Requirements Engineering for Embedded Systems[1)]

Manfred Broy
Technische Universität München, Institut für Informatik
80290 München
e-mail: broy@informatik.tu-muenchen.de
FAX: +49 89 / 289 28183

Keywords: Requirements engineering, specification, dependability, real-time, safety critical

Abstract

In requirements engineering for embedded systems the fundamental functional and non-functional requirements for a system with an embedded software system are discussed, captured, analysed, validated, and documented. We suggest a stepwise proceeding. We advise to describe the requirements in terms of different system views obtained from an abstract logical architecture of an embedded system. By the logical architecture a system and its context are decomposed into four components. We discuss the nature of these components and the need and techniques to model their behaviours.

1. Introduction

The general purpose of an embedded hardware/software system is to regulate a physical device by sending control signals to actuators in reaction to input signals provided by its users and by sensors capturing the relevant state parameters of the system. We therefore call an embedded system also a *reactive system*. The control is achieved by processing information coming from sensors and user interfaces and controlling some actuators that regulate the physical devices.

The reliability of a system with an embedded software system depends on well-actuated reactions according to the users' expectations, even in exceptional situations. This reaction of the overall system is to a critical extend determined by the embedded software system that is a part of the overall system. The task of the embedded system is to guarantee, as far as the physical parts function properly, the adequate behaviour of the overall system. Therefore the requirements for the embedded system have to be captured and formulated in terms of the behaviour of the overall system.

In the development of embedded reactive systems the specification of the required functionality is a most critical issue. Statistics show that in typical application areas more than

[1)] This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", by the BMBF project KorSys, and the industrial research project SysLab.

50 % of the problems (malfunctions) that occur within delivered systems and that are reported by the customers are not problems with the correctness of the implementation but with misconceptions in capturing the requirements (called *conceptual requirements errors*).

Embedded systems - especially when running in risk critical applications - demand a high degree of reliability. Basis of the reliability of a system is the careful capturing of the adequate user requirements.

2. The Requirements Engineering Process

The requirements engineering process (see [IEEE 94]) should help

- the customer to accurately describe what they wish to obtain,
- the supplier to understand exactly what the customer wants.

Typically, there are different groups of people involved in the requirements engineering process. These are users, application experts, customers, marketing experts, project managers, electrical engineers, hardware and software engineers. To these groups of people the requirements engineering process should provide several benefits:

- establish a basis of agreement,
- provide a basis of project planning,
- reduce the development effort,
- provide a basis for cost estimation and schedules,
- provide a starting point for the design and implementation,
- provide a baseline for validation and verification,
- facilitate usability,
- form a basis for enhancement.

We distinguish two overlapping sub-phases in the requirements engineering process

(1) pre-phase: elaboration of the general product strategy and positioning,

(2) main phase: technical requirements engineering.

In the pre-phase (1) the overall goals and marketing issues of the planned product are determined. On this basis a first informal rough requirements document is written. It may contain a number of alternatives and list given constraints.

Based on these results, in the main phase (2) the more technical requirements engineering is carried out. Its goal is to produce an agreement and understanding in and between the groups involved with respect to the requirements. Finally, these should be completely and precisely documented in the technical requirements specification. We mainly concentrate on the main phase (2) in the following.

The requirements engineering phase is followed by the system design phase. Often it is not so easy to get a clear cut between these two phases. In an incremental development process the requirements may be revised after the design and a first experimental implementation phase. Therefore changeability and adaptability are important issues for requirements specifications. Moreover, breaking down the system into subcomponents in the design asks for a specification of the component requirements and often a mini requirements engineering process.

3. The Tasks in the Requirements Development Process

The activities of requirements engineering form an important task in the development process. As mentioned above typically requirements engineering uses the result of a first system analysis phase that provides a general understanding and leads to a specific modelling of the application domain. Based on these results the main phase of the requirements engineering includes the following steps:

- domain analysis and domain modelling,
- requirements capture,
- requirements validation,
- requirements tracing,
- requirements verification.

The first three steps are mainly part of the analysis phase. The second two steps are part of the design, implementation, integration, and test phases.

3.1 Requirements Capture: User Participation and User Adequacy

Requirements engineering comprises the complex of activities that are carried out to analyse, to work out, formulate, and document the overall requirements for a system. In general, at the beginning the groups of people representing the user, the customer, and the supplier have only vague ideas about the requirements of a system. Only in a careful process of sorting out and capturing the requirements the customer and supplier may become aware of a number of aspects and requirements that were only implicit so far and develop a common understanding and agreement. This process needs a well-planned user participation to ensure the user adequacy of the results of the requirements engineering.

