NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 18: Data flow analysis framework

2020-03-10

Michael Engel

Overview

- Data-flow analysis
  - partial orders
  - lattices
  - operators

CFGs revisited

- We defined control flow graphs in terms of
  - Operations
  - Basic blocks of operations (that end in jumps)
  - Program points
- As an example, we looked at live variables...
  - (variables that may still be used before their next assignment)

...how they can be found by traversing a control flow graph…

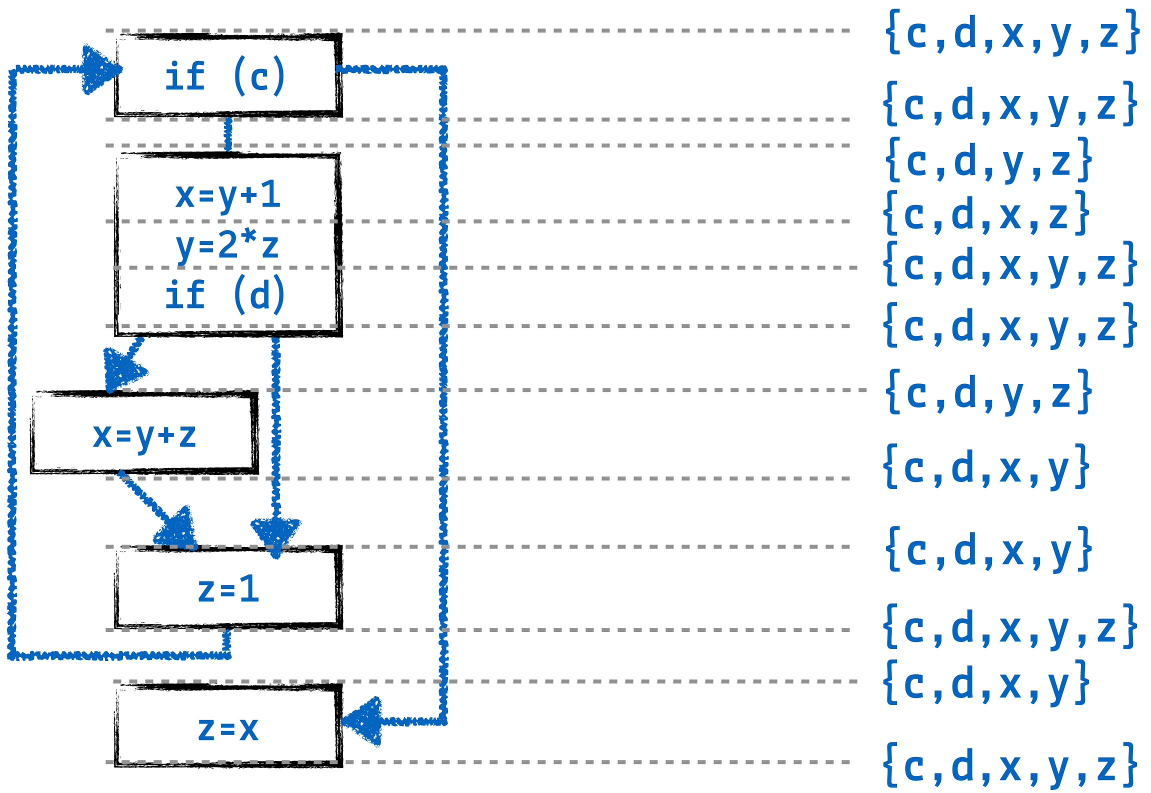
- Collect them in sets attached to program points
- Find out how instructions affect the sets attached to the neighboring program points
- Find out how to handle the sets at points where several control flows meet

…and how the CFG captures every possible execution of the program

(as well as a few impossible ones, to stay on the safe side)

Final result of analyzing liveness

- We have managed to determine the liveness of every variable for every program point



General procedure

- Associate program points with sets that represent the information we are interested in
- Figure out how the sets change
    - As a function of instructions
    - As a function of meeting points between control paths
- Make a safe assumption at an initial point
- Work out the function throughout the graph
- Repeat until the sets stop changing

- But… ***will the sets ever stop changing?***
    - Also, does the analysis get better by repeated application? (we'll talk about this later)

Convergence

- Will this scheme always work?

