Integrated HW/SW Systems: Requirements

- Analysis process
- Functional requirements
- Performance requirements
- Real-time requirements
- Safety and reliability
- Principles and elements of requirements analysis
Development is not a pure **top-down** process
- use of subcomponents from the shelf
  => **bottom-up**
- lack of accurate estimation in early phases
  => **feedback**
- lack of confidence in feasibility
  => **feasibility studies, prototyping**

=> in practice the development process is a mixture of bottom-up and top-down design
The goals of the analysis phase are

- to identify the **purpose, merit and risks of developing the product**, and
- to identify the **purpose of the product** and to understand its exact requirements
Review of the Development Process

The requirements analysis is a detailed study of the requirements of the system as seen from its environment.

Major tasks are to

- identify,
- analyze and
- classify

the requirements of the product to be built.
Requirements Definition: Contents

- Identification of the system (interfaces to the environment)
- Functional requirements (functionality provided at the interfaces)
- Temporal and performance requirements (throughput, response time, delay, jitter)
- Fault-tolerance and reliability
- Quality (absence of errors)
- Safety
- Operating platform (OS, general HW)
- Power consumption
- Heat disipation
- Operating environment (operating temperature, shock-, dust-resistance, etc.)
- Size
- Mechanical construction
- EMC
- Maintainability
- Extendability
- Support
- Documentation
- Cost (development, deployment and operation)
- Date of completion
  => let’s take a look at some details
The Importance of Requirements

Proper definition of the requirements is vital to ensure quality!
**Functional Requirements**

Definition of the exact *behavior of the system as seen at its interfaces*

Description technique highly depends on the kind of system:
- (state) control system -> state machine
- transformational system -> data flow model (e.g. audio encoder)
  => see section on behavioral models for details

Example of **control** system:
  seat belt control

Example of **transformational** system:
  FIR filter \( o(n) = c_1 \cdot i(n) + c_2 \cdot i(n-1) \)
Performance Requirements

Important performance requirements
- Capacity
- Response time
- Jitter

Examples of performance requirements:
- **capacity**: number (and kind) of events processed per second
- **response-time**: time to process an event (95% percentile)

The performance of the system depends on the load imposed on it, i.e. the traffic model.

The performance is highly influenced by the design, especially
- the module/component design
- the available processing and communication resources
- the scheduling strategy

Performance cannot be “added on” to the implementation
Real-time (Temporal) Requirements

Definitions:
- If the result is useful even after the deadline, we call the deadline \textit{soft}.
- If the result is of no use after the deadline has passed, the deadline is called \textit{firm}.
- If a catastrophe could result if a strict deadline is missed, the deadline is called \textit{hard}.
- A real-time computer system that has to meet at least one hard deadline is called a \textit{hard real-time system}.

System design for hard- and soft real-time systems is fundamentally different.
Real-time (Temporal) Requirements

Examples:

- **soft deadlines**
  - public transportation system
  - airport luggage transport system

- **firm deadlines**
  - audio processing
  - video processing

- **hard deadlines**
  - control of nuclear or chemical processes (chain reaction)
  - railway traffic control
  - air traffic control
Real-time Systems – Classification

- On the basis of the **external requirements**
  - hard/firm real-time versus soft real-time
  - fail safe vs. fail operational
    (e.g. train control system vs. fly-by-wire system)

- On the basis of the **design and implementation**
  - guaranteed timeliness vs. best effort
  - resource adequacy vs. no resource adequacy (sufficient computational resources to handle all specified peak loads and fault scenarios)
  - event triggered vs. time triggered
Time Triggered (TT) vs. Event Triggered (ET) Systems

A system is *Time Triggered (TT)* if the control signals, such as
- sending and receiving of messages
- recognition of an external state change
are derived solely from the progression of a (global) notion of time.

A system is *Event Triggered (ET)* if the control signals are derived solely from the occurrence of events, e.g.,
- termination of a task
- reception of a message
- an external interrupt.

Note that the triggering method is often an attribute of the implementation and not necessarily a requirement.
Safety Requirements: Fail-Safe vs. Fail-Operational

Safety requirements define the action taken in the case of a failure.

A system is **fail-safe** if there is a safe state in the environment that can be reached in case of a system failure, e.g. ABS, train signaling system. In a fail-safe application a computer has to have a high *error detection coverage*.

Fail safeness is a characteristic of the application, not the computer system.

A system is **fail-operational**, if no safe state can be reached in case of a system failure, e.g. a flight control system aboard an airplane. In fail-operational applications the computer system has to provide a minimum level of service, even after the occurrence of a fault.
Reliability Requirements

Reliability denotes the probability for a failure or absence from failure of a system

Examples of reliability figures are

- MTTF (mean time to failure)
- MTBF (mean time between failures)
- probability for up-time (e.g. 99.995%)

The reliability of the system can be estimated/calculated (in theory) from the reliability of its components

A system that ensures that it still functions correctly even in the case of failure of some components is called a fault tolerant system (i.e. it is able to tolerate faults of single components of the system)
Predictability in Rare Event Situations

A *rare event* is an important event that occurs very infrequently during the lifetime of a system, e.g. the rupture of a pipe in a nuclear reactor.

A rare event can give rise to many correlated service requests (e.g. an *alarm shower*).

In a number of applications, the merit of a system depends on the *predictable performance in rare event scenarios*, e.g. a flight control system.

