Chapter 1, Part 1: The nature of embedded real-time systems

• Objectives
  – To understand the context in which embedded real-time computer systems are used and to appreciate the human and economic environment in which these systems are built and used.
  – To appreciate the kinds of properties that these systems must satisfy
  – To appreciate the advantage of distributed solutions to embedded real-time systems, as well as the difficulties that distribution imply.

Note: The notes on this Part are largely based on Chapters 1 and 2 of Kopetz’ book.
What is a real-time system?

- An *embedded* system
  - Overall system architecture (see Kopetz’ book, fig. 1.1)
  - Reactive systems: interactions
    » input - output - rendezvous - controllable
    » real-time properties:
      • average response time
      • with deadline
        – soft (result still useful if provided after deadline)
        – firm (if not useful after deadline)
        – hard: firm deadline which leads to catastrophe if missed

Different kinds of functional requirements

- Man-machine interaction
- Process control aspects
  - data collection
  - “direct digital control” (real-time control loops)
  - sequencing of operations
  - supervisory control
- Examples of application-specific functions, e.g.
  - call processing by communication network (human-machine interface with user)
  - fault diagnosis in distributed systems management (human-machine interface with system operator)
  - banking system (human-machine interface with clerk or user, no instrumentation interface)
Dependability requirements

• Reliability R(t)
  – R(t) = probability that the system remains operational until time t, given it is operational at time to.
    » If constant failure rate \( \lambda \), then \( R(t) = \exp(-\lambda(t-t_0)) \), and the mean time to failure (MTTF) = \( 1/\lambda \)

• Safety
  » one distinguishes failure modes: malign (catastrophe) vs. benign
  » malign failure modes must be VERY rare
  » There are certain procedures for certifying safety

• Maintainability
  – mean time to repair (MTTR)

• Availability
  – \( A = \text{MTTF} / (\text{MTTF}+\text{MTTR}) \)

• Security

Temporal requirements

• For average response time requirements
  – consider underlying queuing system for accessing resources

• For real-time requirements
  – data collection
    » the "traffic light" example
    » the effect of timing uncertainty (see Kopetz fig. 1.6)
  – sequencing (different "states" or "modes")
  – control loop dynamics (see K fig. 1.3, 1.4 and B fig. 1.2, 1.3, 1.6, 1.7)
    » response function of the process: “process lag” and “rise time”
    » PID algorithm (see B sect. 2.3.1)
  – supervisory control (see B fig. 2.4)
  – real-time deadlines for error detection (“error detection latency”)
Classification schemes

- hard vs. soft real-time systems
  - different aspects: response time, peak-load performance, control of pace, safety, size of data files and short-term accuracy, redundancy type (limited use of roll-back recovery)
  - fail-safe vs. fail-operational
- guaranteed response vs. best effort
- resource-adequate vs. resource-inadequate (see Kopetz)
- event-triggered vs. time-triggered (see Kopetz)

Why a distributed solution?

- “form follows function”
- Hardware structure of a node
  - application programs
  - communication network interface
    - state messages vs. event messages
  - communications controller
- Composability
- Scalability
  - extensions by adding nodes
  - limiting complexity (possibly using state-oriented interface)
  - decreasing cost of communication controllers
- Dependability
  - error containment regions, if possible fail-silent failure mode
  - replication: if possible, replica determinism
  - classification of alarms: catastrophic, hazardous, major, minor, no effect
- Physical integration
Chapter 1, Part 2: Formal and informal methods for system design

• Objective: To understand the context in which formal methods can be used. To appreciate, in general, what formal methods can provide and which difficulties are related to their use.

• Contents and main ideas
  1. The development process
  2. Different system architectures
  3. Properties to be specified
  4. Descriptive formalisms and methodologies of analysis and design
  5. How to avoid faults
  6. Advantages and difficulties of using formal methods

• Additional readings: see bibliographical notes

1. Development process

• System development process
  – application areas
    » software
    » workflow systems
    » manufacturing systems
    » process control systems

• Classical “waterfall” development process
  – different kinds of usage of the “specification”
    » to be developed and validated
    » reference for further development

• Prototyping
  – implementation with partial / reduced functionality
  – executable specifications
Software Life Cycle

requirements

functional specification

detailed specification

implementation code

Software life cycle activities

informal functional requirements

functional specification

detailed specification

implementation code

test cases
2. Different system architectures

Note: For any architecture, one first builds an abstract model (functional specification), before building an implementation.

