Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler

Design of Complex Embedded Systems

Microcombi: the (presumably) first prototype of the GDR with PC architecture (Technische Hochschule Ilmenau, 1980)
Technische Universität Ilmenau
Department of Computer Science and Automation

Computer Architecture and Embedded Systems Group
P.O. Box 100565
D-98684 Ilmenau, Germany
Phone +49-3677 / 69 2825
E-Mail ra-sek@tu-ilmenau.de
https://www.tu-ilmenau.de/en/ra/

The following material is intended exclusively for the personal use of participants as teaching material for Prof. Dr. W. Fengler's lecture at the Universidade NOVA de Lisboa, Faculdade de Ciências e Tecnologia, 2829-516 Caparica, Portugal during the week from 7 to 11 May 2018. A passing on to third parties is not permitted.

By downloading the file you agree to this condition.
http://ec.europa.eu/programmes/erasmus-plus/

**Erasmus+ is the EU Programme**

in the fields of education, training, youth and sport for the period 2014-2020. Education, training, youth and sport can make a major contribution to help tackle socio-economic changes, the key challenges that Europe will be facing until the end of the decade and to support the implementation of the European policy agenda for growth, jobs, equity and social inclusion.
Studying abroad:

→ Erasmus+ can offer you support during your period of study abroad.

Traineeships abroad

→ Erasmus+ supports traineeships (work placements, internships, etc.) abroad for students enrolled in higher education in a Programme Country. These opportunities are also open to recent graduates.
Technische Universität Ilmenau
The Beginnings 1894

Curiebau

Faradaybau

Fotos: © TU Ilmenau
Technische Universität Ilmenau in Summer
Technische Universität Ilmenau in Winter
# Content

## 1. Embedded Software Development

1.1. Common Example ............................................................................................................... 12

1.2. Design Flow and Process Model ....................................................................................... 26

1.3. Function Block Oriented Design ....................................................................................... 33

1.4. Design Example and Function Block Oriented Design ................................................... 44

1.5. Module Implementation and Realisation ........................................................................... 52

## 2. Simulation Driven Development

2.1. The W Process Model ......................................................................................................... 58

2.2. Variations of simulation-based tests ................................................................................ 61

## 3. Object Oriented Design

3.1. Used Diagrams .................................................................................................................... 68

3.1.1. Use Case Diagram ........................................................................................................ 68

3.1.2. Activity Chart ............................................................................................................... 70

3.1.3. Class Diagram .............................................................................................................. 72

3.1.4. Sequence Chart ............................................................................................................ 75
3.1.5. State Chart .................................................................................................................... 77
3.2. Application Software Implementation ............................................................................. 79
4. Verification ................................................................................................................................. 85
List of Figures .............................................................................................................................. 100
1. Embedded Software Development

1.1. Common Example

what is an Embedded System? (Figure 1) ->

Embedding System

Embedded System

- different application areas
- computer, specialised
- function is realised in software
Figure 1: Embedded System
First Example of a complex embedded system:

(1) The measurement technology represents an important domain for the use of more or less complex embedded computer systems.
(2) Therefore, this domain can very well be used to describe a design process.
(3) Modern measuring technology requires not only a safe system but also a system which is adapted to measurement uncertainty and time behavior.
(4) The complexity of the algorithmic part results from alternative filter, control and signal processing functions.
(5) These requirements must be fulfilled in a specific cost frame of the device.
Figure 2: Nanopositioning and Measuring Machine (NPMM, real images)
The particularly powerful measuring systems in the field of length measurement technique with resolution and measurement uncertainty in the nm boundary region (e.g. nanopositioning and measuring machine (NPMM) Figure 2).

Goal:

Developing a nanomeasuring machine (NPMM-200) with a measuring volume of \(200 \times 200 \times 25 \text{ mm}^3\) that must perform signal acquisition and preprocessing at a high frequency with extremely low latency for subnanometer resolution, high-precision moving stage control, and nanometer uncertainty.
Solution (Figure 3):

Using **three** modular, scalable NI PXI systems and **11 field-programmable gate array (FPGA)** modules to complete the **signal- and data-processing unit** and easily integrate innovative metrological and control concepts.
Figure 4: NPMM: Signal- and Data Processing Unit (hardware structure)
Figure 5: Trajectory Tracking Control Scheme

Figure 6: Structure of a PI State-Space Controller with Preliminary Filter
Common Example for the Following Subchapters: Weighing Machine

(1) Dynamic **weighing technology** require complex and high-performance information processing functions.

