Prof. Dr.-Ing. A. Siggelkow

Requirements and Specification

Hochschule Ravensburg-Weingarten
University of Applied Sciences

Contents

Chapter 1

Requirement and Specification

The design of CPS and IoT systems is a complex process. In principle it is the well known divide and conquer approach, like in the classical design flow. It has tobe broken down into sub-tasks. Every sub-task must have a well defined interface to the other.

The starting point of a system design is the knowledge in the heads some people, at least the customer and/or system architects. Those ideas must be written down into a **requirements sheet**, this requirements sheet will be signed by the customer and the design company. After the requirements analysis has been finalized, the **specification** will be derived on base of the requirements.

1.1 Requirements

There is no common definition of "requirements". There is a requirements analysis according [8]:

- IEEE
- CMMI (Capability Maturity Model Integration)
- Volere
- IIBA (International Institute of Business Analysis)
- and may be more

Because of the electrical and computer science nature of ES, CPS and IoT, I describe the IEEE approach. But nevertheless, in principle, all approaches are similar.

According IEEE, requirements engineering consists of:

Requirements elicitation: Collect all possible requirements of the system.

Requirements analysis: The analysis of the whole set of requirements should lead to a common understanding of the product.

Requirements specification: The requirements will be documented and explained - specified.

Requirements validation: The whole set of requirements must be checked against each other to detect holes or conflicts in the description.

The **elicitation and analysis** of the requirements must follow some criteria:

Complete: The requirements must be described explicit. It should not be the case, that the customer has implicit assumptions of the system.

Clearly defined/bound: Precise definitions help to avoid misunderstandings between the customer and the developer.

Comprehensively described: The customer and the developer should be able to read the requirements with the same understanding.

Atomic: There should be only one requirement described within one section.

Identifiable: Every requirement needs its own ID.

Uniformly documented: The requirements should be documented with the same structure in just one database.

Verifiable: Every requirement needs **acceptance criteria** with which the product can be tested against the requirements. **Test cases** will be derived from the acceptance criteria.

Traceable: It must be traceable, that a requirement has been fulfilled (forwards). Also must be verified, that every implemented functionality has its requirement (backwards).

Consistent: The requirements are not allowed to contradict each other.

After the elicitation, the requirements must be **structured and classified**. This increases the clarity. And this increases the understanding of the relations between the requirements. Criteria for this are:

Dependent: Requirements must be tested if one requirements depends on another, or will be independent.

Belonging together: Requirements, which belongs logically or technically to an other requirement, should be grouped.

Role-based: Every user group will have its own sight on the requirements.

Functional requirements: Those explains, what the system should be able to do. Requirements for the function of the system.

Non-functional requirements: Those explains, how the system should bring ist function (Quality, performance, reliability). Requirements which handles standardization, legal, social, medical, etc. aspects.

Next, the **quality assurance** of the requirements needs to be done:

Correct: The requirements must be consistent.

Feasible: The requirements must be feasible.

Necessary: Requirements not required by the customer are no requirements.

Prioritized: Identify the importance of the requirements for the product.

Usable: Even having just a port of the system, it should be a productive system.

Important aspects for the requirements of small projects planned with students are:

Unitary: The requirement addresses one and only one thing.

Complete: The requirement is fully stated in one place with no missing information.

Consistent: The requirement does not contradict any other requirement and is fully consistent with all authoritative external documentation.

Atomic: The requirement is atomic, i.e., it does not contain conjunctions. E.g., "The postal code field must validate American and Canadian postal codes" should be written as two separate requirements: (1) "The postal code field must validate American postal codes" and (2) "The postal code field must validate Canadian postal codes".

Unambiguous: The requirement is concisely stated without recourse to technical jargon, acronyms (unless defined elsewhere in the Requirements document), or other esoteric verbiage. It expresses objective facts, not subjective opinions. It is subject to one and only one interpretation. Vague subjects, adjectives, prepositions, verbs and subjective phrases are avoided. Negative statements and compound statements are avoided.

