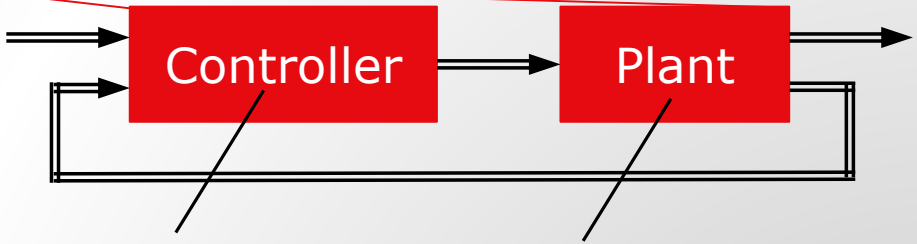
Experiences of using MBD in development of control software for electric drives

Tapani Hyvämäki
Danfoss Drives

Danfoss Drives focuses on **developing, manufacturing and supplying** AC drives



PID controller for: Voltage, current, temperature, power;

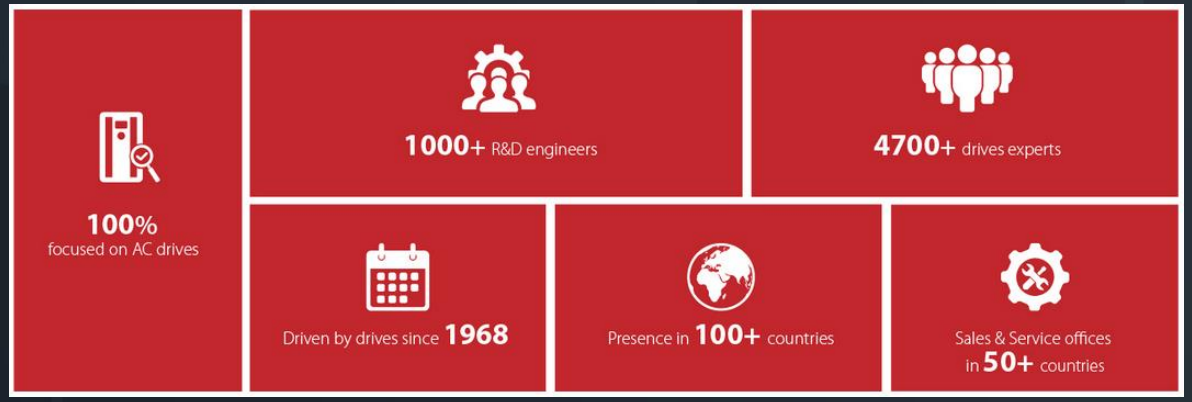
Digital signal processing: Filtering, harmonics;

Estimation, identification, and learning algorithms

running on PLC, CPUs, FPGAs

Power electronics, electric motors, transformers, filters, AC/DC networks, gears, transmission, lifts, elevators, pumps, propellers, fans, conveyor, cranes

Danfoss Drives globally



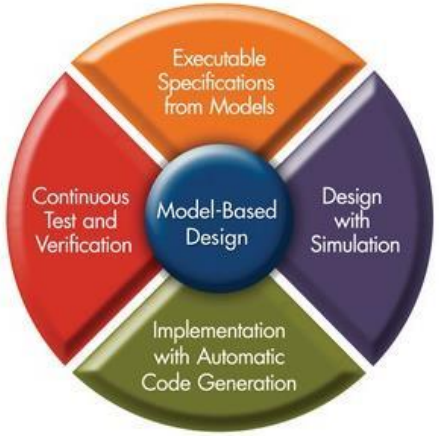
Our history of MBD in brief

- **Our control software developers have used Mathworks products for ~20 years**
  - First they were mainly used as a simulation tool, especially SimPowerSystems
  - Proof-of-concept for control methods

- **The problem of separate simulation models and production code was first solved by importing the production code into Simulink by using s-functions**
  - Improved quality
  - Speeded up development
  - Enabled also first implementations of automated testing by simulation

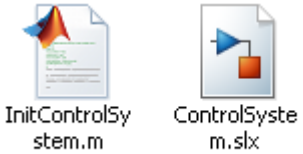
Adopting MBD more comprehensively

- Starting in 2015 our control software development process was shaped more according to MBD
  - Automatic code generation was adopted: C, PLC, HDL
  - Automated testing was adopted more comprehensively: Unit tests, Integration tests/System tests, HIL testing
- The main challenges in the early days were
  - How to deal with large Simulink models
  - How to use version control together with Simulink models
  - How to use continuous integration as part of the development
  - How to deal with asynchronous events in simulink



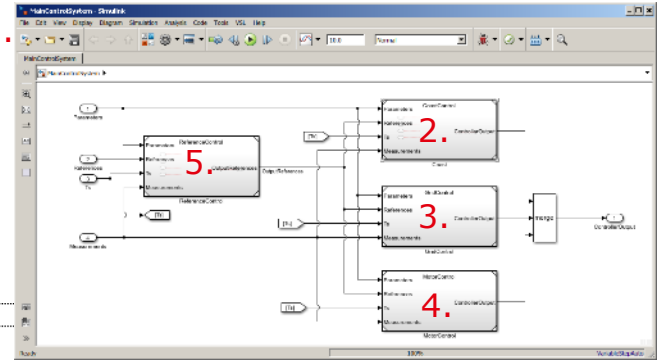
Big Simulink models with big number of developers