(2) The **complexity of the algorithmic part** results from alternative filter, control and signal processing functions.

(3) The **most important objectives** of all measures in information processing, including the measuring technology, are:

- to **shorten** the **measuring time**,  
- **reduce** the **uncertainty** of measurement and  
- **increase** the **resolution**.  

- These requirements must be **fulfilled in a specific cost frame** of the device.

**Weighing machine for industrial use, e.g. in dosing processes in the pharmaceutical industry:**

**Embedding System (mechanics, optics, electrics, electronics (Figure 7)):**
Figure 7: Weighing Machine: Prototype
Figure 8: Weighing Machine: Prototype schematic
The quartz glass bends due to the acting weight force. This changes the angle. The resulting interference optical signals move in one of the two possible directions, depending on the load or relief.
Figure 9: Weighing Machine: Signals

fw (forward), bw (backward): count signals
The interference optical strips are photoelectrically scanned, amplified and form the two digital signals \( x_1 \) and \( x_2 \) (Figure 9).

Depending on the direction of movement, a phase shift of + or - 90 degrees occurs between \( x_1 \) and \( x_2 \).

A task of the embedded system will be to form forward and backward counting signals \((fw, bw)\) from signals \( x_1 \) and \( x_2 \). For this purpose, the phase position must be evaluated and a corresponding counting pulse must be formed for each level change (edge) of \( x_1 \) and \( x_2 \).

Some requirements from the example:
Measuring time: 100 ms (Jump-shaped mass)
Dissolution \( 10^{-5} \)
Uncertainty of measurement/accuracy: \( 4\times10^{-5} \)
Temperature range: \(-10 \, ^\circ C \ldots +50 \, ^\circ C\)
1.2. Design Flow and Process Model

Many embedding/embedded systems are based on the character of complex mechatronic systems. The design of such systems can only be implemented effectively using a model-based design procedure.

The design flow should follow the familiar V-process (Figure 10) model.
Test cases

Figure 10: V-Process Model
(1) In the model-based design, the functional requirements are transferred into a model.
(2) After that the model will be transferred into the implementation structure.
(3) Usually, several models are used with different degrees of abstraction and models of different character are also be used.
(4) The test cases are designed on model level on the left part of the V and can used for simulation and the later test of the implementations.
(5) The integration and test on the right side of the V starts with the lowest abstraction level (modules and module test) and continues with the higher levels till to the implemented or realized functions.

For the considered application domain, particular methods and design methods needs to be supported:
• The overall system design is performed according to the top-down principle, beginning with the highest abstraction level (requirement specification) to the lowest (realization / implementation).

• In the reuse of previous solutions, the meet in the middle approach (combination of top-down and bottom-up principle) can be integrated as well. Both implemented functions and model parts can be used.

• Embedded system (instrumental part) and embedded system (information processing) are developed in parallel and close interaction.

• There is a separation in functional design (algorithms) and structural design (realization and implementation).

• Stages build one after another are carried out model-based. This means that both models are used for the measurement as well as for information processing part.

• The models should be executable (simulable). This allows an early validation of the partial and total system at different abstraction levels. Test scenarios of the models later provide the basis for implementation testing.
• This leads to the later described extension of the V-model to the W-process model.
• Possible parts for realization platforms of information processing are the embedded software, hardware and reconfigurable hardware (FPGA, Field Programmable Gate Array).
• In the transition from the functional to the structural models (platform-dependent), partitioning and possibly later repartitioning into the measuring technical and the three information technological variants part has to be carried out. Whereby for alternative assignable model parts the technically and economically most sensible variant is to be choosen.
• In the abstraction level, in contrast to the design, various integrations of the partial solutions, as well as testing itself, integration and acceptance tests are carried out after realization/implementation. This happens with the use of the above-mentioned model-based test scenarios.
Design steps:

Definition of the requirements:

**Specification of the functions** the realised system shall do.

- written in natural language with special words from the technical direction of the device.

