how the real-time system is not important in order to describe the behavior. In classifying requirements, internal aspects and thus the intrinsic model are neglected in this work. Different kinds of viewpoint to a system, however, ensure a effective transparent and common way to propose how a real-time system should work in the proposed domain.

Definition 5.2 (Viewpoint) *"A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system [MM03]."*

Definition 5.2 on this page describes a viewpoint as defined in the MDA. A *view* of a system is the resulting *viewpoint model*.

Hence, the behavioral information about a platform is considered from a platform-independent and platform-specific viewpoint.

The ***platform-independent viewpoint (PIV)*** focuses on those parts of the system behavior which are not specific for a certain system or system platform. For this reason, the PIV provides all information about the system behavior that does not change when switching from one platform to another within the same technological domain.

To establish a transparent view of the complete system behavior, the ***platform-specific viewpoint (PSV)*** includes the PIV and additionally focuses on the specific implementation and environment in which the system will be executed.

5.1.3 Platform Layers

The nature of a software system platform for a real-time system is typically very complex and unbounded. For this reason, there are approaches to software architecture that advocate the use of layers to modularize the software design. As far as software system behavior is concerned, we render the *platform* concept described in Section 5.1.1 on page 50 more precisely by introducing the term *platform layer (PL)*. A platform layer defines a conceptual part of the entire system platform behavior. There are hardware and software platform layers, with this thesis focusing on software platform layers. Thus, the view of the platform layer of a system significantly differs from the conventional platform perception, as described in Definition 5.1 on page 50.

Figure 5.2 on the next page shows how this novel concepts of a platform layer relates to the system platform and the system platforms behavior. The platform layer *PL b* abstracts platform behavior specified in platform layer, *PL c* and *PL d*. The according system behavior covered by platform layer *PL b* is gray shaded. When following the usual platform perception in Definition 5.1 on page 50, then a conceptual change (for example, a change in the system environment) made below the entire platform is not visible to the user. Changing the intended functionality below the system platform impacts on the classification of requirements to be specified for the entire system.

Each platform layer maps only to fractions of the system behavior. Thus, if the environmental conditions change, this allows a modular change of the requirements specification. This modular specification addresses hardware and software functionality for

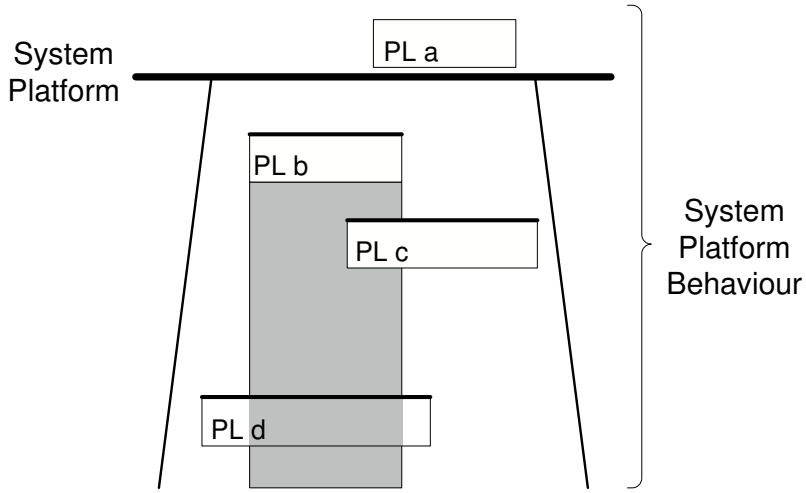


Figure 5.2: Platform Layer Characteristic

the platform below platform *PL a* (see Figure 5.2 on the current page). For this reason a platform layer is considered as a conceptual container that stores requirements. The kind of platform layer is defined by its type or the requirements specified for it. A platform layer defines boundaries between other behavioral parts of the system and finally determines the domain in which the system resides. As described in other approaches to requirements modeling [Dav93] [YZ80], the concept of *container-oriented* platform layers facilitates the requirements specification process. In terms of classification, a platform layer limits the scope of behavior specification and additionally makes it possible to judge, whether a requirement is platform-specific or platform-independent. Section 5.2 on page 56 gives a detailed description of the proper classification strategy to be employed.

The arrangement of platform layers within a system platform follows traditional layer-oriented concepts (for example, horizontally or hierarchically arranged layers). For the classification of requirements the system platform boundaries limit the range in creating platform layers. Any behavioral aspect outside the system platform boundary is assumed to be platform-independent. The functional contents of a platform layer mostly comprise a specific system component of the entire system platform.

The structure of a platform layer and its boundaries are based on subjective considerations made on the modularity of the system to be developed. Basically these subjective considerations are similar

to that of the object-oriented analysis [CY91]. But instead of offering a model, only revealing entities and their contribution to the entire system behavior is from interest. Due to the complexity of real-time systems, different numbers of platform layers may placed into a system platform. Such subjective considerations furthermore result in a dilemma in the specification of platform layers for a concrete system platform. Figure 5.3 on the current page shows an example illustrating the dilemma of different platform perceptions. The plat-

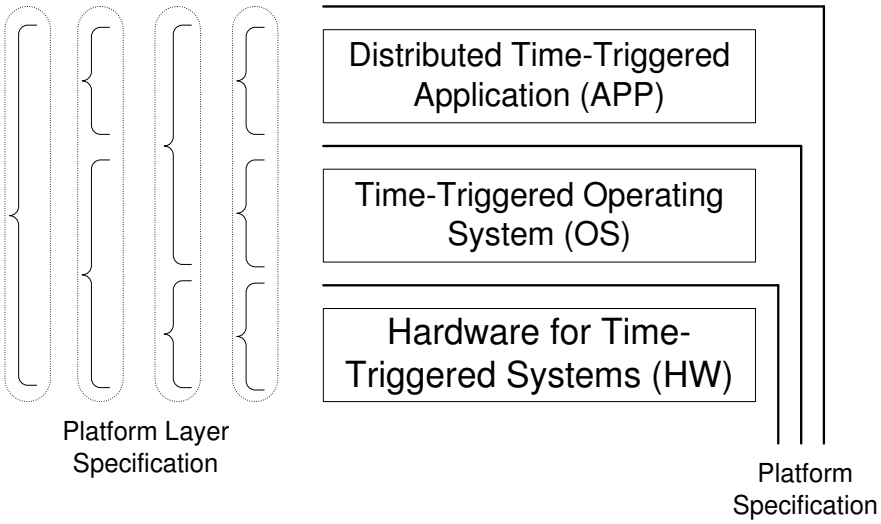


Figure 5.3: The Platform Layer Specification Dilemma

form layer specification modes, shown in Figure 5.3 on this page, have four possible platform layer structures, which are displayed in ovals. The braces represent the respective platform layer perception possible for the system. A feasible structural solution to the perception of platform layers is to separate the whole system platform behavior into three platform layers. This solution constitutes a conventional, initial description, which is displayed with rectangles in Figure 5.3 on the current page. Analogously we determine three (system) platforms labeled as *Platform Specification* in figure Figure 5.3 on this page. Platform layers comprising software-specific items (for example, *APP* and *OS*) can be be combined into one software platform layer. A special case is the platform layer specification displayed rightmost in Figure 5.3 on the current page. In this case, the platform layer equals the entire system platform.

A specification strategy should basically be used to find an appropriate solution to this dilemma. The strategy employed should make use of any domain knowledge existing. Information structures

based on domain knowledge can support a comprehensive statement limiting the platform layer. Furthermore, a modular specification approach should reflect a minimum of cognitive distance, as far as platform layers and their (system) platform are concerned.

5.2 Platform Behavior Classification for Real-Time Systems

A platform used for real-time systems integrates functionality provided by software and hardware. Platform layers divide the entire specification of the system behavior into manageable functional specification parts. This concept helps in classifying software and hardware requirements for a dependable real-time system.

5.2.1 Viewpoints and Cognitive Distance

Cognitive distance between the development of two systems is the amount of intellectual effort expended by an engineer to take the system from one stage of system development to another. Cognitive distance cannot be measured using units and numbers. Instead, it is an informal notion in supporting the evaluation of the effectiveness of an approach to system reuse. According to [Kru92], an approach of applying cognitive distance should follow three characteristics:

a) using fixed and variable abstractions that are both succinct and expressive;

b) maximizing the hidden part of an abstraction;

c) using automated mappings from abstraction specifications to abstraction realization.

A *viewpoint* following the explanation given in Definition 5.2 on page 52 complies with *a)* and *b)*, but cannot provide a specific mapping scenario demanded by *c*. In this work the focus is on requirement classification, neglecting mapping scenarios. The *platform-independent viewpoint (PIV)* is an appropriate means for minimizing the cognitive distance. The PIV embodies an information hiding policy.

System implementation details are not of interest to the PIV. Therefore, commonalities of requirement-specific information exist-

ing among different systems of the same domain are established by using the PIV.