Specify Importance (Prioritized): Many requirements represent a stakeholder-defined characteristic the absence of which will result in a major or even fatal deficiency. Others represent features that may be implemented if time and budget permits. The requirement must specify a level of importance.

Verifiable: The implementation of the requirement can be determined through basic possible methods: inspection, demonstration, test (instrumented) or analysis (to include validated modeling & simulation, a Test-Bench element is required for every requirement).

Identifiable: Every requirement needs its own ID.

1.2 Specification and Modeling

1.2.1 Requirements (for the Specification)

Specifications for ES provide **models** of the system under design. A good definition is:

Definition 1.2.1. [9]: "A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task".

Models are described in languages, the languages should have some features:

Hierarchy: Physical objects are normally too complex to be understood by human beings. Hierarchy and abstraction are key mechanisms helping to solve this dilemma. There are:

- **Behavioral hierarchies:** Behavioral hierarchies contain objects necessary to describe the system behavior. Such objects are: states, events, output signals and more.
- **Structural hierarchies:** Structural hierarchies describe how systems are composed of physical components. Such components are: processors, memories, actuators, sensors. Processors have: registers, multiplexers, adders. Multiplexers have: gates. Gates have: transistors.

Component-based design: If a system is build of single components with a known behavior, it must be possible to predict the behavior of the system.

Concurrency: Real systems operate concurrently, so the system model needs to reflect this concurrency.

Synchronization and communication: Components must be able to communicate and synchronize.

Timing behavior: Many ES are real-time systems. The timing requirements must be specified. Unfortunately, time is not a simple quantity to be modeled. But some aspects must be considered:

- Measure the **elapsed time** (timer).
- **Delay a process** for a specified time (control over interrupts).
- Specify **timeouts** (again, interrupts are a problem)
- Specify **deadlines** and **schedules** (interrupts, caches, pipelines).

State-oriented behavior: FSM are a good mechanism to model reactive systems. The classical FSM needs to updated, to be able to support model timing and hierarchy.

Event handling: Reactive systems are event driven.

Exception-oriented behavior: Exceptions must be considered, they will be needed but there are problems with it.

Presence of programming elements: The specification of programming elements needs to be merged into the specification of systems.

Executability: Specifications could differ from the ideas in people's heads, so an executable specification has its advantages.

Support for the design of large systems: In SW design, object oriented programming is a means to handle large systems. This is needed in the specification of large HW/SW systems, also.

Domain-specific support: Different application domains need different aspects in the specification.

Readability: People (customer) needs to read and understand the specification, on the other hand, the specification should be executable which means, it must be computer readable. There must be a way, to test the written specification against the executable specification.

Portability and flexibility: The specification should be independent of the HW platform.

Termination: Termination processes needs to be specified.

Support for non-standard I/O devices: It is self explaining.

Non-functional properties: Fault tolerance, size, power consumption, etc.

Support for the design of dependable systems: It is self explaining.

No obstacles to the generation of efficient implementations: It is self explaining.

Appropriate model of computation (MoC): The von-Neumann model of sequential computation has problems with respect to concurrent CPS (timing, multithreading and deadlocks, etc.).

1.2.2 Models of Computation

A system is build of many components. In general, **computation** and **communication** will be differentiated.

- Components: ... and the organization of computations in such components: Procedures, processes, functions, finite state machines, etc.
- Communication protocols: Methods for communication between components (asynchronous message passing, rendezvous based, etc.).

Components:

The relation between components may be visualized by means of **graphs** (process or task). Nodes in the graph (task graphs or process networks) represent components performing computations. Edges represent relations between components. E.g., one relation is the causality, this results in a **dependence graph** (fig. 1.1).