The functional requirements for a system to be designed are **transferred into a model** (possibly into an executable specification model or a virtual prototype in the technical system design)
Example weighing machine (see above):

Some requirements from the example:

Measuring time: 100 ms
(Jump-shaped mass support)
Dissolution: $10^{-5}$
Uncertainty of measurement/accuracy: $4 \times 10^{-5}$
Temperature range: $-10 \, ^\circ C \ldots +50 \, ^\circ C$
1.3. Function Block Oriented Design

Preliminary design -> functional design

Basis:

Hierarchical data flow graph with additional information in the function blocks (Figure 11).

Function block: rectangle

Content:

Function is what shall the block do.

Restriction: what shall the block have to do not, or some addition values for the realise of the function which are not direct combined with the behaviour.

For instance (hardware (HW) block): electrical power consumption).
Figure 11: Function Blocks, two Levels
Subfunctions:
the functions that are created when refining the current function.

**Input, Output** (both of blocks), **Connection** between blocks with **type** (data type, not only types form programming languages, possibly hierarchical combined). All are directed.

**Refinement**: Replace blocks with hierarchically subordinate blocks and connect their inputs and outputs to each other and/or to inputs and outputs of the higher-level block.

**Coarsening**: Combining blocks and connecting their inputs and outputs to each other and/or to inputs and outputs of the resulting hierarchically superior block.
Function Block Oriented Design with Matlab/Simulink and Stateflow

(* Academic License - for use in teaching, academic research, and meeting
  * course requirements at degree granting institutions only. Not for
  * government, commercial, or other organizational use.)

MATLAB® is the high-level language and interactive environment used by millions of engineers and scientists worldwide. The matrix-based language is a natural way to express computational mathematics.
Simulation and Model-Based Design


Simulink® is a block diagram environment for multidomain simulation and Model-Based Design. It supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with MATLAB®, enabling you to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.

MATLAB®, Simulink®, Stateflow® are trademarks or registered trademarks of The MathWorks, Inc.
Model and simulate decision logic using state machines and
https://de.mathworks.com/products/stateflow/features.htm

Stateflow® is an environment for modeling and simulating combinatorial
and sequential decision logic based on state machines and flow charts.
Stateflow lets you combine graphical and tabular representations, including state
transition diagrams, flow charts, state transition tables, and truth tables, to model
how your system reacts to events, time-based conditions, and external input
signals.

With Stateflow you can design logic for supervisory control, task scheduling, and
fault management applications. Stateflow includes state machines animation and
static and run-time checks for testing design consistency and completeness
before implementation.
Figure 12: Simulink Function Blocks, Highest Level

Figure 12 shows a **block containing a subsystem**, here named with "function" of the block type "double". The block contains **two inputs** (In1, In2) and **two outputs** (Out1 and Out2), both of type "double".
The references to the subblocks are specified as a backward reference for Simulink, that is, not visible at this level. The restrictions given here are text only.
Figure 13: Simulink Function Blocks, Middle Level

Figure 13 shows the block of Figure 12 as a **refined subsystem**. It contains **three blocks** (again subsystems to be refined), here named "function_1 to 3" of block type "double". The blocks contain **the two inputs** (In1, In2) and **two outputs** (Out1 and Out2) of the higher-level "function" block and an **additional internal connection" **Connection_0"". 
The references to the subblocks are specified as a **backward reference** for Simulink, that is, visible at this level (function_blocks_v2_1b/Subsystem1). The specified, now divided restrictions are also here only text.
After refinements: -> detailed design (structural design, Figure 14)

The Blocks are possible for direct realisation with HW or SW – Elements and the connections are in HW realisable or in SW. There are necessary: data structures and between HW and SW as interface elements.

**Figure 14: Simulink Function Blocks, Lowest Level (combined, software)**
1.4. Design Example and Function Block Oriented Design

Temperature (t) → Measurement logic

Weigt value (w) → Measurement logic

Digital information processing \( r = f(w, t) \)

Result (r) \( \mu u \leq 0.1mg \), \( m_t \leq 0.1s \) → Display

Network

Parameters \( a, b, \ldots \)

... Environment

... Functional blocks

... Functions

... Restrictions

... Data

Figure 15: Design Example Higher Level
In Figure 15:

**Weighing machine** (balance, mechanical, optical) with time response and disturbances (vibrations, temperature). Outputs of this part physically: **weight and temperature**.