The PIV is limited by the *domain boundary* of the system domain. Existing between a domain network, which constitutes a set of domains, the domain boundary is driven by the scope of the system and its application [PD90]. A domain boundary furthermore defines common behavioral entities, operations and relationships of the platform-independent view. It also sets limits to its operational capability, as far as the underlying platform-specific view of the set of software and hardware requirements is concerned. Hence, the PIV is considered as an instrument for *domain analysis*.

For the classification of requirements, the PIV focuses on existing information structures specified in software requirements documents. These documents contain requirements addressing different platform layers and follow the certification guidelines proposed by RTCA/DO-178B, which divides the entire information about a platform layer into a structure of high-level and low-level aspects.

5.2.2 System Properties and Requirements

A behavioral information structure specifies the entities of a system to be developed and the relationships between those entities. An entity that characterizes the system behavior is called *system property*. System properties reside within hardware and software platform layers, which allows the use of platform layers for real-time systems. System layers can additionally belong to one or more platform layers within a system platform.

Definition 5.3 (Requirement (1)) *"We define a requirement as any function, constraint, or other* ***property*** *that must be provided, met, or satisfied to fill the needs of the system's intended user(s) [Abb86]."*

Definition 5.3 on the current page outlines the close relationship between (system) property and requirement. This thesis deals with requirements, as introduced in Definition 5.4 on this page. Thus the requirement statement has to include one or more system properties in its specification. Even the diversity of natural language leads to requirements, where a property defines further properties.

Definition 5.4 (Requirement (2)) *"A requirement is a natural language statement describing the intended system behavior by addressing* ***one or more*** *system properties."*

Property Example	Requirement Statement
1	**Req1** The system *shall* provide a service that initiates the shutdown sequence.
2	**Req2** The `setup_timer` function shall configure the time source of `timer1` and `timer2`

Table 5.1: System property examples

A system property can describe any behavioral aspect of the system platform. Examples for a system property are objects, relationships, goals or system states. On the basis of the platform layer concept described in Section 5.1.3 on page 53, we assume in this thesis that a platform layer is specified by platform-specific and/or platform-independent properties. Furthermore, a requirement can address different platform layers within the same system platform.

In order to understand how to determine a property within a natural language requirement, two sample requirements are given in Table 5.1 on this page. **Req1** and **Req 2** contain system properties of the intended system, which are underlined. Those properties address behavioral aspects of different platform layers.

5.2.3 Requirements Classification Pattern

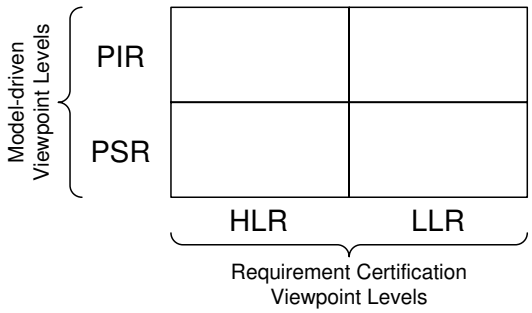


Figure 5.4: Requirement Classification Pattern

Platform-specific and platform-independent properties determine the system behavior of the platform layer stage. In addition to the model-driven platform layer aspects, the classification has to integrate safety-critical aspects. Safety-critical systems have to follow

recommended practices of requirements development. Considerations made on requirements, as proposed by RTCA/DO-178B, divide the system behavior into a high-level and low-level stage, with traceability linking the high-level and low-level requirement stages with each other. RTCA/DO-178B itself addresses certification aspects concerning the safety-critical software domain. The platform layers inherit this software requirement development strategy from the classification of requirements.

A *requirements classification pattern (RCP)* is used for the classification of requirements, as shown in Figure 5.4 on the previous page. The basic idea behind this pattern is to combine the model-driven aspects, as described in Section 5.1.2 on page 52, with the requirements development strategy recommended by the guidelines of RTCA/DO-178B. The use of different platform layers allows a classification of requirements for the entire platform system including software and hardware.

Description	Abbreviation
Platform-**I**ndependent **R**equirements	PIR
Platform-**S**pecific **R**equirements	PSR
High-**L**evel **R**equirements	HLR
Low-**L**evel **R**equirements	LLR

Table 5.2: Four different viewpoints to the platform layer behavior

The four viewpoints of a platform layer, as listed in Table 5.2 on the current page, span a two-dimensional space over the entire system behavior. *PIR* and *PSR* reflect the *model-driven* impact. They are summarized as *model-driven viewpoint levels*, whereas the high-level and low-level requirements are *requirements certification viewpoint levels* reflecting the safety-critical aspects of a requirements classification.

Figure 5.4 on the preceding page divides the "pattern space" into four equal-sized rectangles, each representing a so called *requirement classification window (RCW)*. Table 5.3 on the next page describes the four different RCWs. It should be noted that a platform layer requirement resides within one of these four RCWs.

5.2.4 Classification Axioms

A requirement resides within one of the four requirement classification windows shown in Figure 5.4 on the preceding page. The four different viewpoints (see Table 5.2 on the current page) de-

Viewpoint Combination	Description	Requirement Classification Window [1]
HLR/PIR	A **r**equirement at **h**igh-level stage and platform-**i**ndependent.	RHI
HLR/PSR	A **r**equirement at **h**igh-level stage and platform-**s**pecific.	RHS
LLR/PIR	A **r**equirement at **l**ow-level stage and platform-**i**ndependent.	RLI
LLR/PSR	A **r**equirement at **l**ow-level stage and platform-**s**pecific.	RLS

Table 5.3: The four requirements category windows of the RCP

termine an approach to assigning an individual requirement to its requirement classification window. Software and hardware requirements are stated in requirements documents. RTCA/DO-178B recommends separating requirements into high-level and low-level requirements and specifying them in separate requirements documents. If such a separation of requirements has not been done already, a set of requirements is separated into the two categories mentioned by using the *What?/How?* interrogative principle[2]. Properties also determine whether a requirement is platform-specific or platform-independent. The platform layer concept has an essential influence on the determination of platform-specific properties.

Definition 5.5 (Platform-specific Property)
"A platform-specific property of a requirement is a property that describes the intended behavior of a platform layer that is ***below*** *the platform layer of the requirement."*

Definition 5.5 on the current page is the basis for the separation of requirements into platform-specific and platform-independent requirements. Thus, the treatment of properties also plays a decisive

[2]The *What?/How?* interrogative principle is used to separate the specified system behavior into a high-level and low-level requirements stage (see Section 4.2.1 on page 41). In this thesis, we assume that a separation of requirements into high-level and low-level requirements is available due to the reuse of requirements in the course of certification activities.

role in applying the RCP.

Classification Axiom 5.1 *If a requirement contains a low-level property, the requirement is a low-level requirement (LLR).*

Classification Axiom 5.2 *If a requirement does not contain any low-level property, the requirement is a high-level requirement (HLR).*

Classification Axiom 5.3 *If a requirement contains a platform-specific property, the requirement is a platform-specific requirement (PSR).*

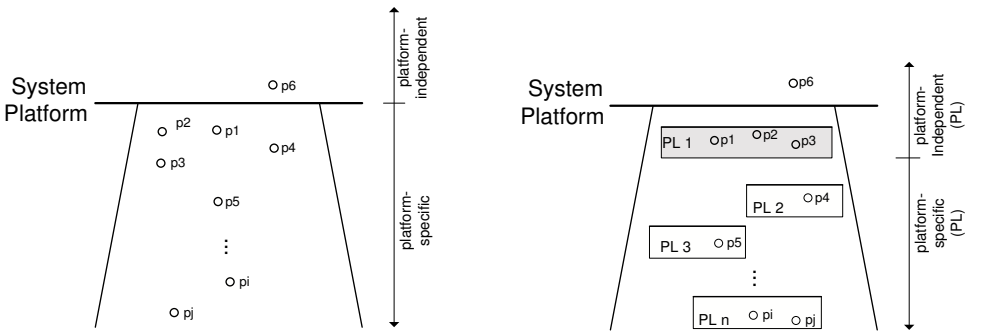
Classification Axiom 5.4 *If a requirement does not contain any platform-specific property, the requirement is a platform-independent requirement (PIR).*

These four axioms enable a transparent RCW assignment procedure for requirements. Axioms 5.1 and 5.2 deal with the certification viewpoints, whereas Axioms 5.3 and 5.4 refer to model-driven viewpoints assessing the properties specified in a requirement statement.

The classification axioms enable a comprehensive assignment of requirements to their respective requirement classification window. Before classification takes place on a system platform, as it is illustrated in Figure 5.5a on the following page, a set of platform layers has to be specified, as depicted in Figure 5.5b on the next page. Furthermore Figure 5.5b on the following page illustrates a system platform, in which different kinds of platform layers encompass different kinds of system properties.

In Figure 5.5a on the next page only one property is platform-independent because the property resides above and thus outside the system platform. The novel concept of platform layers, which is introduced to the system platform in Figure 5.5b on the following page, allows to take a look inside the system platform behavior and its system properties. In this context the use of specified classification axioms focuses on a classification of requirements belonging to platform layer *PL 1*, which for clearness reasons is grey shaded. Thus the separation into platform-specific and platform-independent system behavior has to be aligned to the position of *PL 1* within the system platform.