The result of the requirements engineering process are a number of documents that describe the requirements in an informal, semiformal, or formal way. The requirements can be classified into *functional* and *non-functional* requirements. Non-functional requirements impose constraints on the system requirements. We distinguish between functional requirements, design constraints, external interface requirements, and quality attributes. Non-functional requirements may include quite unrelated aspects such as constraints on hardware and software solutions, as well as performance. They may also include requirements for the cost, development time, reliability, portability, maintainability, and security of the system. The non-functional requirements influence the functional requirements. Therefore, also in documents capturing the functional requirements it is useful to refer the relevant non-functional requirements and to give rationales for decisions taken in the requirements engineering process. The documentation of non-functional requirements is a valuable basis for decisions that have to be taken when capturing and implementing the functional requirements. We mainly concentrate on functional requirements in the following.

In requirements engineering we describe the required behaviour of an embedded system on an abstract, logical level. One of the decisive tasks is the selection of a good domain and system model to express the requirements. It should be as simple, abstract, and suggestive as possible. Moreover we need specification techniques to represent the model.

A very helpful technique in capturing function requirements are scenarios or use cases. There typical cases of interactions between the user, the system, and the environment are described. We recommend to work with a number of scenarios (use cases) and views of the system usage that describe typical functions and services of a system. There we should define as far as relevant for the system:

- the user processes at the interface of the system,
- its modes of operation,
- critical situations and conditions,
- forms of operations and the user model.

Use cases are an excellent means to let the use participate in the requirements capture process. The goal of the requirements engineering process is not only to produce a couple of documents that describe the requirements, its goal is also to achieve a common system understanding of the development team at an abstract, logical, implementation independent level.

Often it is useful to distinguish between exceptional and non-exceptional cases in requirements: In general, it is advisable to work in requirements engineering only with non-exceptional cases to begin with and then add the treatment of exceptional cases in a systematic way.

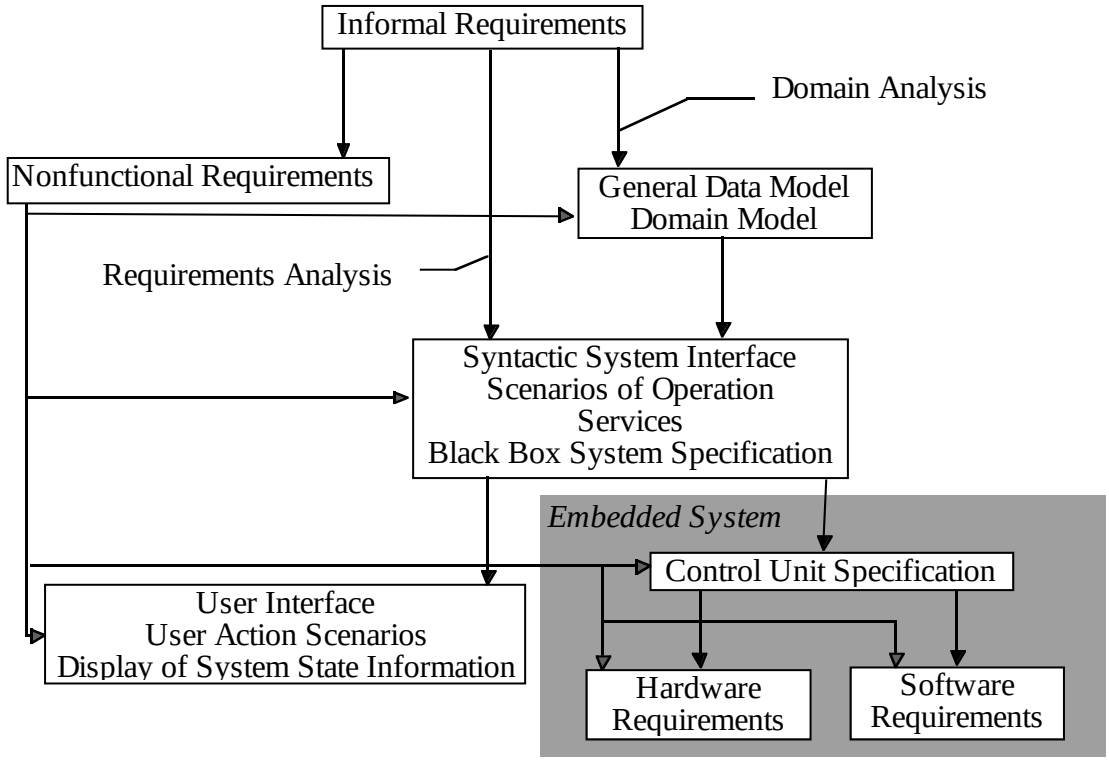

Fig. 1 Information Flow in the Requirements Engineering Process

In section 5 we describe a general logical structure called the *reference architecture* of an embedded system. It is intended as the reference structure (also called the *logical architecture*) for the development of embedded systems.