**Measuring logic**: Contains all parts of the conversion of the physical quantities into digitally represented quantities, whereby these represent a more or less proportional function to the original quantities.

**Digital information processing**: Main part of the embedded system, converts the disturbed and time-dependent signal \( wd \) into a weight value that is correct within the requirements specification.

**Display**: Everything you need to display the calculated weight value.

**Network**: Connection to a network for parameter transfer.

**Interfaces** see Figure 15: abstract, complex (possibly refine)
Digital information processing
\[ r = f(w, t, a, b, ...) \]
Subfunctions:
- Dynamic correction
- Static correction
- Secundary computation

Parameters
\[ a, b, \ldots \]

Result
\[ \mu < 0.1 \text{mg} \]
\[ m_t < 0.01 \text{s} \]

Figure 16: Design Example Information Processing, Higher and Middle Level
Refinement of the Digital information processing block (Figure 16):

3 sub-blocks, these operate interfaces to the outside, possibly refined (here parameters)
additional internal connections.
Refinements of the other blocks: …

**Figure 17: Design Example Dynamic Correction, Middle and Lower Level**
Refinement block Dynamic correction (Figure 17):

Block **Mean value**: for the amplitude of smaller but higher-frequency ambient disturbances. Its output becomes input of the following:

**Filter** (digital filter)

both are **atomic blocks**, i.e. they are directly implemented, e.g. as library functions, but also design decisions: e.g. which filter type, which parameters etc.
The block “Measurement logic” of Figure 15 contains: a block Direction discriminator (conversion of the balance signals $x_1$, $x_2$, see Figure 9) into forward/backward pulse (fw, bw) values given to a counter block which forms wd (uncorrected digital weight value) by corresponding up-counting and down-counting.

Higher level of Direction discriminator, with interfaces to the outside (Figure 18, Simulink, Stateflow-Chart):

Inputs: $x_1$, $x_2$; Outputs: fw, bw

Lower level to it (Figure 19, Refinement of the chart of Figure 18):

Finite automaton, listed here as StateChart.

The inputs $x_1$, $x_2$ form variables of the transition functions. Outputs are generated at the transitions or states.

Figure 19 shows the internal state chart.
Figure 18: Stateflow Function Blocks, Highest Level
Figure 19: Stateflow Function Blocks, Lower Level
1.5. Module Implementation and Realisation

After refinements: -> detailed design (structural design)

Blocks are possible for **direct realisation with HW or SW** – Elements und the connections are in HW realisable or in SW. There are necessary: data structures between HW and SW as interface elements.

**Module implementation:**

from realisation structure without concrete elements or programme modules to real elements with special values and written software (new or reused) for the SW function parts.

Example see Figure 20, Figure 21.
Coded software Implementation (C++):

* Simulink Coder version : 8.13 (R2017b) 24-Jul-2017 *

```c
{ case Chart1_IN_Z00:
    /* During 'Z00': '<S1>:1' */
    if (Chart1_U.X1 && (!Chart1_U.X2)) {
        /* condition x1 && !x2 == 1 ? */
        /* Outport: '<Root>/bw' */
        /* Transition: '<S1>:14' */
        Chart1_Y.bw = 1;
        Chart1_DW.is_c1_Chat1 = Chart1_IN_Z10;
    } else { /* Chart1 is still in Z00 */
        /* condition x1 && !x2 == 1 */
        /* Outport: '<Root>/fw' */
        /* Transition: '<S1>:5' */
    }
}
```

Chart1 to Z10 */
/* Entry 'Z10': '<S1>:9' */

```
else {
    if (((!Chart1_U.X1) && Chart1_U.X2) { /* condition x1 && !x2 == 1 */
        /* Outport: '<Root>/fw' */
        /* Transition: '<S1>:5' */
    }
```

Content 1
Chart1_Y.fw = 1; /* output fw = 1 */
Chart1_DW.is_c1_Chart1 = Chart1_IN_Z01; /* condition = 1 ->
Chart1 to Z01 */
/* Entry 'Z01': '<S1>:2' */
}
}
*** /* and so on */