Definition 1.2.2. [5]: A dependence graph is a directed graph $G = (\tau, E)$ where τ is the set of **vertices** or **nodes** and E is the set of **edges**. $E \subseteq \tau \times \tau$ imposes a relation on τ.
If $(\tau_1, \tau_2) \in E$ with $\tau_1, \tau_2 \in \tau$, then τ_1 is called an **immediate predecessor** of τ_2 and τ_2 is called an **immediate successor** of τ_1. Suppose E^* is the transitive closure of E.
If $(\tau_1, \tau_2) \in E^*$, then τ_1 is called a **predecessor** of τ_2 and τ_2 is called a **successor** of τ_1.

A dependence graph is a special form of a **task graph**. Tasks graphs allow more information:

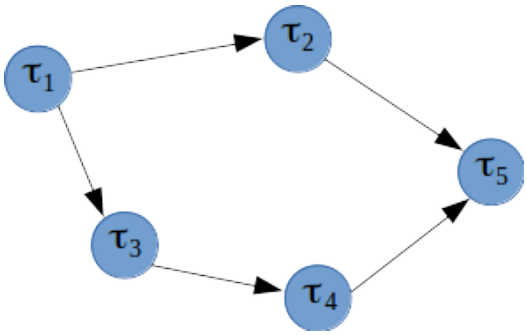

Figure 1.1: Dependence Graph

1. **Timing information:** Arrival times, deadlines, periods, execution times, etc.

2. **Distinction between different types of relations** between computations: Precedence relations, communication described by the edges, I/O nodes (fig. 1.2, partially filled circles are I/O edges).

3. **Exclusive access to resources:** Some computations may require exclusive access to to resources, e.g., I/O, memory space for communication.

4. **Periodic schedules:** Distinguish between a task and ist execution (**job**), such graphs used in signal-processing can be periodic and also infinite (fig. 1.3).

5. **Hierarchical graph nodes:** The granularity depends on the application, but if hierarchy is needed, it can also be shown in the graph (fig. 1.4).

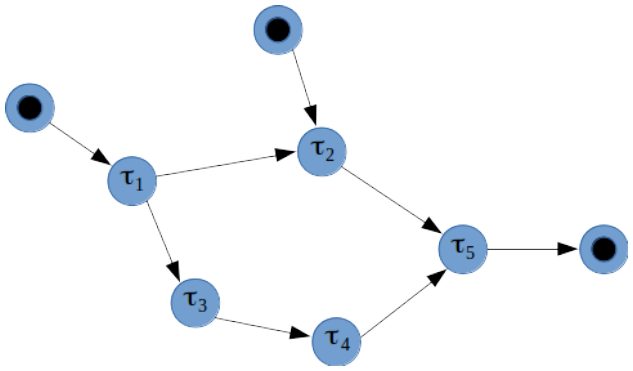

Figure 1.2: Graph including I/O nodes and edges

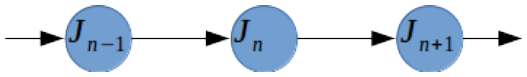

Figure 1.3: Graph including jobs

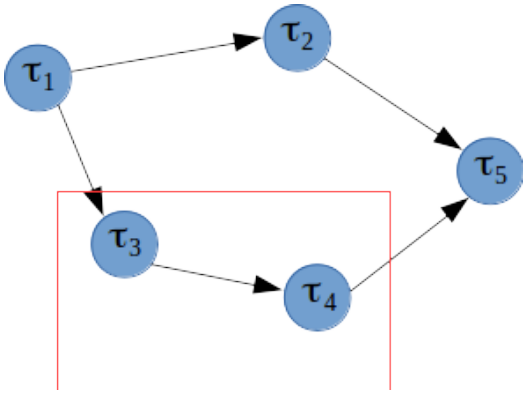

Figure 1.4: Hierarchical task graph

Communication (and computation):

Communication will be reflected by the edges in the task graphs. And the computation in the nodes.