Some conditions have to hold:

- If the sets have a maximum and minimum possible size **and**
- if the changes we make either **only** add or remove elements

⇒ they will necessarily reach a point where they stop changing

⇒ analysis ends there

- This is obviously a useful property, otherwise the compiler might run forever…

Precision

- How good is the outcome of the analysis?

We call an analysis **precise**:

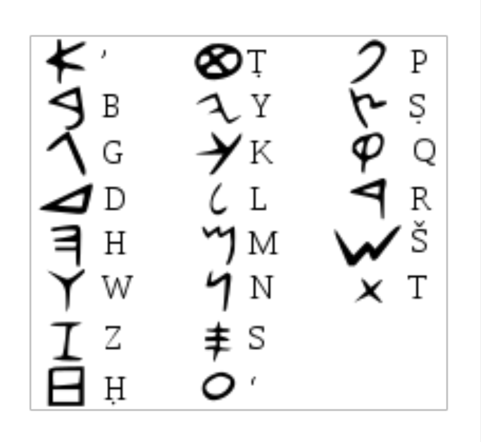
- If it reflects *all* control flows the program *can/will take* **and**
- *none* of the control flows it *will not take*

- A perfectly precise analysis cannot be derived by a computer

- Nevertheless, it is still useful to know if we can assess why quality is lost and how much
    - We need a bit of mathematical background for this…

Sets and orders

- Some sets have a (natural or implied) ***order relation***, e.g.
  - The set of natural numbers: `1 < 2 < 3 < 4 < …`
  - The ordering relation here is "less than", written as '<'
    - Order defined using axioms and a rule system (Peano)
  - Letters in the alphabet: `a < b < c < … < z < æ < ø < å`
    - Lexicographical order by definition (from Phoenician alphabet)
- These are **total orders**
  - they put any pair of set elements in relation to each other
- Other sets do not have an order relation
  - e.g. complex numbers: is `1 < 1i`?
- Some sets let you pick a **consistent** order
  - we write the ordering relation using a ***special comparison operator*** $\sqsubseteq$ to distinguish it from $\leq, \subseteq$



Partial order relations

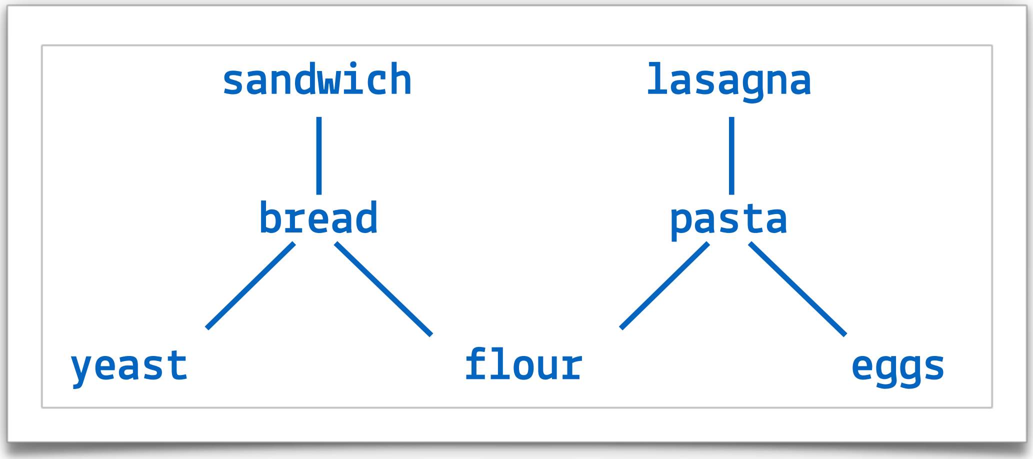
- A ***partial order*** $(P, \sqsubseteq)$ contains
  - a **set** of 'things' (elements) **P**
  - a partial order **relation** $\sqsubseteq$
- Properties of the partial order relation
  - **reflectivity**: $x \sqsubseteq x$
  - **antisymmetry**: if $x \sqsubseteq y$ and $y \sqsubseteq x \Rightarrow x = y$
  - **transitivity**: if $x \sqsubseteq y$ and $y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- For a ***total order*** it must hold that for every x, y: **either** $x \sqsubseteq y$ **or** $y \sqsubseteq x$
- In ***partial orders***, not every pair needs to be comparable