Figure 20: Excerpt from the Generated c++ Program (brown comments not by the coder)
Figure 21: Stateflow Function Blocks, Lower Level (Figure 19): Connection to the code
Coded hardware Implementation (VHDL):

-- Generated by MATLAB 9.3 and HDL Coder 3.11 --
CASE is_Chart1 IS
  WHEN IN_Z00 =>         -- Chart1 is in Z00
    sf_internal_predicateoutput_0 := hdlcoder_to_stdlogic((x1 AND ( NOT x2)) = '1');
    -- condition x1 && !x2 == 1 ?
    IF sf_internal_predicateoutput_0 = '1' THEN
      bw_reg_next <= to_signed(16#01#, 8);   -- output bw = 1
      is_Chart1_next <= IN_Z10;
    ELSE            -- Chart1 is still in Z00
      b_sf_internal_predicateoutput_0 := hdlcoder_to_stdlogic((( NOT x1) AND x2) = '1');
    END IF;
  WHEN IN_Z10 =>         -- Chart1 is in Z10
    is_Chart1_next <= IN_Z10;
  ELSE            -- Chart1 is still in Z10
    is_Chart1_next <= IN_Z10;
  END CASE;
IF b_sf_internal_predicateoutput_0 = '1' THEN
  fw_reg_next <= to_signed(16#01#, 8);  -- output fw = 1
  is_Chart1_next <= IN_Z01;     -- condition = 1 -> Chart1 to
  Z01
  ***            -- and so on

*Figure 22: Excerpt from the generated VHDL program (brown comments not by the coder)*
2. Simulation Driven Development

2.1. The W Process Model

Simulation Driven Development enables early testing and thus the earliest possible error correction. This leads to significantly more effective development and, above all, shorter development times.

- **Simulation models** of the system to be developed are used during the entire developing process

- In addition to the simulation model of the system to be developed, SDD uses a *simulation model of the developing process* itself

- This simulation model allows automation of development steps in the form of workflows like optimization cycles, test cycles, revision controls and document sharing with component manufacturers

The V development process is extended to the W development process (Figure 23).
Figure 23: W-Process Model
(1) The **validation of the implementation structure** against the requirements is complemented by the **validation of the model against the requirements** and the **implementation structure compared to the model**.

(2) The implementation structure can be furthermore **formally verified against the model** if the techniques have been developed or are available for this purpose.

(3) **Test is possible on the left side of the W** if the used models have the possibility to simulate them (executable models).

(4) The **test cases** are designed on model level **on the left part of the W** and can **used for simulation** and the later **test of the implementations**.

(5) The integration and test on the right side of the W **starts with the lowest abstraction level** (modules and module test) and **continues with the higher levels** till to the implemented or realized functions.
2.2. Variations of simulation-based tests

From the **point of view of the embedded system**, simulation based tests can be carried out at various levels:

- **Model of Embedding System** (Model Simulator on Host Computer)
- **Model of Functions** (for Software Design) (Model Simulator on Host Computer)

![Figure 24: Model in the Loop Test](image_url)

Both **the embedding** and **the embedded system** are simulated (model in the loop test, Figure 24)

This is especially useful on the W stages of the functional and technical design.

The procedure as **workflow** shows Figure 25.

The **Simulink example** (Figure 12) used above could be simulated in this way.
Modelling of function

Modelling of test data generator of test case

Modelling of test result evaluator of test case

Generating simulation model

Executing simulation model

Change models of function and/or test data generator and/or test result evaluator

Test result not ok

Test result ok

Test result ok

Documentation of models (tested function, test data generator, test result evaluator and test results of test case)

Figure 25: Simulation Driven Development
Figure 26: Simulink Testing with Simulable Models
Early testing according to in1, in2: Simulation of all blocks with different types and parameters with suitable **signal generation** (by library blocks or recorded data of the current embedding system, possibly connection of the same, here: function generator blocks) and **value display** related to time.

**Software in the loop** (Figure 27) is used for the integration tests of software modules and functions of the embedding system.

![Diagram](image)

*Figure 27: Software in the Loop Test*

The **embedded system** and the target **hardware do not yet have to be physically present.**
Hardware in the loop (*Figure 28*) can be used for hardware and software systems under test with functions of the embedding system (especially before physical integration with it).