After determining the system platform with its platform layers, we are now able to define an examples for demonstrating the use



(a) System platform and system properties without platform layers

(b) Introduction of platform layer to system platform and system properties

Figure 5.5: System platform and platform layers

of the four classification axioms. A set S of requirements r with a number of properties p defined for the system platform shown in Figure 5.5b on the current page are as follows:

$$S = \{r_1(p5), r_2(p1, p2), r_3(p1, pi), r_4(p2, p3, p4), r_5(p1, p4), r_6(p6, p3)\}$$

The set S comprises requirements that are documented in a requirements specification complying with RTCA/DO-178B. In order to classify the set of requirements, classification axioms are applied to S. Examining the properties of the platform system shown in Figure 5.5b on this page yields the following results:

- *platform-independent* requirements are :
  $S_{pi} = \{r_2(p1, p2), r_5(p1, p4), r_6(p6, p3)\}$
- *platform-specific* requirements are :
  $S_{ps} = \{r_1(p5), r_4(p2, p3, p4), r_3(p1, pi)\}$

Set S_{pi} follows classification axiom 5.4. As there is no requirement with platform-specific information in S_{pi}, the latter shows platform independence. If there is even one property with platform-specific information, that requirement has to be added to S_{ps}. S_{ps}, however, can contain platform-independent information as can be seen in requirement $r_4(p2, p3, p4)$. S_{ps} is platform-specific, thus the corresponding requirements are platform-specific too.

As mentioned at the beginning of this section, certification guidelines complying with RTCA/DO-178B proceed from the assumption that requirements are separated into the categories “high-level” and

"low-level requirements". Once the platform-related partition has been determined, S_{pi} and S_{ps} can be embedded into the traceability scheme. A traceability scheme assigns low-level requirements to high-level requirements and vice versa. Moreover, a traceability scheme exactly identifies whether a requirement r is a high-level or low-level requirement. For this reason, the traceability scheme identifies the following requirements of set S as

- *high-level requirements* are :
  $S_{hl} = \{r_4(p2, p3, p4),\ r_3(p1, pi),\ r_5(p1, p4)\}$
- *low-level requirements* are :
  $S_{ll} = \{r_6(p6, p3),\ r_1(p5),\ r_2(p1, p2)\}$

When we combine the set of requirements S_{pi} and S_{ps} (see listing above) with the information provided by S_{hl} and S_{ll} we can determine specific sets for each requirement classification window (see Table 5.4 on the current page).

$S_{RHI} = S_{hl} \cap S_{pi} = \{r_5\}$	$S_{RLI} = S_{ll} \cap S_{pi} = \{r_2\}$
$S_{RHS} = S_{hl} \cap S_{ps} = \{r_4, r_3\}$	$S_{RLS} = S_{ll} \cap S_{ps} = \{r_6, r_1\}$

Table 5.4: Results of the requirements classification procedure

Chapter 6

Evaluating Requirements Classifications of Safety-Critical Software

The requirements classification pattern (RCP) introduced in Section 5.2 on page 56 can be applied to the field of safety-critical software development. For demonstration purposes, we examine a fault-tolerant real-time system, TTP-OS [TG94], to prove the practical use of the requirements classification pattern for the engineering of safety-critical requirements.

6.1 TTP-OS – A Dependable Real-Time Software Platform

6.1.1 TTP-OS (System Overview)

TTP-OS is a fault-tolerant, hard real-time operating system that comprehensively complies with the essential needs demanded by the safety-critical industry, namely:

- *robustness* with regard to design and accurate execution mode of the operating system,
- *resource efficiency* in terms of minimum CPU, RAM and ROM cycles,
- *fault-tolerant hard real-time behavior* that provides a generic mechanism preventing hazards from occuring in the operating system

- *system openness* with regard to ensuring a composable, reusable and maintainable software system design.

TTP-OS is conceptually divided into two components:

1. an **on-line** runtime kernel, which is the minimum set of functionality responsible for system execution, and
2. an **offline** node design component provided by TTP-Build[1]. All functionality that can be done offline is moved out of the runtime kernel and specified offline by TTP-Build, which complements the TTP-OS kernel, using the design data generated by the cluster design tool TTP-Plan[2].

Core aspects of TTP-OS in terms of functionality are as follows :

Task execution The core function of any operating system is the activation of tasks. TTP-OS uses a time-based scheduling policy, which is configured offline, and supports task preemption. A time-triggered task running in the system can always be in one of three possible states at any point in time: *running*, *ready* or *preempted*. "Preempted" means that a higher-priority task becomes ready to run at its statically defined activation time and preempts the currently executing task by moving the latter from the running state into the preempted state. There are two different kinds of time sources used to activate a task, the *local time*, provided by the CPU, and the *global time*, which is a system-wide, synchronized timebase, the fault-tolerant time provided by TTP. A deadline has to be specified for each task, with TTP-OS monitoring and checking this deadline for violations (deadline monitoring).

Fault management TTP-OS provides different kinds of fault management techniques that prevent hazardous situations from occurring in the system by avoiding *task blocking*, doing specific *configuration checks* and interacting with the *fault-tolerant communication layer*.
When a system error occurs, TTP-OS calls an error handler, which passes the exception code to the corresponding exception handler. When a task raises an exception, the exception

[1] TTP-Build is an offline node design tool that generates the interface between the cluster level and the node level (the application software) and that provides essential design data required for the execution of TTP-OS.

[2] Further information about TTP-Plan and TTP-Build can be found at http://www.tttech.com

code is passed to the corresponding exception handler, with further task execution depending on the respective exception configuration. An exception itself is configured by the application or TTP-OS.

Application mode(s) An application mode determines the temporal behavior of an application, which is the predefined points in time at which tasks become activated. An application can define several application modes that can be changed during runtime. For each application mode defined, the node design tool TTP-Build generates a configuration that stores the current state of TTP-OS. Interfaces provided by TTP-OS enable special mechanisms to protect the application modes defined.

6.1.2 TTP-OS Requirements Documentation

As TTP-OS was developed as a certifiable, safety-critical software product complying with the RTCA/DO-178B Level A guidelines [Rad92], it can serve as a good example of how to use the RCP. To this end, two documents were subjected to closer examination for this thesis:

1. the *Software Requirements Document (SRD)*, specifying the high-level requirements for the software implementation of TTP-OS, and
2. the *Software Design Document (SDD)*, specifying the low-level requirements for the software implementation of TTP-OS.

For the understanding of the requirement examples in this chapter, a deep technical understanding is not necessary. Such specific knowledge would reflect specific architectural considerations and concepts. However, a system behavior reflects a black-box-oriented view to the system, neglecting internal objects and relationships. Also the requirements were slightly modified to demonstrate the use of the RCP so that they furthermore can serve as an initial point for discussion.

6.2 TTP-OS Requirements Classification (Using the RCP)

Safety-critical systems following the distributed, time-triggered paradigm are developed for a specific hardware platform. The respective platform layer corresponding to the requirements classification of TTP-OS is determined by examining existing architectural platform considerations. Eventually, requirements are assigned to each requirement classification window (RCW), as described in Section 5.2.3 on page 58, using exemplary TTP-OS requirements.

6.2.1 Evaluating Time-Triggered Real-Time System Platforms

Applying the concept of a model-driven architecture (MDA) (see Definition 5.1 on page 50) to the domain of distributed, time-triggered real-time systems results in three fundamental platform specifications, shown in Figure 6.1 on the current page. These platform specifications limit the system domain boundaries for the platform layer determination, which is done in Section 6.2.2 on the following page. The partitioning of an entire system (or subsystem) into differ-

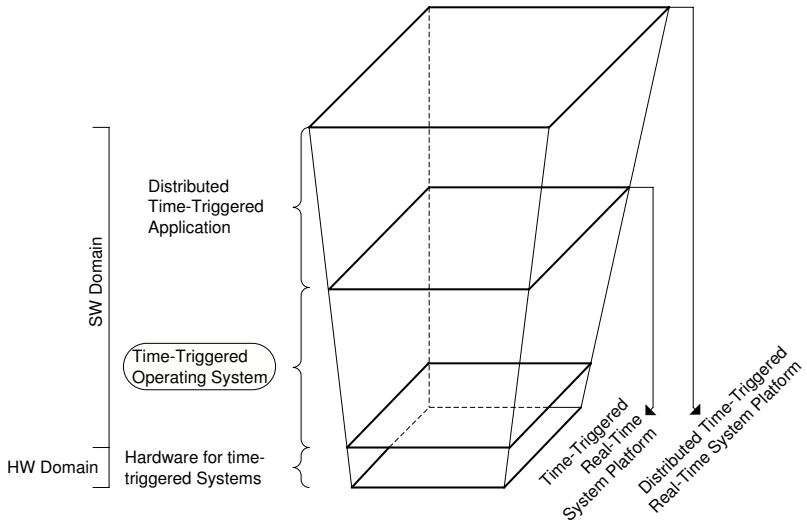


Figure 6.1: System platform specifications for distributed, time-triggered real-time systems