- **Models of communication:** Two different models: **shared memory** and **message passing**.
  - Shared memory: Components can access the same memory. One needs locked-writes, semaphores, conditional critical regions, monitors, spin locks.
  - Message passing: Messages are sent and received (slower than shared memory). There is: asynchronous message passing (non-blocking communication), synchronous message passing (blocking communication, rendezvous based communication), extended rendezvous (remote invocation).
- **Organization of computations within the components:** Differential equations, FSMs, data flow, discrete event model, Von-Neumann model.
- **Combined models:** Actual languages combine some models of communication.

StateCharts: StateMate, StateFlow, BetterState

1.2.3 Early Design Phases

The first ideas could be captured in a very informal way, also natural language.

Use Cases

Use cases (e.g. UML Use Cases; KDE Umbrello) deliver a good understanding of the system in the very early design phase. It shows what the contributors of the system are, what they can do and which functions are needed. It will not show the sequence of the functions.

Example: Figure 1.5 is a simplified use case diagram for an answering machine.

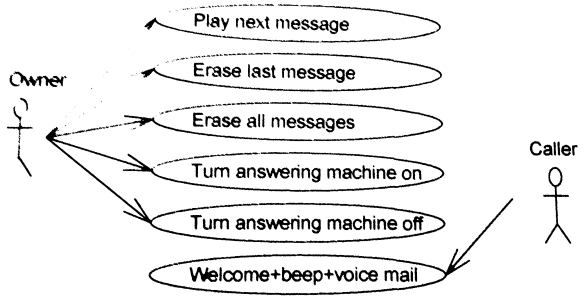

Figure 1.5: Use case example

(Message) Sequence Charts and Time/Distance Diagrams

Here, the sequence and timing of the messages could be seen, it is just another perspective of the same story. Example of a Sequence Chart (SC), figure.

In SC, it is not planned to transport timing, for this, time/distance diagrams (TDD) will be used [5].

Example: Figure 1.6 shows an example.

Differential Equations

Differential equations are needed to model the real world environment of CPS.

1.2.4 Communicating Finite State Machines (CFSMs)

The importance of Finite State Machines (FSM) has already been mentioned. FSMs, where only one state at a time is active, will be called **deterministic** FSMs.

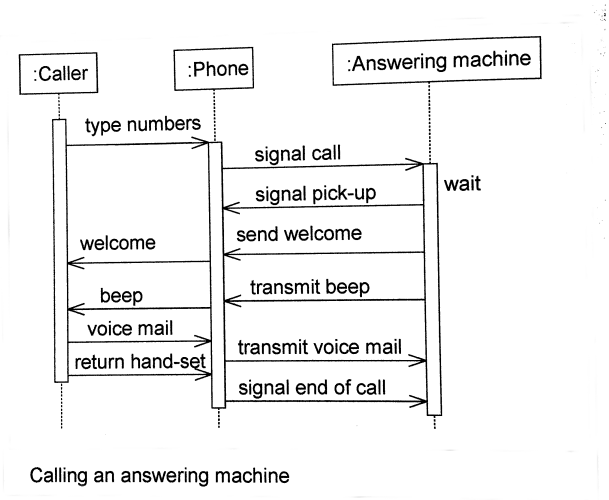

Figure 1.6: Answering machine in UML

FSMs, which are clocked, are called **synchronous** FSMs. FSMs can be described by state diagrams, fig. 1.7.