*Figure 28: Hardware in the Loop Test*
For *hardware in the loop* test, the *target system* (embedded system) and the *simulator* of the model of the embedding system **must be coupled via interfaces** (Figure 29).

**Figure 29: Hardware in the Loop Test via Interfaces**
It is necessary to **synchronize the behaviour of the HW-SW** (Embedded System) **with time behaviour of the simulator** (e.g. with the simulation clock).

In normal case the Embedded System is faster as the simulator. Then for instance should it work with a lower clock.
3. Object Oriented Design

3.1. Used Diagrams
3.1.1. Use Case Diagram

Use cases of the system (Figure 30):

- Which tasks does the system perform?
- What services does the system provide?

Participants in the system:

- Which persons or other systems are to be considered?
- Which participants have influence on or are influenced by use cases?
Figure 30: Simplified Use Case Diagram of the Weighing Machine (Figure 8)
3.1.2. Activity Chart

Representation of processes (Figure 31):

- **Steps of a procedure**,  
- "Flowchart",  
- Description of e.g. a method,  
- **Focus:**  
  - **parallelism/concurrence**,  
  - **sequences**,  
  - **splits, joins (and, or)**,  
  - **loops**,  
  - **hierarchy**.
Figure 31: Activity Chart of mesure_execute
3.1.3. Class Diagram

- **Static structure** (Figure 32, Figure 33):
  - Existing elements (classes, types)
  - Internal Structure of a System
  - Static relationships of the force elements
- No time-dependent information
Figure 32: Class Diagram of the Weighing Machine (Figure 8), Highest Level
Figure 33: Class Diagram of the Weighing Machine (Figure 8), information_processing
3.1.4. Sequence Chart

(= Message Sequence Chart)

**Representation of a scenario** (Figure 34):

- Objects involved with lifeline
- Exchange of messages between objects

**Sequence of communication**

Focus: sequential processes
Figure 34: Sequence Chart of the Weighing Machine (Figure 8) (dynamic and static correction, simplified, without digital_filter)
3.1.5. State Chart

States of an element (Figure 35):
Which states can an element (object, class) have?
How do you get from one state to another?

Actions that change the state
Reaction to incoming events
Figure 35: State Chart of f_b_discriminator (similar to Figure 19)
3.2. Application Software Implementation

**Figure 36: Class Diagram of the Weighing Machine (Figure 8), Partitioning (simplified)**
Partitioning in Figure 36:

Environment: embedding system, user

Controller components: user interface

Operating system: sensor control, interrupt system, timing

Software: **application software** to realise the specified functionality

1. **Code generation from class diagram**
   - Generation of C++:
     - header files
     - source code files
   - **Implementation** of:
     - Class and Interface Declarations
     - Relationships:
       - inheritance
       - aggregation
- association

2. Definition of methods

→ Functionality

Example: Interaction between objects (Figure 37, coded by a programmer, excerpt of Figure 38)

![Figure 37: Interaction between objects]
void state_change ()
{
    //input
    const bool x1 = input_obj->sensor_read(0);

    ***
    //output
    if ( -- conditions -- )
    {
        m_output = false; //forward
        f_b_counter_obj->count(m_output);
    }

    ***
}

Example: f_b_discriminator (Figure 35)

Definition of a method:

**State chart**: internal processes of an object

Coded by a programmer (Figure 38):

→ similar to Figure 20.

```c
void f_b_discriminator::state_change()
{
    const bool x1 = input_obj->sensor_read(0);
    const bool x2 = input_obj->sensor_read(1);
    //input
    if (((m_state == 0) & !x1 & x2) || ((m_state == 1) & x1 & x2) ||
        ((m_state == 2) & !x1 & !x2) || ((m_state == 3) & x1 & !x2))
    {
        m_output = false; //forward
        f_b_counter_obj->count(m_output);
    }
}
```
if (((m_state == 0) & x1 & !x2) || ((m_state == 1) & !x1 & !x2) ||
((m_state == 2) & x1 & x2) || ((m_state == 3) & !x1 & x2))
{
    m_output = true; //backward
    f_b_counter_obj->count(m_output);
}

//next_state_function
if (!x1 & !x2) m_state=0;
if (!x1 & x2) m_state=1;
if (x1 & !x2) m_state=2;
if (x1 & x2) m_state=3;
}

Figure 38: Excerpt from C++ Program of the State Chart of Figure 35
4. Verification

In model-based design, validation can occur in several variants:

- Realization structure against functions
- Model versus functions
- Model at a lower abstraction level compared to that at a higher abstraction level.
- Implementation structure compared to the model.