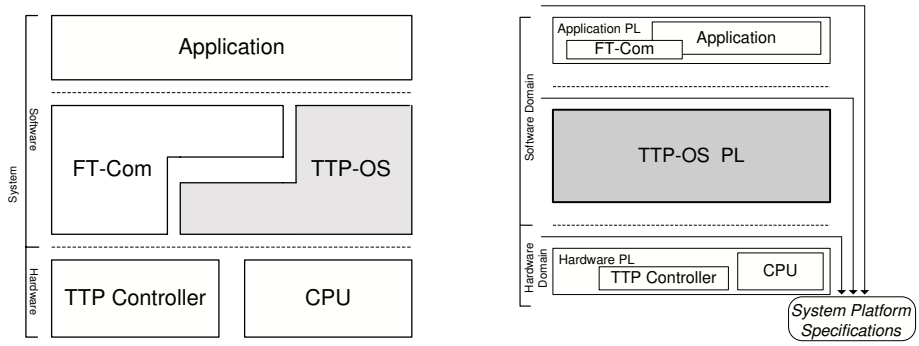
ent system platforms and corresponding platform layers may start with the *hardware for time-triggered systems*(see Figure 6.1 on the current page), which reflects the hardware platform of the system

to develop. Although we focus on software requirements, services provided by the hardware are not neglected in the platform model. Those services determine and possibly constrain the behavior of the software levels following above. On the next level, the *time-triggered operating system* ensures additional platform functionality to extend the services provided by the underlying hardware platform. Combining these two platforms can serve as a first interpretation of a real-time system (see also section Section 5.1.1 on page 50) and constitutes the so-called *time-triggered real-time software platform*. In most cases, this software platform functionality resides on the corresponding hardware entity locally, which is the main difference to the next platform lying above, the *distributed, time-triggered real-time system platform*. Finally, the system uses services provided by the application that runs on the distributed, time-triggered software platform to ensure a correct execution of the system functions.

The system platform used for the case study used in this thesis is the time-triggered operating system platform. The platform perception includes platform layers that inherit software and hardware issues.

6.2.2 TTP-OS Platform Layer Specification

The requirements classification pattern separates requirements of a specific platform layer into four requirement classification windows. Before classifying software requirements for TTP-OS, a corresponding TTP-OS platform layer has to be specified. Figure 6.2 on the current page shows a possible specification of platform layers that can be used for requirements classification. Basically distributed



(a) TTP-OS software architecture

(b) TTP-OS platform layers

Figure 6.2: TTP-OS platform layer specification

time-triggered real-time systems have three platform specifications, shown in Figure 6.1 on page 67. When assigning a platform layer to TTP-OS, we can transform these three platform specifications into three individual platform layers. Additionally, this transformation is conceptually merged with an existing architectural description. Figure 6.2a on the previous page shows the current architecture of TTP-OS, where a set of architectural layers is arranged hierarchically. In this particular case, the platform layer determination benefits from this existing layer-oriented separation. The resulting TTP-OS platform layer resides between application and hardware platform layer and summarizes all architectural concepts relating to the operating system.

The layer separation offers a way to change the behavioral structure of the original platform separation, shown in Figure 6.1 on page 67. A requirements specification done in terms of platform specification takes a bird's eye view on the system behavior. As described in Section 5.1.1 on page 50 this situation may lead to a restriction of the system platform behavior. Only functionality on top of the platform is visible to subsequent platform extensions. This assumption also holds for considerations made on the behavior described in Section 5.1.3 on page 53. Any information belonging to hardware- or software-specific issues are hidden, because an overlying platform makes use of platform functionality only by means of well-defined behavioral interfaces (for example, services). The TTP-OS platform layer provides these "access points" in terms of platform perceptions, but the platform layer boundaries allow a separation and encapsulation of properties residing only in the TTP-OS platform layer.

The lowermost platform (hardware platform) comprises the hardware platform layer. The hardware examined for this thesis includes a specific communication unit, the TTP controller. Additionally, the CPU of the hardware target is added to the hardware platform layer. Above the TTP-OS platform layer, which is of interest for this thesis, the FT-COM layer and the application constitute the application platform layer. Properties defined in the application platform layer are assumed to be platform-independent in the sense of Definition 5.5 on page 60. If software requirements include only properties of the application platform layer or properties combined with the TTP-OS platform layer, they are considered as platform-independent. In order to classify requirements with regard to TTP-OS, the separation, as shown in Figure 6.2b on the preceding page, correctly determines, whether a requirement is platform-specific or platform-independent.

6.2.3 RCP Sample Application For TTP-OS Certification Documents

Having defined a platform layer for its use with TTP-OS, requirements are assigned to their respective requirements classification windows. Table 6.1 on the current page and Table 6.2 on the following page list a set of high-level and low-level software requirements arbitrarily taken from the TTP-OS SRD and SDD.

The excerpt of sample requirements listed in these two tables is to show a first practical use of the RCP of the existing certification documents. The entire semantic information contained in the set of requirements focuses on time-triggered aspects of TTP-OS system behavior (for example, static scheduling).

Id High-Level Software Requirement Statement
r1 The TTP-OS service `Start TTP` *shall* start the schedule table associated with the time source of the TTP controller so that the schedule table is synchronized with the cluster.
r2 TTP-OS *shall* check the deadline of an application when the application task ends.
r3 TTP-OS *shall* activate all application tasks according to a time schedule calculated offline.
r4 Each time table in TTP-OS *shall* have its own time source.
r5 TTP-OS *shall* provide a service which updates the life-sign of the TTP controller.
r6 TTP-OS *shall* provide services to save and restore CPU resources to allow floating point operations for preempted and preempting tasks.

Table 6.1: Exemplary set of high-level software requirements S_{hl}

Both sets of requirements (S_{hl} and S_{ll}) contain *platform-independent* and *platform-specific* information about the system behavior in terms of system properties. Thus, in a first step, when identifying the properties of the two sets (S_{hl} and S_{ll}), it is essential to decide, whether the requirement is platform-specific or platform-independent. A solution to the way of identifying system properties is a systematic walk-through of the requirements, which extracts all the identified system properties into a separate list.

Such a list, which refers to the platform layer specified, is created systematically when the requirements are classified. In the

[3] "HLSR ID" stands for "High-Level Software Requirement Identifier". The HLSR column contains references providing traceability to the corresponding high-level software requirements.

Id Low-Level Software Requirement Statement	HLSR ID[3]
r7 TTP-OS *shall* calculate the start of the schedule table by increasing the time stamp as follows : ($time\_stamp = time\_stamp + \frac{cycle\_length}{2}$)	r1
r8 TTP-OS *shall* pass the time stamp of the time source as parameter when calling the service `hal_cur_time`.	r1
r9 The function `check_deadline` *shall* return value `DEADLINE_E` if the currently activated task chain violates its deadline.	r2
r10 The function `all_schedule` *shall* assign the following values of the schedule table state according to its time source : `status` to `STATE`, `cycle_start` to `start_time`, `int_time` plus `start_time` to recalculate `time_sync_phase`.	r4

Table 6.2: Exemplary set of low-level software requirements S_{ll}

TTP-OS context, such a list refers to the TTP-OS platform layer, so that this list is called *platform layer property list (PLPL)*. PLPLs for

schedule table
`Start TTP`
in the future
TTP Controller
floating point operations
task

Table 6.3: High-level system properties

`hal_get_current`
time_stamp
start of schedule table
`int_time`
`DEADLINE_E`

Table 6.4: Low-level system properties

S_{hl} and S_{ll} are listed in Table 6.3 on the current page and Table 6.4 on this page. Once identified, these lists support the separation of system properties into requirements corresponding to these properties. A PLPL property is furthermore marked as platform-specific for the TTP-OS layer, which is based on Definition 5.5 on page 60. Whenever the property of a requirement addresses a behavioral aspect of a platform layer below a platform layer of the requirement, then this requirement is platform-specific.

Based on the classification axiom described in Section 5.2.4 on page 59 we assume in terms of requirement classification that platform-specific properties are from more interest than platform-independent properties within a requirement. Thus a way to re-

duce the number of properties in the PLPLs is to omit platform-independent properties.

In **r5** the platform-specific property life-sign of the TTP controller resides in the hardware platform layer. The TTP controller is part of the hardware platform layer. Thus the life-sign resides below the TTP-OS platform layer. **r5** is a high-level platform-specific requirement.
The properties of **r3**, in contrast, describe a behavior characteristic of the TTP-OS layer. Moreover, **r3** does not contain specific information about any conceptual capability of the below residing platform layer. Therefore **r3** is a platform-independent requirement. A property referenced outside the system behavior boundary does not influence the platform-oriented decision process.
The same strategy can be applied to S_{ll}, following an iterative process to identify system properties with a list that corresponds to Table 6.4 on the previous page. The only platform-independent requirement in S_{11} is **r7** , where all the properties identified address behavior that is characteristic of the TTP-OS platform layer. The remaining requirements in S_{11} are platform-specific. Using classification axioms, as described in Section 5.2.4 on page 59, and Definition 5.5 on page 60 for the requirement sets S_{hl} and S_{ll}, we can describe the results in the following requirements classification windows (see Table 6.5 on this page):

$RHI = \{$**r2**, **r3**, **r4**$\}$	$RLI = \{$**r7**$\}$
$RHS = \{$**r1**, **r5**, **r6**$\}$	$RLS = \{$**r8**, **r9**, **r10**$\}$

Table 6.5: Requirements classification windows of S_{hl} and S_{ll}

6.3 Using the RCP for TTP-OS (Results)

Section 6.2.3 on page 70 gives us a first impression of how to use the requirements classification pattern. A detailed examination has been carried out on the TTP-OS requirements documents, with Table 6.6 on this page listing the total number of high-level and low-level requirements. Table 6.7 on the current page lists the results of

	Analysed Number (Existing Number)
High-level requirements	113 (113)
Low-level requirements	180 (559)
Derived Low-level requirements	74 (129)
Total number of classified requirements	**367** (801)

Table 6.6: Total number of all requirements for TTP-OS

the requirements classification, which has been done using the requirements classification pattern that is described in Section 5.2.3 on page 58, with Table 6.8 on the current page showing an additional RCW comprising a classification for derived low-level requirements.