3.2 Requirements Validation

Not only implementations contain bugs, but also requirements documents may contain misconceptions or inconsistencies. Therefore it is important to do a careful analysis and review of the requirements after they have been captured and documented. The activity that checks the requirements documents to make sure that they are properly structured and capture the users' intentions is called *requirements validation*.

In requirements analysis and validation we work with the following techniques

- requirements reviews,
- structured walk throughs,
- tool based consistency checks,
- verification of safety critical properties,
- prototyping and simulation.

All of these techniques can be supported by tools. Verification can be done by mathematical proof techniques, deduction based support tools, or by model checkers. It is a well-known fact that bugs in the requirements are more costly to fix the later they are found. Therefore validation efforts should start very early.

3.3 Requirements Tracing

In requirements tracing we make sure that the requirements are probably taken care of in the design and implementation steps. The most effective way of requirements tracing is achieved by explicitly taking the requirements into account in each of the design and implementation steps as well as documenting how the requirements are ensured. The consequent way to do this is a stepwise development of the design and implementation and its correctness proof by refinement steps departing from the requirements specification.

Other possibilities are to do the requirements tracing after the development steps have been carried out as part of quality assurance. In any way it is advisable to think about the steps needed to ensure the requirements by requirements tracing already when formulating the requirements specification. There the question how these requirements can be traced in the various steps of the development process should be answered.

3.4 Requirements Verification

Finally, we have to make sure that the requirements, at least the safety-critical requirements, are met by the implementation. The ultimate technique is formal verification by logical deduction or model-checking. If all interfaces and system parts are formally specified we may prove the correctness of the designed and implemented system with respect to the captured requirements. This correctness depends of course strictly on the validity of the models used to map the requirements. Gaining a fully formal verification using a proof support system is a difficult and costly task and may not be realistic, today. Other options are testing, structured walk throughs and reviews of the system code. A very helpful concept is to fix the ways requirements are to be

tested or verified already in the requirements specification phase. The production of a requirements verification plan as part of the requirements engineering is very helpful.

Once we have formalised the specification, we can formulate the basic proof obligations. An embedded system is called *correct*, if the four views described below are consistent. This means that the behaviour generated by the controller in the interaction with the user interface and the environment meets the required system reactions and observations.

4. Structuring Requirements by System Views

A complex information processing system embedded within a technical system cannot be described in a monolithic way. For comprehensibility, it is better to structure its description into a number of complementing views. In this section we give a brief overview over the particular aspects of such systems and techniques for their description. They include:

- data models: data type declarations, abstract data types, entity relationship diagrams,
- process models: event traces, message sequence charts, process diagrams,
- system component models (interface models, black box view): syntactic interface diagrams, input/output relations,
- distributed system models: data flow diagrams,
- state transition models: state transition diagrams.

The integration of these description techniques is an indispensable prerequisite for more advanced system and software requirements, their specification and description methods as well as support tools. In the requirements development process the various description methods serve mainly the following purposes:

- as a means for the analysis and requirements capture for the developer,
- basis for communication and agreement between the engineers and the users and application experts,
- documentation of development results and input for further development steps,
- as a basis for the testing and validation of development results.

Of course, the usefulness of a description formalism has always to be evaluated with respect to these goals.

Typically embedded software systems are time critical. The responses to sensor signals have to be generated within certain time bounds to achieve the required behaviour. To capture such time restrictions is an important part of the requirements engineering task. For many description techniques available today in practice the time models are not fully formalised and/or show some deficiencies.

Whether we are able to work out a complete logical requirements specification depends crucially on the type of system that we deal with. For certain systems we may give a comprehensive description of the causality between the user actions, the events in the environment, and the reactions of the system. For other systems we may only be able to give general informal requirements (like "optimise the energy consumption"). For certain systems with fuzzy requirements an extensive experimentation with prototypes may be necessary to sort out the requirements.

A special case are hybrid systems where parts of the system are modelled by discrete event systems and other parts are time or message continuous. Then mixed "hybrid" description formalisms have to be used.

5. Logical Architecture of Embedded Systems

An embedded system typically can be decomposed into of the following five constituents:

- the controlling device (control unit, CD, the embedded hardware/software system),
- the controlled physical device (PD),
- the user interface (UI),
- the environment (EV),
- the user.