Additionally for realization structure against model: verification (formal), which generates proof of correctness with regard to provable properties. (e.g. model checking for automat-based models, Petri-Netz property analyses, complete simulation)
Validation: "Do I build the correct (desired) system".
Methods: testing, simulation (the second see chapter 2. page 58) etc.

Verification (formal): "Do I build the system correctly"
Methods: formal methods using models

Verification using Petri Nets

Verification can be applied mainly to the design (left side of the W). This involves the formal proof of certain properties of the models that are to be adhered to. This is based, for example, on formulated restrictions.
In the following, Petri nets are used for the property liveliness (no deadlocks) and time interval weighted Petri nets for real-time properties.

Basic knowledge of Petri nets (structural definition, transition rule, reachability graph, liveliness, time weighting) is assumed here.
- Starting point are function blocks with discrete event behaviour.
- The input data has event character. The function block starts when all input data is available and generates output data after execution, again with event character.
- Events that are used are consumed, that is, they are only available for reuse after they have been regenerated.
- Events can be linked with AND or OR at the input and output.

A function block example with time interval restrictions is used. Interval means that a minimum and maximum execution time is known or assumed. In the concrete case, the actual value lies between these two values.
Figure 39: Function Block Example for Verification

Figure 40: Petri Net Model of Figure 39
The transformation into Petri nets uses as rules:

- Functions become transitions with time interval weighting and start and end place.
- An OR split becomes a place with one pre transition and two or more post transitions.
- An OR join becomes a place with two or more pre-transitions.
- An AND join becomes a transition with two or more post places and one pre place.
- An AND join becomes a transition with two or more pre places and one post place.
- Transitions of functions and AND structures can be combined.
- Start and end places of functions and OR structures can be combined.

The resulting Petri Net of the Example of Figure 39 shows Figure 40, the resulting reachability graph Figure 41.
You can see that there are two dead markings. The process structure is not correct.

For the consideration of the property liveliness is to be supplemented: transitions start and end and a place between them. All start places of the process are post places of a start transition (start) and all end places are the pre places of the end transition (end). The whole is completed by a place for which end is pre transition and start is post transition.
Figure 41: Reachability Graph of PN of Figure 40
Figure 42: Function Block Example for Verification, Corrected

Figure 43: Petri Net Model of Figure 42
Figure 42 shows a possible correction.

The OR split between function_1 and function_2 and function_3 is replaced by an AND split. The resulting reachability graph (Figure 44) shows that the net is live. There are not deadlocks.
The following shows the principle of calculation values for execution time intervals that consist of several functions (from the process start to its end or partial processes within).
(1) The net must be live.
(2) An OR split must be terminated with OR join.
(3) An AND Split must be terminated with AND join.
(4) Between split and join can be other constructions: sequences and split-join structures with property (2), (3).

The resulting calculations of interval values for

- sequence shows Figure 45,
- OR structure Figure 46 and
- AND structure Figure 47.

The calculation of the corrected example of Figure 43 using this rules shows Figure 48.
both functions: \([ (t_{1\text{min}} + t_{2\text{min}}), (t_{1\text{max}} + t_{2\text{max}}) ] \)

*Figure 45: Time interval calculation: Sequence*
function_1 \[t_{1\text{min}}, t_{1\text{max}}\]

start

end

function_2 \[t_{2\text{min}}, t_{2\text{max}}\]

both functions:
\[\min(t_{1\text{min}}, t_{2\text{min}}), \max(t_{1\text{max}}, t_{2\text{max}})\]

*Figure 46: Time interval calculation: OR-Structure*
function\_1

\[ [t_{1\text{min}}, t_{1\text{max}}] \]

function\_2

\[ [t_{2\text{min}}, t_{2\text{max}}] \]

both functions:

\[ \max(t_{1\text{min}}, t_{2\text{min}}), \max(t_{1\text{max}}, t_{2\text{max}}) \]