$RHI = 71$	$RLI = 71$
$RHS = 42$	$RLS = 109$

Table 6.7: Number of classified TTP-OS requirements and their RCWs

This RCW extension was necessary because derived requirements do exist only at a low-level stage in the TTP-OS requirement documentation. Table 6.8 on this page lists the number of derived low-level requirements classified for TTP-OS.

	$RLI = 16$
	$RLS = 58$

Table 6.8: Evaluated RCWs for derived low-level TTP-OS Requirements

Establishing this overview of classification results was linked with problems concerning the identification of system properties. A

generic advice for estimating the reuse potential of a requirements specification was found, and finally a top-down order of RCW can support the development of requirements for reusable system components.

6.3.1 System Property Identification In Natural Language Requirements

Natural language as a representation to specify software requirements usually leaves room for more than one interpretation, which is due to the use of natural language itself. In some cases, considerations made on the representation of properties have led to several classification problems, discussions and solutions.
Sometimes, the classification of requirements done on the basis of the information provided can make it quite difficult to decide immediately, whether a property is platform-specific or platform-independent.

Therefore, the requirement classification process made on high-level and low-level requirement documents has resulted in the following *informal guidelines (IG)*, which can be used for making platform-oriented decisions. Using these informal guidelines, engineers are now able to elicit properties on the basis of a common requirements classification strategy:

- **IG 1** When no assumption can be made whether a requirement is platform-specific or platform-independent, assign the requirement – for safety reasons – to a platform-specific requirement classification window.
- **IG 2** If additional semantic information is referenced by or in the requirement statement, use that information.
- **IG 3** Requirements conforming to Definition 5.5 on page 60 and classification axiom 5.3 are platform-specific.
- **IG 4** Properties that do not reside within the system platform (specification) are platform-independent.
- **IG 5** Make a first iteration to determine the system properties for the platform-layer according to Definition 5.5 on page 60, make a second iteration to finally assign the requirements corresponding to their properties by using classification axiom 5.3.

In order to demonstrate the practical use of these informal guidelines, they are, by way of example, applied to the requirements given below.

HL Req 1 TTP-OS *shall* define two possible consequences for each exception: "Continue" and "Shutdown"

Requirement **HL Req 1** at first glance shows platform-independent behavior. With reference to *IG 2*, no additional information is provided. The requirement, however, specifies two possible consequences in the event of exceptions being raised, which in turn suggests a platform-specific, conceptual background. In order to avoid further discussion, we use *IG 1*, according to which **HL Req 1** is assigned to a platform-specific requirements classification window. Another good reason in favor of using *IG 1* is safety. Wrongly specified platform-independent requirements can result in wrong platform-independent designs, which finally can cause hazardous situations. Advocating a platform-specific solution protects the system from malicious situations and can resolve a potential ambiguity concerning the interpretation of natural language requirements.

LL Req 1 TTP-OS *shall* initialize the error handler by using the service routine `init_errorhandling` (Reference ref)[4].

Requirement **LL Req 1** at first glance suggests that the requirement statement specifies platform-independent behavior. To be on the safe side, we use *IG 2*, which allows including additional information specified in ref. This referenced information shows program-related details on `init_errorhandling` (for example, data types being used), which in turn is covered by *IG 3* so that we can clearly say that requirement **LL Req 1** is a platform-specific requirement.

HL Req 2 TTP-OS shall activate all application tasks according to a time schedule calculated offline.

The property `time schedule`, which is intended to be created offline is a suitable example for a property belonging to the system but residing outside of the system platform specification. *IG 4* is used, the requirement **HL Req 2** itself is platform-independent.

The impact of whether choosing platform-independent or platform-specific for a specific property is shown in example **HL Req 3**. The system property *time source* in this context literally is an abstract term but references to a hardware unit.

[4]In this context ref is used as an abstraction for content-related references stated within requirement specifications.

6.3.2 RCP Extension using a Domain Model

The modeling character of a platform layer is emphasized by [LMV97], where *"the process of creating reusable requirements is aided by having a road-map for structuring the domain and organizing reusable requirements knowledge."* The requirement examples **HL Req 3** raises another modeling issue resulting in an extension of the RCP by domain models.

HL Req 3 TTP-OS *shall* raise an exception *E1* if synchronization has lost one time source.

A property that describes a hardware entity resides below the TTP-OS platform layer (see Section 6.2.1 on page 67). In this context, the requirement is platform-specific, however, making the two iterations, as mentioned in guideline *IG 5*, may suggests that the system property `time source` still is platform-independent. This assumptions is true if we argue that `time source` resides within an domain model for the specific technological domain of time-triggered systems. Section 3.1.1 on page 22 briefly describes the concept of domain analysis and describes approaches how to create a domain model [KCH+90] [BdC91].

A vague domain model may also be developed based on experience made within a specific domain. For example a hardware engineer always assign a CPU or ROM unit to a hardware controller domain model. Thus following the time-triggered paradigm in [KB03], we assume a `time source` constitutes an core concept and thus an entity in the domain model developed for a time-triggered system platform as illustrated in Figure 6.2 on page 68.

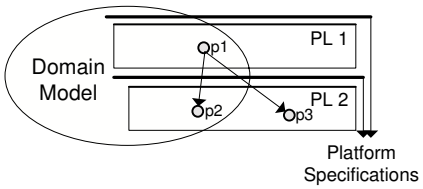


Figure 6.3: RCP extended by a domain model

Figure 6.3 on this page shows the concept of a domain model (see eclipse), which is integrated into the platform system and the corresponding platform layer concept. The domain model participate in the behavioral description of both platform layers PL 1 and PL 2. System properties may reside within or outside of the domain

model. Domain analysis identifies commonalities within a specific technological domain through this domain model. Thus these fractions of the platform layers, which are covered by the domain model, are platform-independent.

Furthermore this domain model extension affects the classification of requirements. Given a requirement r and the two system properties $p1$ and $p2$ in Figure 6.3 on the previous page, if we conform with Definition 5.5 on page 60, $r(p1, p2)$ is a platform-specific requirement. However, property $p2$ resides within the domain model, constituting a common behavioral aspect in the specified domain and thus has to be platform-independent.
Thus if a domain model is available, $r(p1, p2)$ is platform-independent. For this reason we have to extend Definition 5.5 on page 60 by the concept of a domain model.

Definition 6.1 (Platform-specific Property (Domain Model))
*"A platform-specific property of a requirement is a property that describes the intended behavior of a platform layer that is **below** the platform layer of the requirement and resides **outside** the domain model of the system platform."*

In context of Definition 6.1 on this page the property $p3$ in Figure 6.3 on the previous page is platform-specific and $r(p1, p3)$ a platform-specific requirement. In terms of TTP-OS requirements we assume that the corresponding domain model comprises only the system property `time source` as shown in figure Figure 6.4 on this page.

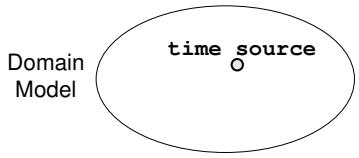


Figure 6.4: Simplified domain model of TTP-OS requirements used by RCP extension

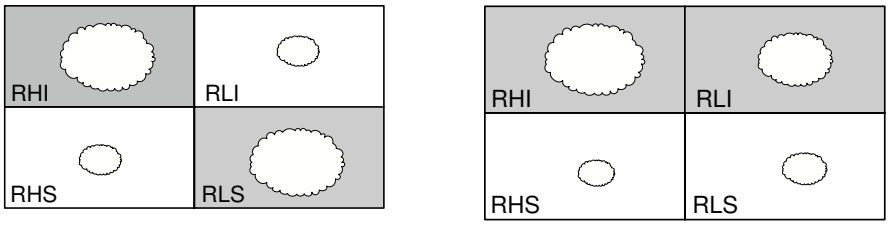
If we apply this assumption in combination with Definition 6.1 on the current page on high-level requirements in Table 6.7 on page 73, it will lead to a change toward a more platform-independent distribution of requirements within the corresponding RCWs. Table 6.9 on the next page provides the resulting RCW with its requirements distribution using a RCP extended by a domain model.