The overall architecture of an embedded system consists of these constituents and describes how they are connected and interact. The overall architecture can be depicted as a data flow diagram as shown in Fig 2. The arrows correspond to communication channels. The physical device may be connected to the environment in a sophisticated way that is very difficult to formalise.

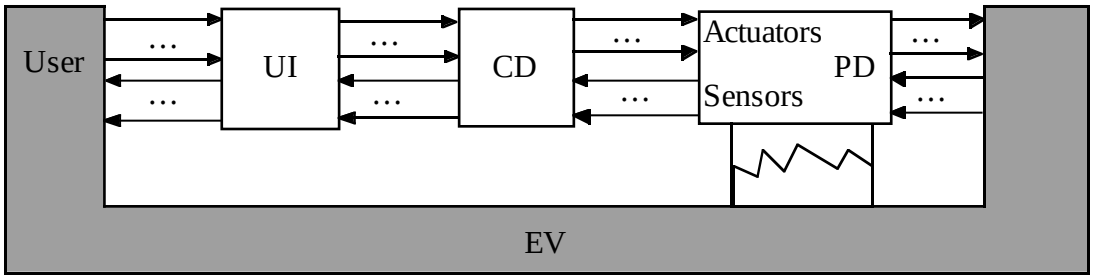

Fig. 2 Logical Reference Architecture of an Embedded System as Data Flow Diagram

Each arrow stands for an input or an output channel. These arrows represent channels on which streams of discrete messages or continuous signals are exchanged. Note that the control device is not connected to the environment but only to the user interface and the physical device. Note, moreover, that we have seen the user and the environment of the physical device as a more vague entity where the connection between the user and the environment is left unspecified. We also have indicated that the interaction between the physical device and its environment can be rather sophisticated such that it cannot easily be described by the concept of a discrete message exchange over channels.

Often the communication links between the different components of a system need particular care. They may work with special communication links and special protocols. These aspects, however, are rather a part of the design than of the requirements specification, in general.

5.1 The Domain Model

As a basis for the requirements capture we have to understand the basic theories and properties of the application domain. These have to be captured into a general domain model that contains all the basic data structures and functions needed to describe the signals and messages that are exchanged by the system as well as the functions and properties that are needed as a basis to describe the system parts and states as shown in the system reference architecture. To describe the domain model we typically apply techniques from data modelling such as type declarations, abstract data types, or entity/relationship diagrams.

5.2 Observations: the Overall Behaviour of the System

The correctness (or better the usefulness) of an embedded system has to be determined by observing the behaviour (reactions to input by the user and the environment) of the overall system. More precisely, we do not want to observe the behaviour of the controller, in general, but of the physical device. If it shows some intolerable behaviour we do not accept the design. Of course, observations need contain not only the information about the events initiated by the physical device but also the input of the user and the environment. In other words, we have to consider the user input together with the observed behaviour of the physical device to be able to classify what behaviour we require. There may be the following types of reasons for an incorrect and unexpected behaviour:

- the assumptions about the behaviour of the physical device are incorrect,
- the controller does not emit the appropriate signals in time,
- the user interface does not convey the right information between the user and the controller.

As pointed out in connection with the treatment of the physical device it may be very complicated to capture formally the behaviour of the physical device in its interaction with the environment. A typical example is the motor management in cars. In such cases we have to deal often with a very restricted artificial notion of observation that captures only parts of the required behaviour or we have to deal with complex mathematical models. Aspects that are difficult to express by logical and mathematical means have to be captured with the help of prototyping and experimentation.

5.3 The Controlling Device (CD)

The controlling device receives signals from the sensors of the physical device and from the user interface. These inputs are processed and in return signals are sent back to the user interface and to the actuators of the physical device. The controlling device is certainly the core of the embedded system.

Of course, the controlling device need not be a centralised component but may be decomposed into a number of subcomponents that are distributed and interact in parallel with the user interface and the physical device. However, questions of parallelism and distribution of the CD are part of the glass box views on the controller. We are mainly interested in black box

views in requirements specifications[1]). Even if there is a direct connection between the user interface and the physical device this can be modelled as a special case of the CD as long as all specifications provide appropriate black box views for the component of the distributed system.

5.4 The Physical Device (PD)

The physical device is a heterogeneous unit, in general, which may consist of electronic, electrical, and mechanical parts. We consider also the sensors and actuators as part of the physical device. Therefore the physical device can be - at least viewed from the perspective of the control unit - seen as a component the operates with discrete or continuous message streams that gets signals from the controller as input and produces signals by its sensors as output.