An example

- We can partially order food ingredients as a (stupid?) example
- Let `x⊑y` denote that `x` is an ingredient of `y`
  - `flour ⊑ bread`
  - `flour ⊑ pasta`
  - `eggs  ⊑ pasta`
  - `yeast ⊑ bread`
  - `pasta ⊑ lasagna`
  - `bread ⊑ sandwich`

Visualizing relations: Hasse diagrams

- We can graphically represent the example order (making use of transitivity) like this:

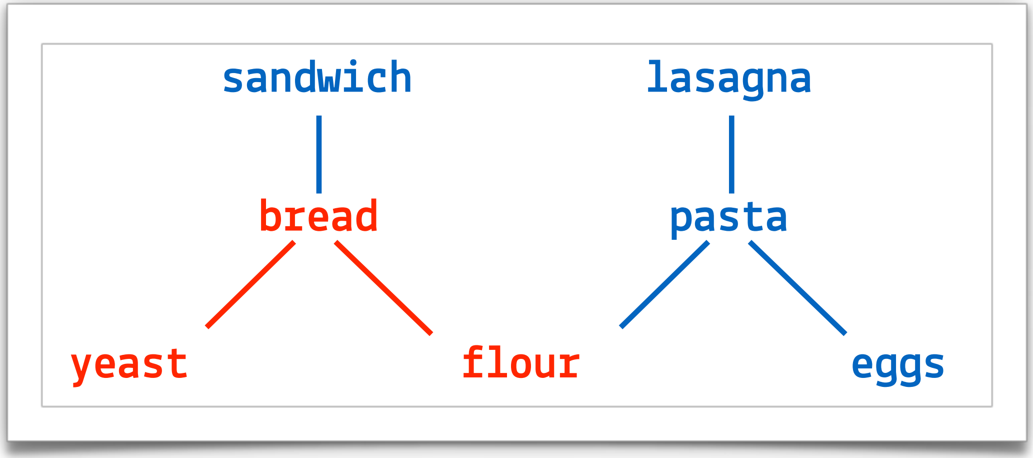


- Here, it is implied that yeast goes into making a sandwich via the bread connection
- There are pairs here which are not comparable using our ingredient relation

Least Upper Bound (LUB)

- The least upper bound of an element pair is the first thing they have in common when **going up** the order

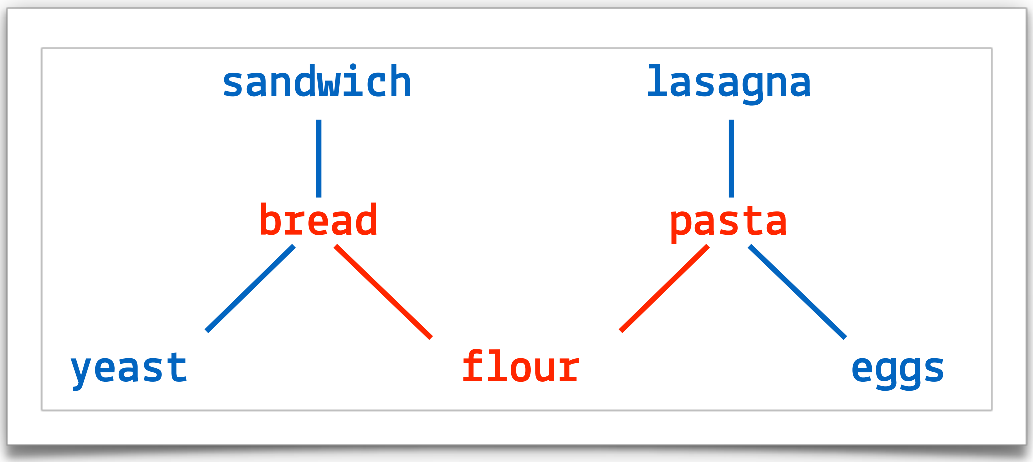
`LUB(yeast, flour) = bread`



Greatest Lower Bound (GLB)