$\underline{RHI = 87}$	$RLI = 71$
$\underline{RHS = 26}$	$RLS = 109$

Table 6.9: RCWs using a domain model for property `time source`

6.3.3 Distribution of Requirements of a Platform Layer to enable Reuse

Table 6.7 on page 73 lists the results of the RCP examination made on the TTP-OS platform layer and shows that the majority of high-level requirements reside in the platform-independent RCW *RHI*, whereas the majority of the classified low-level requirements reside in the platform-specific RCW *RLS*. In Figure 6.5a on the current page the according distribution is emphasized through the gray-shaded RCWs.



(a) Conventional distribution of requirements in dependable real-time systems

(b) Proposed requirements distribution in dependable real-time systems for better reuse of requirements

Figure 6.5: Two requirement distributions of RCWs

One reason for that is the system safety approach, which at all requirements development stages requires traceability to be established for a dependable real-time system. Another reason for the assumed conventional distribution – in particular as far as *RHS* is concerned – is the fact that source code should be directly developed on the basis of these behavioral statements. A platform-specific low-level requirement describes the system behavour in terms of concrete implementation details. Thus a platform-specific low-level requirement instruments the developer more precisely in creating according source code than a platform-independent low-level requirement. If there were not any precise mapping strategy between the low-level platform-independent stage and the low-level platform-specific stage or source code (for example, code generator), there would be room for interpretation, which would cause ambiguity and a deviation from implementation, as described in Section 2.3.2 on

r_{ta} The application task *shall* activate `timerfunction(C167)`. The timer function `timerfunction(C167)` is a platform-specific property addressing functionality residing in an underlying platform layer . `timerfunction(C167)` monitors the temporal behavior of the application task, which is limited by a predefined temporal interval. *Attempt 1*: r_{ta}' The application task *shall* assign the predefined application task time interval to the 16bit timer of the hardware target. *Attempt 2*: r_{ta}" The application task *shall* activate a task monitoring function.

Table 6.10: Changing a platform-specific property to a more platform-independent property

page 16.

As far as safety is concerned, the AC 20-148 (see Section 3.1.2 on page 24) introduces a way of establishing and using a reusable certifiable component that is based on a worldwide recommended safety standard. A requirements classification is able to be reused if it it reflects a majority of high-level and low-level requirements residing in the platform-independent RCW. The gray-shaded RCWs shown in Figure 6.5b on the preceding page document this assumption. The question of how to create a more reusable requirements classification can be answered by moving platform-specific information toward platform independence while not tailoring the basic semantic information about the requirement.

The platform-specific requirements specified in the respective classification windows have to be modified to achieve more platform independence. An evaluation of the results, as given in Table 6.7 on page 73, does not always allow the creation of platform-independent requirements on the basis of corresponding platform-specific requirements. The example given in Table 6.10 on this page illustrates a possible change from a platform-specific property to a platform-independent property.

The timer function `timerfunction(C167)` sets a specific flag in the timer configuration that allows assigning a predefined time interval value to the application task. The system should check whether the task is executed within this predefined time interval or not.

If `timerfunction(C167)` were removed from r_{ta} , there could

be one possible solution to change the requirement. The abbreviation ta in r_{ta} stands for *timer activation*. Different hardware targets have specific timer configuration settings in common. This domain-related information is used by $\boxed{r_{ta}\text{'}}$. The property 16bit timer specifies a platform-specific requirement, but allows changing the underlying platform layer. A possible specific functionality is hidden by a different interpretation of the basic intent. In $\boxed{r''_{ta}}$, the abstraction change process is extended. The result is a platform-independent requirement, which is characterized by the property task monitoring function. Unfortunately, this requirement allows different implementations of the intended behavior. Thus $\boxed{r_{ta}\text{"}}$ is a potential source of behavioral hazards. With regard to requirement reuse, it is therefore more appropriate to change the original requirement in the sense of requirement $\boxed{r_{ta}\text{'}}$ rather than in the sense of $\boxed{r_{ta}\text{"}}$.

Additionally, the example given in Table 6.10 on the previous page shows that a change of requirements toward more platform independence is the harder the more platform-specific properties a requirement has. Requirements ambiguity based on semantic information change might lead to incorrect system behavior. For this reason, requirements should be changed if at least one platform-independent property is available[5]. A set of only platform-specific properties in a requirement always specifies the underlying platform layer. A requirement specifying different platform layer properties is also called *cross-platform layer requirement (CPLR)*.

6.3.4 Requirements Development Using RCW Concepts

The proposed classification of requirements offers a systematic reuse-oriented strategy in developing requirements. Basically it follows the development guidelines defined in RTCA/DO-178B [Rad92]. Based on the separation of requirements into high-level and low-level requirements as proposed by RTCA/DO-178B, adds the model-driven aspect to the development of requirements specifications.

The core concept of this development approach is the order of the RCW. For this reason, the pattern is split up into four classification windows, which are connected by transformation arrows. Each RCW describes a method to specify requirements. The result is shown in Figure 6.6 on page 82, which reflects a three-level requirements development process. The specification process for

[5]Note: A property itself may be a summary of properties, which makes it difficult to define “one” property.

a system to be developed has to pass through each of the levels shown.

RDL[6]	Requirements
1	**RHI1** The system *shall* activate all application tasks according to an off-line calculated time schedule.
2	**RLI1** A task chain *shall* have only one schedule interrupt. **RLI2** The task chain *shall* save the current task chain configuration after completion of every task. **RLI3** A service *shall* prepare the next scheduling interrupt using task-specific information. **RHS1** TTP-Build *shall* generate application-specific schedule configuration data for TTP-OS
3	**RLS1** The function `saved_current` *shall* save the current task chain configuration `cur`. **RLS2** Schedule `interrupt Ix` *shall* have a reference only to `task chain x`. **RLS3** The service `set_interrupt` *shall* prepare the next scheduling interrupt by passing the saved task chain `cur` as second parameter.

Table 6.11: Requirements development (example)

The first level specifies a first abstract description of the intended system behavior. At the first stage, requirements development includes information hiding. Furthermore, a requirement residing in the RHI window gives a comprehensive answer to the *What?* interrogative.

The requirements specification continues at level 2 (see Figure 6.6 on the next page), with the specification process differentiating between RHS and RLI methodology. Both reside in the second level because they do not provide details on the final implementation. Internally, the main difference between the RHS and RLI is traceability. The low-level characteristic of RLI implies a traceability scheme to the RHI, which results in a corresponding effort in developing low-level platform-independent requirements. In contrast the RHS only enriches implementation-specific details on the existing RHI requirements.

The final development stage is the RLS, which refines the requirements model to its final shape. Requirements defined at RLS level specify implementation details of the system to be developed. The information described by these requirements should enable engineers to translate them directly into source code. Whereas RLS is a platform-specific refinement of RLI behavior, the transition from

Level 1	Level 2	Level 3
RHI1	**RLI1**	**RLS1**
RHI1	**RLI2**	**RLS2**
RHI1	**RLI3**	**RLS3**
RHI1	**RHS1**	**RLS1**, **RLS2**, **RLS3**

Table 6.12: Traceability matrix

RHS to RLS has to establish a traceability scheme.

Table 6.11 on the preceding page, by way of example, shows the development of requirements, with the individual development stages being listed hierarchically. Each level is marked by an RCW label put right in front of the respective requirement sentence. Certification and implementation can be based on a traceability scheme. Table 6.12 on the current page lists a traceability matrix that offers an adequate means for these development activities.

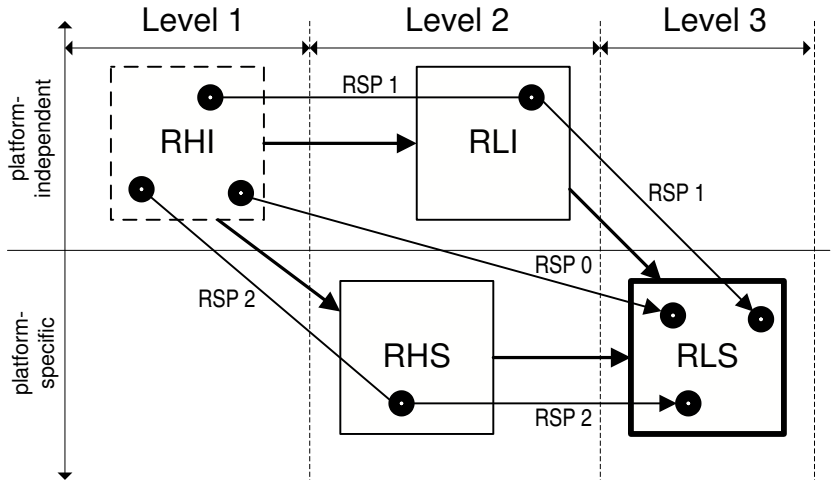


Figure 6.6: Requirements categorization windows as requirements development strategy

The example given in 6.11 also determines three possible *requirement specification paths (RSP)*:

- **RSP 0**: RHI → RLS
- **RSP 1**: RHI → RLI → RLS
- **RSP 2**: RHI → RHS → RLS

RSP 0 reflects the conventional development process of requirements following RTCA/DO-178B which neglects RCWs on Level 2.