**Figure 47: Time interval calculation: AND-Structure**
Calculated result of the example of Figure 43: duration as time interval from start to end.

\[
[t_{\text{min}}, t_{\text{max}}] = [(t_{1\text{min}} + \max (t_{2\text{min}}, t_{3\text{min}}) + t_{4\text{min}}), (t_{1\text{max}} + \max (t_{2\text{max}}, t_{3\text{max}}) + t_{4\text{max}})]
\]

**Figure 48: Duration as time interval from start to end**
List of Figures

Figure 1: Embedded System ........................................................................................................... 13
Figure 2: Nanopositioning and Measuring Machine (NPMM, real images) ........................................ 15
Figure 3: NPMM: Signal- and Data Processing Unit (block structure) ............................................ 17
Figure 4: NPMM: Signal- and Data Processing Unit (hardware structure) ....................................... 18
Figure 5: Trajectory Tracking Control Scheme ............................................................................... 19
Figure 6: Structure of a PI State-Space Controller with Preliminary Filter ....................................... 19
Figure 7: Weighing Machine: Prototype ........................................................................................ 21
Figure 8: Weighing Machine: Prototype schematic ......................................................................... 22
Figure 9: Weighing Machine: Signals ............................................................................................ 24
Figure 10: V-Process Model .......................................................................................................... 27
Figure 11: Function Blocks, two Levels .......................................................................................... 34
Figure 12: Simulink Function Blocks, Highest Level ...................................................................... 39
Figure 13: Simulink Function Blocks, Middle Level ........................................................................ 41
Figure 14: Simulink Function Blocks, Lowest Level (combined, software) ...................................... 43
Figure 15: Design Example Higher Level ........................................................................................ 44
Figure 16: Design Example Information Processing, Higher and Middle Level ............................... 46
Figure 17: Design Example Dynamic Correction, Middle and Lower Level ..................................... 47
Figure 18: Stateflow Function Blocks, Highest Level ................................................................. 50
Figure 19: Stateflow Function Blocks, Lower Level ................................................................. 51
Figure 20: Excerpt from the Generated c++ Program (brown comments not by the coder) .... 54
Figure 21: Stateflow Function Blocks, Lower Level (Figure 19): Connection to the code ......... 55
Figure 22: Excerpt from the generated VHDL program (brown comments not by the coder) .... 57
Figure 23: W-Process Model .................................................................................................. 59
Figure 24: Model in the Loop Test .......................................................................................... 61
Figure 25: Simulation Driven Development ............................................................................. 62
Figure 26: Simulink Testing with Simulable Models ............................................................... 63
Figure 27: Software in the Loop Test ..................................................................................... 64
Figure 28: Hardware in the Loop Test ................................................................................... 65
Figure 29: Hardware in the Loop Test via Interfaces ............................................................... 66
Figure 30: Simplified Use Case Diagram of the Weighing Machine (Figure 8) ......................... 69
Figure 31: Activity Chart of measure_execute ....................................................................... 71
Figure 32: Class Diagram of the Weighing Machine (Figure 8), Highest Level ...................... 73
Figure 33: Class Diagram of the Weighing Machine (Figure 8), information_processing ........ 74
Figure 34: Sequence Chart of the Weighing Machine (Figure 8) (dynamic and static correction, simplified, without digital_filter) ............................................................................... 76
Figure 35: State Chart of f_b_discriminator (similar to Figure 19) ............................................ 78
Figure 36: Class Diagram of the Weighing Machine (Figure 8), Partitioning (simplified) ......... 79
Figure 37: Interaction between objects

Figure 38: Excerpt from C++ Program of the State Chart of Figure 35

Figure 39: Function Block Example for Verification

Figure 40: Petri Net Model of Figure 39

Figure 41: Reachability Graph of PN of Figure 40

Figure 42: Function Block Example for Verification, Corrected

Figure 43: Petri Net Model of Figure 42

Figure 44: Reachability Graph of PN of Figure 42

Figure 45: Time interval calculation: Sequence

Figure 46: Time interval calculation: OR-Structure

Figure 47: Time interval calculation: AND-Structure

Figure 48: Duration as time interval from start to end