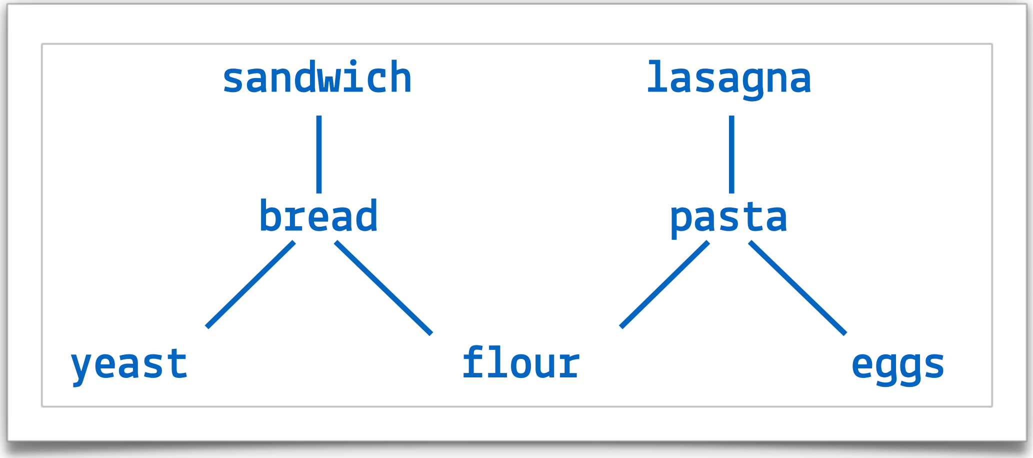
- The greatest lower bound of an element pair is the first thing they have in common when **going down** the order

`GLB(bread, pasta) = flour`



Maximum and minimum

- Partial orders do not necessarily have a unique top or bottom
    - `GLB(yeast, eggs)` does not exist
    - `LUB(sandwich, pasta)` neither



Lattices

- A partial order is a ***lattice*** if ***any finite*** (non-empty) subset has a LUB and a GLB
- *Example:* the natural numbers ordered by '**<**' form a lattice
    - for any finite subset:
        - LUB is the biggest number in the set
        - GLB is the smallest number in the set
- The natural numbers have a **unique bottom element** ($\bot$)
    - it’s the number zero
- They do **not** have a **unique top element** ($\top$)
    - since there are countably infinite many natural numbers
- You can pick **infinite subsets**
    - e.g. even numbers, primes, numbers > 42, …

Complete lattices

- A lattice is called ***complete*** if ***any*** (non-empty) subset has a LUB and a GLB

- These have top ("biggest") and bottom ("smallest") elements
  - For a complete lattice $(L, \sqsubseteq)$
    - $\top = LUB(L)$
    - $\bot = GLB(L)$

- Every finite lattice (lattice with a finite number of elements) is complete

Meet and join relations

- Just to have some symbols that are independent of how we choose the order, define two operators

- "Meet"
    - $x \sqcap y = GLB(x,y)$

- "Join"
    - $x \sqcup y = LUB(x,y)$

- These can be naturally extended to sets of more elements:
    - $x \sqcap y \sqcap z = GLB(GLB(x,y),z)$

Power sets

- Consider the set {a,b,c}

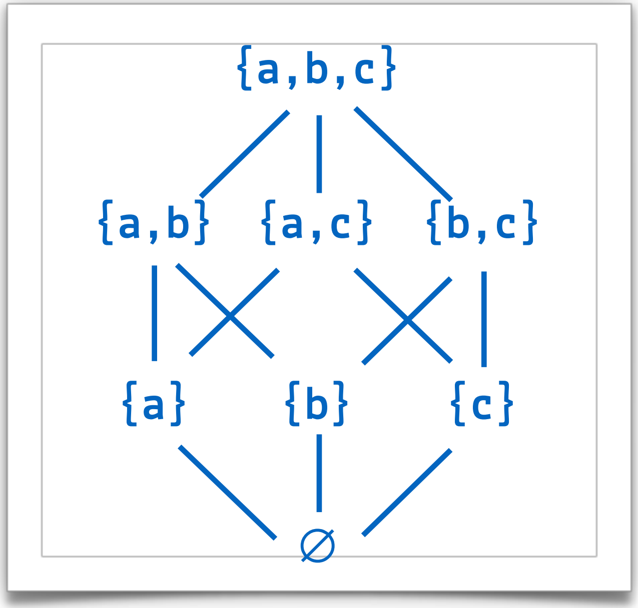
- Its **Cartesian product** with itself is the set of all pairs:
  - {{a,b},{a,c},{b,c}}