RSP 1 integrates all considerations made on system behavior and design at the first two levels (Level 1 and Level 2). In order to finish the requirements development process, only platform-specific information about the implementation has to be added. Therefore RSP 1 is recommended for a possible reuse.

RSP 2 integrates considerations made on the concrete design of the system behavior only when the second transition (RHS $\rightarrow$ RLS) is done. Whenever the specific platform is changed, the entire effort expended to provide details on a concrete design has to be done again in the requirements specification. Thus, RSP 2 is not recommended for a possible requirements reuse.

Chapter 7

Conclusion

Reusing requirements for dependable real-time systems is a method to reduce the amount of time-consuming and costly certification activities, which are done in compliance with international safety standards.

This thesis focuses on the development of a requirements classification pattern of dependable real-time systems by combining model-driven aspects with safety-relevant guidelines used to develop requirements. In the domain of safety-critical real-time systems, requirements are specified by preventing unexpected malicious situations from occurring in the system design. Different safety approaches, however, entail different interpretations on how to arrange safety requirements. System safety defines them at system level and inherits the safety-critical impact on software requirements specified at subsequent development stages. System hazards are revealed by using the fault-tree analysis (FTA), and the hazard and operability analysis (HAZOP).

As far as the reuse of system development artifacts is concerned, the development of an information structure is followed by a classification of commonalities inherent to the proposed domain. In this thesis, commonalities can be found in the requirements development approach recommended by the guidelines of RTCA/DO-178B and the reuse-oriented concept of the model-driven architecture (MDA). Relying on the viewpoint concept of the MDA, we can divide the system behavior into a platform-independent and platform-specific part, whereas the RTCA/DO-178B approach to requirements development separates requirements into high-level and low-level requirements.

In order to provide a limited and manageable view on the reuse of requirements, with regard to the system platform determination we

extended the MDA platform by the platform layer concept. The platform layer allows viewing inside the system platform, limiting the scope of a corresponding requirements classification. System properties are assigned to each individually determined platform layer in order to decide whether the respective requirement is platform-independent or platform-specific. Based on the RTCA/DO-178B software safety guidelines, the requirements classification pattern constitutes a generic reuse classification strategy, owing to its principle of generalizing natural language requirements.

Existing certification requirements documents can serve as an ideal source for applying the requirements classification pattern. If there is no two-level documentation, as is available for TTP-OS, the requirements structure has to be developed by introducing a traceability scheme. Requirements are specified in natural language. The resulting system property elicitation, which is the basis for the assignment of properties to a platform-specific or platform-independent RCW, can be facilitated by platform layer property lists and informal guidelines.

Whenever a classification of requirements has been established, the number of requirements defined in each of the RCWs provides information about the effort to be expended to create a more reusable requirements documentation. A possible modification from platform-specific to platform-independent requirement must not change the information about the behavior specified by the requirements. On the other hand, if a requirement is changed from being platform-independent to being platform-specific, the requirements development reflects a stepwise refinement of the system behavior. These steps are already defined as requirements categorization windows and hierarchically ordered in three ways as requirement specification paths (RSP). RSP 0 reflects the conventional requirement specification following RTCA/DO-178B. RSP 1 and RSP 2 describe the final platform-specific low-level behavior of a dependable real-time system by an additional requirement development level. However, for reuse-oriented requirement development approach we recommend the use of RSP 1 because the majority of the intended system behavior resides in the platform-independent RCW even in the low-level requirement stage. Furthermore the introduction of a simple domain model has increased the reuse potential of the classified set of requirements.

In order to develop a safe real-time system, it may be essential to reuse an already safe real-time system.

Glossary

AC AC 20-148 Reusable Software Components [Fed04]

COTS Commercial off-the-shelf

DAL Design assurance level [Rad92]

Dependable system A system is dependable if it is trustworthy enough that reliance can be placed on the service it delivers [ALR01].

DER Designated engineering representative [Rad92]

E/E/PE Electrical/electronic/programmable electronic [Int98a]

Error An error is a part of the system state that may lead to a failure [Kop97].

Event-triggered system A real-time system is even-triggered (ET) if all communciation and processing activities are triggered by an event.

Fail-operational The ability of a system to continue to deliver service in degraded mode and with known safety risks after the occurrence of a failure.

Fail-safe The ability of a system to reach a safe state after the occurrence of a failure.

Failure A failure is an observable deviation from the specification [Kop97].

Fault A fault is the cause of an error.

Fault forecasting Estimating the present number, the future incidence, and the likely consequences of faults [ALR01].

Fault tolerance The ability of a functional unit to continue to perform a required function in the presence of faults or errors.

FHA Failure hazard analysis

FT-COM layer The Fault-tolerant Communication Layer is the interface between the application layer and the communication layer (the network). The FT-COM layer has tasks and messages that ensure the exchange of data between these layers [TG94].

FTA Fault tree analysis

Functional safety Part of the overall safety relating to the equipment under control (EUC) and its control system, which depends on the correct functioning of the E/E/PE safety-related systems, other technology safety-related systems and external risk reduction facilities [Int02].

GSSR Generic software safety requirement

Hard real-time system A real-time computer system that must meet at least one hard deadline.

Hazard A hazard is an undesirable condition that has the potential to cause or contribute to an accident

HAZOP Hazard and operability analysis

HLR High-level requirement

JODA JIAWG object-oriented domain analysis

LLR Low-level requirement

MC/DC Modified Condition/Decision Coverage

MDA Model-driven architecture

MISRA Motor Industry Software Reliability Association

NL Natural language

OS Operating system

PFH Probability of a dangerous failure per hour

PIR Platform-independent requirement

PIV Platform-independent viewpoint

PL Platform layer

Platform A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented [MM03].

PLPL Platform layer property list

Problem domain The part of the world where the problem solved by a piece of software resides, and in terms of which that problem is defined [Kov98].

PSA Preliminary system assessment

PSAC Plan for software sspects of certification [Rad92]

PSR Platform-specific requirement

RCP Requirement classification pattern

RCW Requirement classification window

RDL Requirement Development Level

Real-time entity A real-time entity is a state variable of relevance for the given purpose [Kop97].

Real-time image A real-time image is the current picture of an RT entity [Kop97].

Real-time object A real-time object is located on a node and contains an RT entity or RT Image [Kop97].

Real-time system A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when the results are produced.

Requirement A natural language statement describing the intended system behavior by means of system properties.

Requirements engineering Requirements engineering is the discipline concerned with understanding and documenting software requirements.

RHI Platform-independent high-level requirement

RHS Platform-specific high-level requirement

Risk Risk is the product of hazard severity and hazard probability. The severity of a hazard is the worst-case damage of a potential accident related to the hazard [B.S03].

RLI Platform-independent low-level requirement

RLS Platform-specific low-level requirement

Robustness Robustness is the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions [IEE91].

RSC Reusable software component

RSP Requirement specification path

RT Real-time

RTC Robustness test cases

RTCA Radio Technical Commission for Aeronautics

Safety Dependability with respect to the non-occurrence of dangerous failures. Measure of continuous delivery of either correct service or incorrect service after benign failure.

Safety case A safety case is a combination of a sound set of arguments supported by analytical and experimental evidence substantiating the safety of a given system [BB98].

Safety integrity The probability of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time.

Safety integrity level (SIL) Discrete level (one out of a possible five) for specifying the safety integrity requirements of the safety functions to be allocated to the E/E/PE safety-related systems, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 0 has the lowest.

Safety life-cycle Necessary activities involved in the implementation of safety-related systems, occurring during a period of time that starts at the concept phase of a project and finishes when all the safety-related systems are no longer available for use.

Safety-critical system A system where a failure can cause damage on persons, property or the environment.

Service The service that a system delivers is the behavior as it is perceived by a user.

SIL see Safety integrity level.

Soft real-time system A real-time computer system that is not concerned with any hard deadline.

Software engineering (SE) (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) the study of approaches as in (1) [IEE91]

SSA System safety assessment

SSAP System safety assessment process

SSSR Specific software safety requirements

System A system is a set of components – both computer-related and non-computer-related – that provides a service to a user.

Time-triggered protocol A communication protocol where the point in time of message transmission is derived from the progression of the global time [Kop97]

Time-triggered system A real-time computer system is time-triggered (TT) if all communication and processing activities are initiated at predetermined points in time at an a priori designated tick of a clock.

TTA Time-triggered architecture

TTP-OS Time-triggered protocol operating system

TTP/A Time-Triggered Protocol SAE class A

TTP/C Time-Triggered Protocol SAE class C

User A user is another system (for example, a human or computer) that interacts with the system at the service interface.

Bibliography

[Abb86] Russell J. Abbott. *Software Development - An integrated Approach*. Wiley-Interscience, March 1986.

[AK01] K. Allenby and T. Kelly. *Deriving safety requirements using scenarios*. In *Fifth IEEE International Symposium on Requirements Engineering*, pages 228–235, August 2001.