In most cases, typical workload testing will *(not)* cover the rare event scenario.
Principles and Elements of the Analysis Model

Guidelines for the analysis:

- understand the problem first! (before you begin to create the analysis model)
- record origin and reason for every requirement
- use multiple views of requirements (data model, functional model, behavioral models)
- prioritize requirements
- eliminate ambiguities

Elements of the analysis model:

- data dictionary
- process specification (data-flow diagram)
- control specification (state-transition diagram)
- data object description (entity-relationship diagram)
- functional specification (sequence diagram)

Specific methods and tools for various application areas have been proposed, e.g. real-time systems, transformational systems, control systems, communication systems, etc.
References

Behavioral Models and Specification Languages

**Behavioral Models**
- Finite State Machine (FSM)
- NDFSM
- composed FSM
- Petri Net (PN)
- Data Flow Graph (DFG)
- Control Flow Graph (CFG)
- Control/Data Flow Graph (CDFG)

**Specification Languages**
- StateCharts
- SDL
- VHDL
- SystemC
- ...

**Basic Concepts**
- concurrency
- hierarchy
- communication
- synchronisation
- exception handling
- non-determinism
- timing
Why do we need Behavioral Models?

Models abstract from the real world to some extent (describing part of the system from a specific view)

- Models may serve very different purposes, e.g. to describe, document or reason about
  - the behavior of a system
  - the performance of a system
  - or some other purpose (safety, reliability, shock, temperature, ...)

Behavioral Models (or Models of Computation) represent formal frameworks to describe the behavior of systems or parts thereof

- Formal definitions ease reasoning and refinement

- The description of the behavior may be abstract or concrete, e.g.
  - an abstract state machine describing the behavior of an elevator control
  - a C program, assembler program or VHDL description

- Behavioral models may concentrate on
  - the externally-visibly behavior (analysis phase) or/and
  - the internal details of the system (design and implementation phase)
Behavioral Models and Specification Languages

**Behavior models** (e.g. FSM, PN, DFG) represent well defined basic mechanisms with underlying syntax and semantics to model specific aspects of a system or parts hereof.

**Specification languages** (e.g. Statecharts, SDL, VHDL) employ a set of basic modeling mechanisms useful to specify systems of a specific application domain.

Behavioral models and specification languages allow to:

- capture **unambiguously** the required functionality
- **verify correctness** of the functional specification wrt. given properties
- **synthesize** part of the specification
- use a variety of tools

**Important characteristics** of behavior models and specification languages:

- Expressiveness
- Generality/applicability
- Simplicity
- Compilability/synthesizability
- Verifiability
Behavioral Models (discrete time and state)

**State-oriented** models (behavior)
- Finite State Machines
- Petri Nets
- Hierarchical concurrent FSM

**Action-oriented** models (behavior)
- data flow graph
- control flow graph

**Structural Models**
- Structure-oriented models
  - component connectivity graph
- Data-oriented models
  - entity relationship diagram
  - Jackson’s diagram

**Heterogeneous** models
- control/data flow graph
- structure chart
- programming language paradigm
- object-oriented model
- program-state machine
- queueing model
More Models and Languages ... to complete the confusion!

More models ...

- continuous time
- continuous state
- structure-oriented
- data-oriented
- ...

Each of these provides a formal framework for reasoning about certain aspects of the system

Focus here

- behavior rather that structure
- discrete time rather than continuous
- discrete state (digital) rather than continuous (analog)
Behavioral Models – Application Needs

**Telecommunication**
- Heterogeneous specifications including
  - data processing
  - control functions
- **Data processing**, e.g. encryption, forward error correction, …
  - computations done at regular (often short) intervals
  - efficiently specified and synthesized using Data Flow models
- **Control functions** (data-dependent and real-time)
  - say when and how data computation is done
  - efficiently specified and synthesized using FSM models

Need a common model to perform global system analysis and optimization

**Reactive Real-Time Systems**
- “react” to external environment
- maintain permanent interaction
- ideally never terminate
- timing constraints (real-time)
- As opposed to
  - transformational systems
  - interactive systems
Finite State Machines (FSM)

Functional decomposition into states of operation

- finite states
- transitions between states
- event triggered transitions
- neither concurrency nor time (sequential FSMs)

Typical applications:

- reactive (control) systems
- protocols (telecom, computers, ...)

Integrated Communication-Systems

Andreas Mitschele-Thiel

13-Feb-12
FSM Example: Seat Belt Control

Informal specification:
If the driver
- turns on the key and
does not fasten the seat belt within 5 seconds
then an alarm beeps for 5 seconds, or
- until the driver fastens the seat belt, or
- until the driver turns off the key

Formal specification:

Interpretation:
Conditional expression => Output
If none of the conditions is satisfied, implicit self-loop in the current state
Non-deterministic FSM (NDFSM): Incomplete Specification

Example: parity error checking and synchronization after 8 bit

- Partially specified (incomplete):

Focus on synchronization after 8th bit, not on parity

- More completely specified as *even parity*.
NDFSM: Unknown parts of the Behavior

Modeling the environment

- Nondeterminism is useful to:
  - optimize
    (don’t care conditions)
  - verify
    (exclude impossible cases)

Example: driver model

- Can be refined
  e.g. introducing timing constraints
    (minimum reaction time of 0.1 s between actions)
NDFSM: Time Range

Special case of unspecified/unknown behavior, but so common to deserve special treatment for efficiency

Example: nondeterministic delay (between 6 and 10 s)
NDFSMs and FSMs

- Formally FSMs and NDFSMs are equivalent (Rabin-Scott construction, Rabin ‘59)
- In practice, NDFSMs are often more compact (exponential blowup for determinization)

Example: non-deterministic selection of transition a in state s1

Equivalent deterministic FSM
Finite State Machines – Discussion

Advantages:
- Easy to use (graphical languages)
- Powerful algorithms for
  - synthesis (SW and HW)
  - verification

Disadvantages:
- Sometimes over-specify implementation
  (sequencing is fully specified)
- Number of states can be unmanageable
- Numerical computations cannot be specified compactly
  (need Extended FSMs)