- Its **power set** is:
  - {ø,{a},{b},{c},{a,b},{a,c},{b,c},{a,b,c}}

- The power set gives a partial order by the subset relation $\subseteq$

The power set lattice

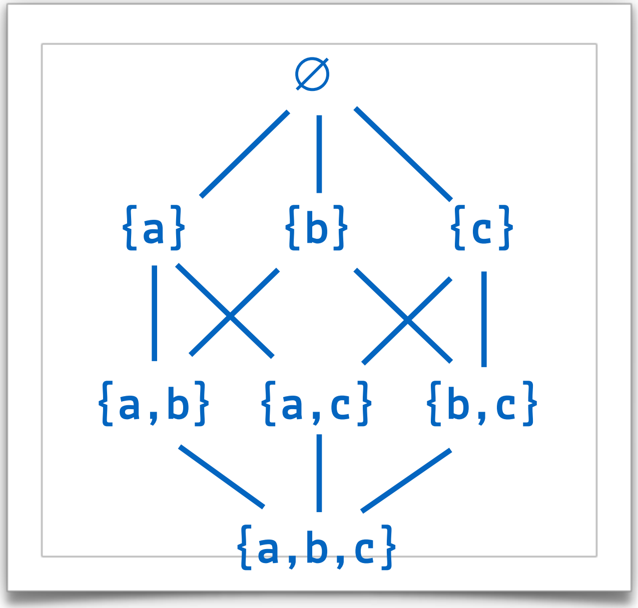
- Ordering relation: $\subseteq$
- Meet operator: $\cap$
- Join operator: $\cup$
- Top: {a,b,c}
- Bottom: $\emptyset$



We can turn it upside down

Just switch the operators around:

- Ordering relation: $\supseteq$
- Meet operator: $\cup$
- Join operator: $\cap$
- Top: $\varnothing$
- Bottom: {a,b,c}



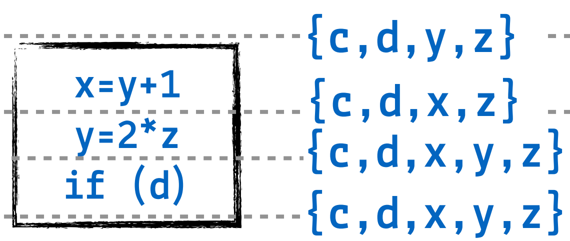
So, how can we use this theory?

Analysis of live variables

- If we take `{a,b,c}` to be the three variables in a short program, every possible choice of live variables corresponds to a point in the power set lattice
- If we can express the effect of statements as a **transfer function** from one place to another in the lattice, we can argue that the set attached to a program point only moves in one direction wrt. the order when it is applied repeatedly
- That means it will either end up at the top, or stop somewhere before it

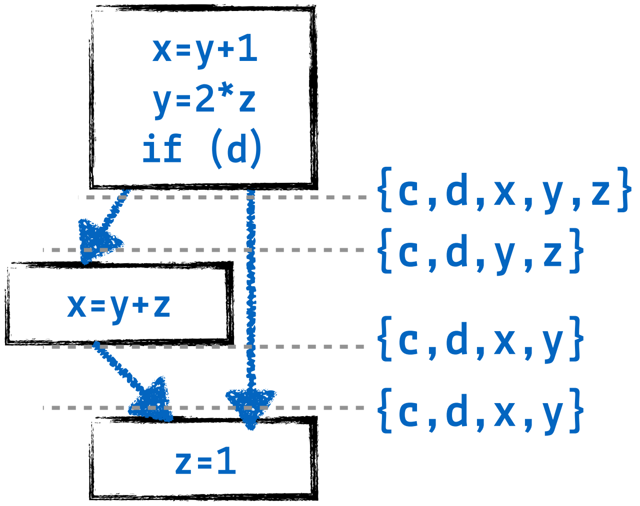
Transfer functions

- This is just a formalization of the idea that the instruction between two program points is a function from one place in the lattice to another
- For an instruction `I`:
    - Forward analysis: `out[I] = F(in[I])`
    - Backward analysis: `in[I] = F(out[I])`
- Accordingly, for basic blocks, the function of a block `B` is simply the nesting of the functions of `B`'s component instructions $I_1 \ldots I_n$:
    - Forward:
      $\mathtt{out[B]} = F_1(F_2(\ldots(F_{n-1}(F_n(\mathtt{in[B]})\ldots)))$
    - Backward:
      $\mathtt{in[B]} = F_1(F_2(\ldots(F_{n-1}(F_n(\mathtt{out[B]})\ldots)))$