- The simplest architecture: A single component
  - Model: abstract interface definition
  - Implementation: “concrete” access interface (API) in hardware or software (two kinds of refinement: (1) refinement of interface, (2) refinement of the “body” which implements the function)
Different system architectures (ii)

Layered architecture

- API
- Implementation of underlying layer
- Desired properties

Abstract Model
- Specification of new layer
- Abstract interface
- Specification of underlying service
- Vérification
- Proprétés of service provided by new layer

Different system architectures (iii)

Other architectures (see diagram on other slides):

- Layered architecture (e.g. layered functions of an operating system) (see last slide)
- Embedded systems: to model the complete system
  - "work flow analysis"; to model the whole work process including automated and non-automated parts
  - Process control systems: model of the plant and the control system (see figure below)
- Communication services and protocols: a distributed and layered architecture (see slides below)
  - Service specification provides the global view of the service provided by a layer (the distribution aspect is ignored, a single distributed component is considered)
  - Protocol specification: the view is distributed and abstract (e.g. abstract layer interfaces, certain functional properties of the specified protocol entities can be left implementation-dependent)
A process control system

Physical environment to be controlled

Control program

Desired system properties

Abstract model of global system

Model of the environment

Specification of the control program

Properties

Architecture of communication services and protocols

Abstract view

Communication service

Underlying service

Protocol entity

Implementation view

Local interface A

Local interface B

Communication service

Underlying service

Impl.A

Impl.B

Service view

Protocol view

Site A

Site B

Site A

Site B

Copyright 1999
G.v. Bochmann

Copyright 1999
G.v. Bochmann
3. Properties to be specified

- **Functional properties**
  - properties of a sequential program: results as a function of the input
  - properties of a reactive system:
    - Safeness properties: results (or output(s)) depending on the inputs and the history of all previous interactions (also called “trace”, and which represents the state of the system).
    - Liveness properties: (a) response time (guarantee of a response, guarantee of an average response time, guarantee of a maximum response time); (b) fairness properties (for the case of several “clients”)
    - behavior in exceptional situations (in addition to the behavior in the “normal” situations)
Properties to be specified (II)

• Different levels of abstraction
  When is a specification complete? -- Note: One should leave 
  “implementation details” unspecified!
  − specification of behavior (using abstract interfaces) leaves certain aspects 
    unspecified
  − abstract interfaces (leave the details of "concrete interface" unspecified), 
    based on abstract interaction primitives, such as
    » message passing
    » operation call, possibly over distance (e.g. remote procedure call), 
      where the caller waits for the results
    » rendezvous (with reciprocal waiting)
  − examples of concrete (implementation-level) interfaces
    » a set of procedures to be called at the level of a programming 
      language (typical “application programming interface” (API))
    » communication protocol (X.25, TCP/IP over Ethernet, bus interface 
      within a workstation, etc.)
      • Note: It is important to distinguish between X.25 protocol interface between host 
        computer and network node, and the API between the application program and the 
        operating system to access the communication service provided by the X.25 
        protocol entity of the operating system

Properties to be specified (III)

• Non-functional properties (not much considered by most 
  formal methods)
  − of the product: speed, memory requirements, reliability, robustness, 
    portability, ease of use
  − of the development process: development method, programming language, 
    documentation standards
  − external properties: interoperability, cost, properties required by law
4. Specification formalisms and methodologies of analysis and design

• Descriptive formalisms
  – for the requirements: natural language, data flow graphs, use cases, scenarios, etc..
  – for the functional specification: ...
  – for the detailed design: ...
  – for the code: programming language

• Methodologies for the development activities
  – how to obtain a description of the requirements
  – how to obtain the function specification
  – how to obtain the detailed design
  – how to obtain the code
    » this depends, among others, on the descriptive formalisms used
    » in each case, one has to generate a new description and to verify/validate it

Note: In this course, we are mainly interested in the descriptive formalisms and the question how these formalisms support the automation of the development activities.

5. How to avoid faults?

• Causes of faults
  – inattention
  – wrong understanding of reference document
  – ambiguity of the reference document

• Verification at each development stage (see below)
  – The properties to be checked for the more detailed specification (which is closer to the implementation) include:
    » internal consistency (general properties to be satisfied by any system)
    » the properties implied by the more abstract specification
    » sometimes one distinguishes between the terms verification (“we build the system is right”, involving properties mentioned above) and validation (“we build the right system”, which involves checking, with the user, the requirements which have not been clearly specified)
  – Two general approaches exist in this context:
    » exhaustive validation / proof
    » testing / simulation (this involves (i) test case selection, (ii) test execution, and (iii) test result analysis
6. Advantages and difficulties of using formal methods

• Strong points
  – descriptions without ambiguities
  – automated tools which help with
    » the verification
    » code generation
    » the generation of a test suite
    » the systematic documentation

• Weak points
  – descriptions are more difficult to read
  – the formal descriptions must be complemented with an informal overview description (in order to make the whole more understandable)
  – compromise concerning the choice of the specification language: trade-off between expressive power and efficiency (or effectiveness) of verification
  – the verification tools encounter a complexity limit when applied to complex, realistic system specifications
  – the automatically generated code is sometimes less efficient, its integration with manually written code, especially existing one, is sometimes tedious

Factors influencing the acceptance of formal methods (for protocols)

Factors determining acceptance of formal methods in protocol development:

(1) User training
(2) Intuitive language features
(3) Relation between formal and informal specifications
(4) Wide applicability
(5) Simple tools
(6) Integration into the general software / hardware life cycle