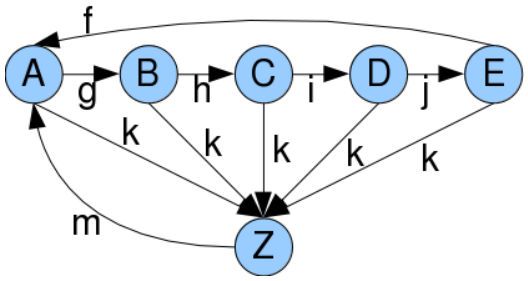

Figure 1.7: State diagram

1.2.5 Executable Specification

see SystemC

Chapter 2

Literature

Bibliography

[1] https://www.tuev-nord.de/explore/en/explains/whats-the-difference-between-safety-and-security/ (8.8.2017)

[2] Swarup Bhunia, Sandip Ray, Susmita Sur-Kolay (Editors). *Fundamentals of IP and SoC Security: Design, Verification, and Debug*. Cham, Switzerland, Springer, 2017, ISBN 978-3-319-50055-3

[3] https://en.wikipedia.org/wiki/Verification_and_validation

[4] Lee, E.A.: Computing foundations and practice for cyber-physical systems: a preliminary report. Tech. Rep. UCB/EECS-2007-72, EECS Department, University of California, Berkeley (2007). http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-72.html

[5] Marwedel, P. (2018) Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things, Third Edition, Springer, Berlin

[6] Marwedel, P. (2003) Embedded System Design, Kluver Academic Publishers, Norwell

[7] Guisto, D., Iera, A., Morabito, G., Atzori, L. (eds) (2010) The Internet of Things, 20th Tyrrhenian Workshop on Digital Communications, Springer, Berlin

[8] https://de.wikipedia.org/wiki/Anforderungsanalyse_(Informatik), 13. Sept. 2017

[9] Jantsch, A. (2004) Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation, Morgan Kaufmann, San Francisco

[10] Kao, C.-F., Chen, H.-M., Huang, I.-J. (2008): Hardware-Software Approaches to In-Circuit Emulation for Embedded Processors. IEEE Design and Test, 25 (5): 462 - 477

[11] Syed Farid Syed Adnan, Mohd Anuar Mat Isa, Khairul Syazwan Ali Rahman, Mohd Hanif Muhamad, Habibah Hashim (2015): Simulation of RSA and ElGamal Encryption Schemes using RF Simulator. IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE), 2015 IEEE

[12] A. Lopez, K. Vatanparvar, A. Prasad, S. Yang, S. Bhunia, M. A. Al Faruque. "A Security Perspective on the Battery Systems of the Internet of Things". *Springer Journal of Hardware and Systems Security (Springer HASS 17)*, vol. 1, pp. 188-199, 2017.

[13] S. Chhetri, N. Rashid, S. Faezi, M. A. Al Faruque. "Security Trends and Advances in Manufacturing Systems in the Era of Industry 4.0," ACM/IEEE International Conference on Computer-Aided Design (ICCAD 17), Irvine, CA, November, 2017

[14] Heise: https://www.heise.de/security/meldung/Forscher-demontieren-kaputten-Krypto-Standard-P1735-der-IEEE-3883590.html

[15] Animesh Chhotaray, Adib Nahiyan, Thomas Shrimpton, Domenic Forte, Mark Tehranipoor, Standardizing Bad Cryptographic Practice, CCS '17, October 30-November 3, 2017, Dallas

[16] Alexander de Graf, SystemC: an overview, EEMCS/ME/CAS, study script, 10-5-2016, TU Delft

[17] David C. Black, Jack Donovan, Bill Bunton, Anna Keist, Systemc: From the Ground Up, Second Edition, Springer, 2010

[18] Matthias Menge (2005) Moderne Prozessorarchitekturen: Prinzipien und ihre Realisierungen, Springer

[19] David A. Patterson, John L. Hennessy (2018) Computer Organization and Design: The Hardware/Software Interface, Risc-V Edition, Morgan Kaufmann Publishers

[20] John L. Hennessy, David A. Patterson (2019) Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers

[21] Christian Siemers (2014), Embedded Systems Engineering 1, Skript, TU Clausthal

[22] Christoph Krauß, Cyber-Physical Systems: Sicherheit im Internet der Dinge, https://www.sit.fraunhofer.de/de/cyberphysicalsystems/, 02.04.2019