Where paths meet again

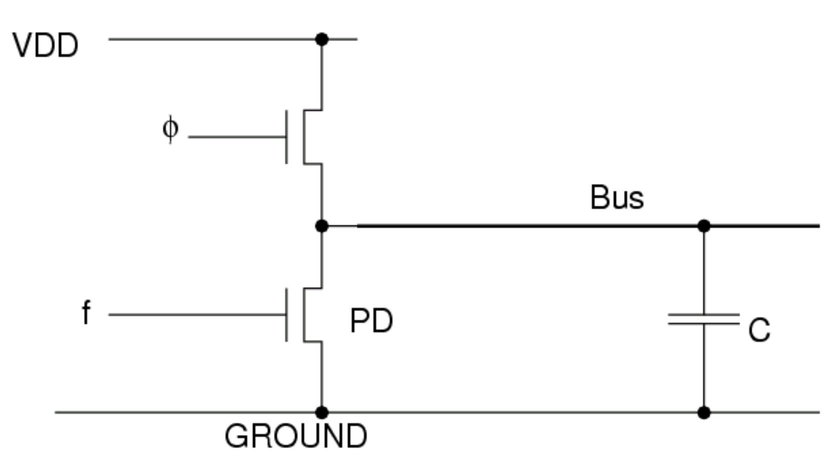
- For the points where multiple control flows intersect:
  - Forward: `in[B] = ⊓ {out[B’] | B’` is a **predecessor** of `B}`
  - Backward: `out[B] = ⊔ {in[B’] | B’` is a **successor** of `B}`
- If we really wanted to, we could use ⊔ instead and reverse the orders
  - With ⊓, transfers in the lattice move toward its bottom
  - With ⊔, transfers in the lattice move toward its top

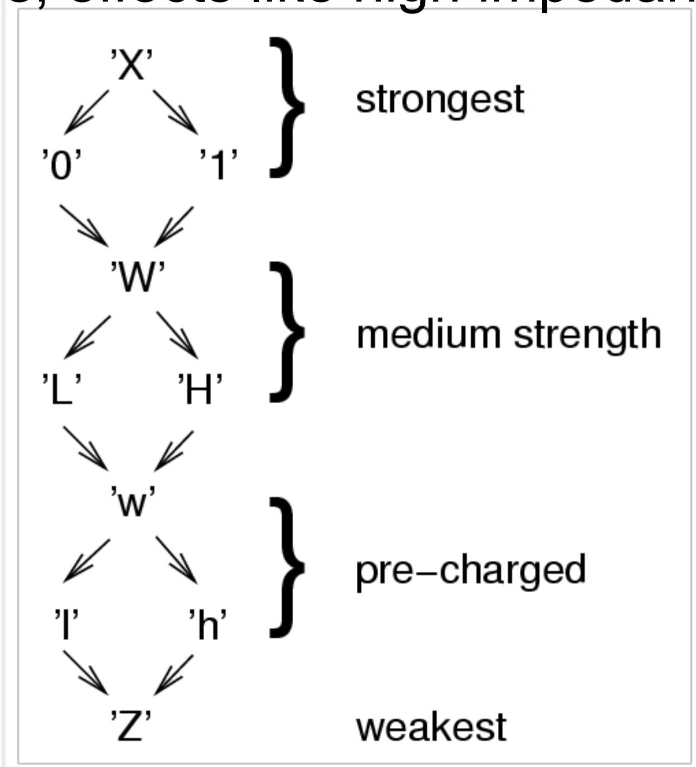


Another application of Hasse diagrams

…no food involved, example from hardware modelling (from [2])

- The VHDL hardware description language allows for the definition of user-defined value sets, e.g. to describe **signal strength**
  - model components such as pull-ups, effects like high impedance





What's next?

- More on data-flow analyses

References

[1] Peano, Giuseppe (1889).
Arithmetices principia, nova methodo exposita
[The principles of arithmetic, presented by a new method], pp. 83–97

[2] Peter Marwedel (2018), Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things, Springer 2018, ISBN 9783319560458