[ALR01] A. Avizienis, J. Laprie, and B. Randell. *Fundamental Concepts of Dependability*, 2001.

[App97] B. Appleton. *Patterns and Software: Essential Concepts and Terminology*, 1997.

[BB98] Peter Bishop and Robin Bloomfield. *A Methodology for Safety Case Development*. In F. Redmill and T. Anderson, editors, *Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium, Birmingham 1998*, pages 194–203. Springer, 1998.

[BdC91] J. Burnham and D. de Champeaux. *Object oriented (domain) analysis*. In *OOPSLA '91: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 83–90, New York, NY, USA, 1991. ACM Press.

[BH95] J.P. Bowen and M.G. Hinchey. *Seven More Myths of Formal Methods*. *IEEE Software*, 12(4):34–41, 1995.

[B.S03] B.S.Dillon. *Engineering Safety*, volume 1. World Scientific Publishing Co. Pte. Ltd., Series in Industrial and System Engineering edition, 2003.

[CW96] E.M. Clarke and J.M. Wing. *Formal methods: state of the art and future directions*. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[CY91] P. Coad and E. Yourdon. *Object-oriented analysis (2nd ed.)*. Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[Cyb98] J.L. Cybulski. *Patterns in Software Requirements Reuse*. In *Proc. 3rd Australian Conference on Requirements Engineering ACRE'98*, pages 135–153, Deakin University, Geelong, Australia, October 1998.

[Dav93] A. Davis. *Software Requirements: Objects, Functions and States*. Prentice Hall, 1993.

[Fed00] Federal Aviation Administration (FAA). *System Safety Handbook*. December 2000.

[Fed04] Federal Aviation Admistration (FAA). Advisory Circular 20-148 *Reusable Software Components*, December 2004.

[GHJV02] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: abstraction and reuse of object-oriented design*. *Software pioneers: contributions to software engineering*, pages 701–717, 2002.

[GKP98] R. Gotzhein, M. Kronenburg, and C. Peper. *Reuse in Requirements Engineering: Discovery and Application of a Real-Time Requirement Pattern*. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 65–74, London, UK, 1998. Springer-Verlag.

[Hal90] A. Hall. *Seven Myths of Formal Methods*. *IEEE Software*, 7(5):11–19, 1990.

[HED93] S.D. Harker, K.D. Eason, and J.E. Dobson. *The change and evolution of requirements as a challenge to the practice of software engineering*. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 266–272, 1993.

[HHCB06] Wolfgang Herzner, Bernhard Huber, György Csertan, and András Balogh. The DECOS Tool-Chain: Model-Based Development of Distributed Embedded Safety-Critical Real-Time Systems. *ERCIM News*, 67, Oct. 2006.

[Hil98] M. Hiller. *Software Fault Tolerance Techniques from a RealTime Systems Point of View: An Overview*. Technical Report 98-16, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, 1998.

[HKK04] B. Hardung, T. Kölzow, and A. Krüger. *Reuse of software in distributed embedded automotive systems*. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 203–210, New York, NY, USA, 2004. ACM Press.

[HR99] V. Hamilton and C. Rees. *Safety Integrity Levels: A Industrial Viewpoint*. In F. Redmill and T. Anderson, editors, *Towards System Safety - Proceedings of the Seventh Safety-critical System Symposium, Huntington, UK, 1999*, pages 111–126. Springer, 1999.

[HV01] K. J. Hayhurst and D. S. Veerhusen. *A Practical Approach To Modified Condition/Decision Coverage*. In *20th Digital Avionics Systems Conference (DASC)*, volume 1, pages 1B2/1–1B2/10, Daytona Beach, Florida, USA, October 2001.

[IEE91] IEEE. *IEEE Std 610.12-1990 - Standard Glossary of Software Engineering Terminology*, January 1991.

[IEE98] IEEE. *IEEE Std 1012-1998 - Standard for Software Verification and Validation*, March 1998.

[Int98a] International Electrotechnical Commission (IEC). IEC 61508, Part 1, *General requirements*, 1998.

[Int98b] International Electrotechnical Commission (IEC). IEC 61508, Part 3, *Software Requirements*, 1998.

[Int02] International Electrotechnical Commission (IEC). *Functional Safety in IEC 61508 - A basic Guide*, 2002.

[KB03] H. Kopetz and G. Bauer. *The Time-Triggered Architecture*. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[KCH+90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasability Study*. ESD-90-TR-222 CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, November 1990.

[KG94] H. Kopetz and G. Grünsteidl. *TTP-A Protocol for Fault-Tolerant Real-Time Systems*. *Computer*, 27(1):14–23, 1994.

[Kop93] H. Kopetz. *Should Responsive Systems be Event-Triggered or Time-Triggered? Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems*, E76-D(11):1325–1332, 1993.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Kov98] B. L. Kovitz. *Practical Software Requirments - A Manual of Content and Style*. Manning Publications Co., 1998.

[Kru92] C.W. Krueger. *Software Reuse*. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[LCS91] N.G. Leveson, S.S. Cha, and T.J Shimeall. *Safety verification of Ada programs using software fault trees*. *IEEE Software*, 8(4):48–59, 1991.

[Lev86] N.G. Leveson. *Software safety: why, what, and how*. *ACM Comput. Surv.*, 18(2):125–163, 1986.

[Lev91] N.G. Leveson. *Software safety in embedded computer systems*. *Commun. ACM*, 34(2):34–46, 1991.

[Lev95] N.G. Leveson. *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[Lev03] N.G. Leveson. *White paper on Approaches to System Engineering*, April 2003.

[Lia00] L. Liao. *From requirements to architecture: The state of the art in software architecture design*. Technical report, Department of Computer Science and Engineering, University of Washington, 2000.

[LMV97] W. Lam, J.A. McDermid, and A.J. Vickers. *Ten steps towards systematic requirements reuse*. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 6–15, Dept. of Compuer Science, York University, January 1997.

[McD02] John A. McDermid. Software hazard and safety analysis. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 23–36, London, UK, 2002. Springer-Verlag.

[MM03] J. Miller and J. Mukerji. MDA Guide 1.0.1. Object Management Group, June 2003.

[NE00] B. Nuseibeh and S. Easterbrook. *Requirements Engineering: A Roadmap*. In *ICSE - Future of SE Track*, pages 35–46, 2000.

[Par93] S. Park. *Software Requirement Text Reuse*. In *The proceedings of the Sixth Annual Workshop on Software Reuse*, Center for Software Systems Engineering George Mason University, 1993.

[PD90] R. Prieto-Díaz. *Domain Analysis: An Introduction*. *SIGSOFT Softw. Eng. Notes*, 15(2):47–54, 1990.

[PGK97] C. Peper, R. Gotzhein, and M. Kronenburg. *A Generic Approach to the Formal Specification of Requirements*. *ICFEM*, 00:252, 1997.

[PK98] S. Poledna and G. Kroiss. *The Time-Triggered Protocol TTP/C*. *Real-Time Magazine*, 4:98–102, 1998.

[Rad92] Radio Technical Commission for Aeronautics (RTCA). DO-178B/ED12 - *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[Rad00] Radio Technical Commission for Aeronautics (RTCA). DO-254/ED-80 - *Design Assurance Guidance for Airborne Electronic Hardware*, April 2000.

[Rus91] J. Rushby. *Measures and Techniques for Software Quality Assurance*. Technical report, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, 1991.

[Rus93] J. Rushby. *Formal Methods and the Certification of Critical Systems*. Technical report, Computer Science Laboratory, SRI International, Menlo Park CA 94925, USA, December 1993.

[Sch01] Schlatterbeck, R. and Elmenreich, W. *TTP/A: A Low Cost Highly Efficient Time-Triggered Fieldbus Architecture*. In *Proceedings of the SAE World Congress 2001, March 2001, Detroit, Michigan, USA*, March 2001.

[Soc96a] Society of Automotive Engineers (SAE). ARP 4754 - *Certification considerations for Highly Integrated or Complex Aircraft Systems*, 1996.

[Soc96b] Society of Automotive Engineers (SAE). ARP 4761 - *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 1996.

[Sof98] Software Engineering Standards Committee of the IEEE Computer Society. *IEEE Guide for Developing System Requirements Specifications*, December 1998.

[Spi00] C.R. Spitzer. *The Avionics Handbook*. CRC Press, December 2000.

[Sto96] N. Storey. *Safety-Critical Computer Systems*. Pearson, 1996.

[Sur94] N. Suri. *Advances in ULTRA-Dependable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[TG94] C. Tanzer and M. Glueck. *TTP-Os - The Time-Triggered and Fault-Tolerant RTOS*. *Real-Time Magazine*, 4:61–64, 1994.

[The98] The Motor Industry Software Reliability Association (MISRA). *Guidelines For The Use Of The C Language In Vehicle Based Software*, April 1998.

[YZ80] R.T. Yeh and P. Zave. *Specifying Software Requirements*. In *Proceedings of the IEEE*, volume 68, pages 1077–1085, 1980.

[ZG02] D. Zowghi and V. Gervasi. *The Three Cs of Requirements: Consistency, Completeness, and Correctness*, 2002.

Index