In addition, the physical device is usually in some interaction with its environment. Sometimes it may not be so obvious where to cut best between the physical device and its environment. This is a crucial decision of the systems engineer. Once decided, we have to model the interaction between the environment of the system, and the physical device.

Of course, this can be very difficult in certain cases, since this interaction may consist of complicated physical (mechanical or electrical) or chemical processes with very complex causal dependencies. Therefore it may be very difficult to find a good model for representing the output of the physical device to the environment and vice versa. Of course, we can also model analog output and input (as a continuous function depending on time). However, even with this technique often the interaction cannot be properly modelled.

We therefore make a simplifying assumption. We assume that we can capture the interaction with the physical device and the environment by a number of time discrete or time continuous message streams. The discrete messages are used to encode events. In continuous streams we may encode continuously changing parameters of the system state. The differences in the reactions of the environment can be modelled by nondeterminism or by predicates that describe the set of expected behaviours.

In cases where such a simple model does not work we may choose the physical device much smaller or in the extreme case study only the interaction between the controller and the physical device. Another possibility is to choose the physical device much larger and to incorporate a large part of the environment into it. This may lead to an interface description for the physical device that is easier to deal with. In certain situations it may be necessary to work with probabilities and to express system requirements in terms of probabilities.

5.5 The User Interface

Most embedded systems do not only interact with the physical systems that they control but also take input from users and produce output to the users to inform them about the status of the physical device. This is the function of the user interface.

Like the physical device itself we cannot view the user interface as a pure hardware/software system since in general it consists of a number of devices for receiving input (like buttons,

---

[1]) Of course, the distribution of the controller, for instance for safety reasons, can be part of the requirements; this is a typical example where requirements constrain the design.

keyboards, joysticks, etc.) and for displaying output (like signal lights, screens, etc.). Often there are parts of the physical system that can either be seen as part of the user interface or of the controlled physical device. Again the structuring has to be decided on by the system engineer. The user interface can again be decomposed into a number of distributed user stations.

6. The Semantic Views

According to the structure we have described in the previous section we have to specify the following views on an embedded system to deal with proper behaviours:

- the behaviours (observations) showing the causality between user actions and the interactions of the physical device with respect to its environment,
- the behaviour of the physical device showing the causality between the control signals send by the controller and the sensor signals and the observable actions of the physical device,
- the behaviour of the controller consisting of the causal relationship between the input from the user interface and the sensors and the output to the user interface and to the actuators,
- the user interface consisting of the causal relationship between the input from users and the controller and the output to the user and the controller.

All these views of an embedded system have to be formally specified in a proper development process. They comprise all the requirements for the overall system and its parts.

7. Description Techniques

As we described above the requirements capture introduces several views. These views mainly describe parts of the system as components. The modelling of the components of an embedded system, in general, should consist of the following descriptions:

- the syntactic interface,
- the typical processes of interaction (event traces),
- the individual component behaviour.

The syntactic interface is described by the actions that connect a component with its environment. Such actions may be discrete such as

- receiving or sending messages and signals,
- reading or writing attributes (variables).

We may also work with continuous signals and continuous state variables, of course. Important, in general, is the timing of the actions.

The process of interaction may be described by message sequence charts or process diagrams. By such techniques we represent use cases which are typical runs of the system showing the causal relationship of the individual actions of the components and the environment. It strongly depends on the type of system whether we try to give a complete set of use cases or only selected class of instances.

The individual component behaviour can either be described by input/output relations or by state machines. Input/output relations describe the causality between the messages in the streams of input and the messages in the streams of output actions without providing a concrete model for the state of a component. State machines describe the behaviour of a component in terms of its state and state changes in relation to time and input/output actions.

8. Conclusion

Embedded systems are an important area of application of reactive systems. Their proper formal and methodological treatment is of major importance. For a systematic development standards are needed such as process models and reference architectures. It is the purpose of this paper to give an overview and to outline the general issues of requirements engineering for embedded systems. The process of requirements engineering is decisive for the quality of service of the system. For embedded systems it is typically a part of the systems engineering. It has to pay attention to all aspects of the system behaviour and therefore needs models that go beyond the classical models for hardware or software systems.

Acknowledgement

This work was strongly influenced by discussions within a joint project with the ESG and my colleagues Eva Geisberger and Jan Philipps.

References

[IEEE 94]
IEEE Std 830 -1993: IEEE Recommended Practice for Software Requirements Specification. IEEE 1994

[Parnas 92]
D. L. Parnas: A Technique for Software Module Specification with Examples. CommACM 1992

[Pohl 96]
K. Pohl: Process-Centered Requirements Engineering, Research Studies Press Ltd.: Tannton, John Wileg & Sons Inc.: New York, 1996