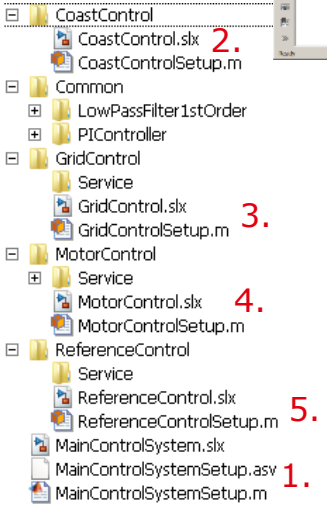
- Quite soon it became evident that the size and amount of Simulink models can grow very big
- Challenges:
  - How to manage the big number of simulink models
  - How to manage the models when dozens of developers are possibly changing them simultaneously
- The worst scenario would be that we have one huge simulink model file with everything defined in it, accompanied with one huge init script
- To properly tackle the problem it was essential to divide the big models into smaller pieces -> **Modularization**



Big Simulink models with big number of developers

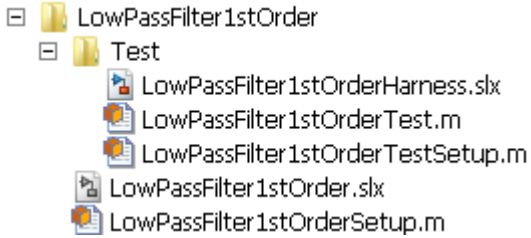
- The simulink model hierarchy - a big tree structure with subsystems, subsubsystems etc.
- First step: split into separate files by using *libraries* and *model reference subsystems*
- Second step: form elementary building blocks (*modules*) and construct the main system from them
- Good characteristics of a **module**:
  - **Reusable**: can be used multiple times
  - **Self-contained**: contains the information of the resources it requires (parameters, paths, busses, enums, configuration, etc.)
  - **Unit-tested** (including coverage analysis)





Big Simulink models with big number of developers

- A minimal module consists the files shown on the right
- The Setup file defines and links all the resources together
  - Adds folders to matlab search path
  - Defines Bus objects
  - Calls the setup files of submodules
- The Setup file is defined as a matlab class that
  - Allocates resources upon initiation
  - Frees the resources when destructed (cleans up the workspace)
- The unit tests defined in the *Test* folder tests only the functionality related to this module
  - Submodules have their own unit tests
  - Implemented with Matlab Unit Test Framework



Continous integration and version control with Simulink models

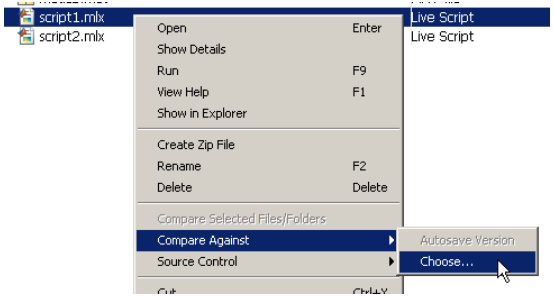
- Conforming the MBD workflow with our version control system and CI posed two big challenges:
  - How to do version controlling with binary files?
  - How to do automatic server builds with Mathworks tools when adopting CI?

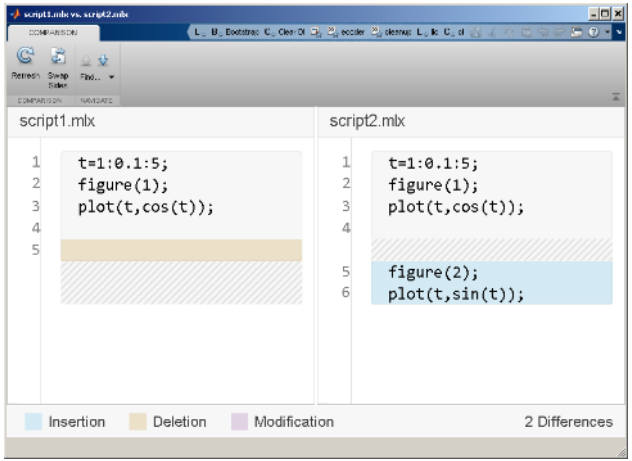
Version control with Simulink models

- Version control systems (Git/SVN) do not handle binary files as ’wisely’ as ascii files
  - Increased disk space requirement
  - Merge/rebase conflicts -> extra manual work
- We decided to avoid using binary files when possible, instead the source files are stored in ascii format into version control
  - Documents, mex files etc. are generated from their source files
  - For data, configurations etc. we use m-files or more generic formats (json/xml) if possible instead of mat files

Version control with Simulink models

- When binary files cannot be avoided reasonably, we use Matlab version control features
  - Solving of *merge conflicts* is well supported for many filetypes: .slx, .mlx, .mat
  - Solving git *rebase conflicts* are not equally well supported
    - In some cases it is a drawback since conflicts needs to be solved fully manually
    - Still the Matlab *file comparison tool* can be of much help



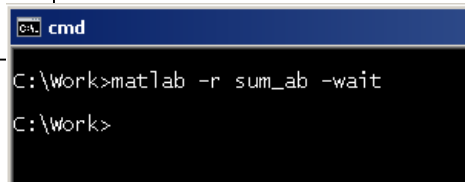


Continuous integration with Simulink models

- The initial idea was to run unit tests, generate code etc. in similar way as we build software by typing ”make all” or ”make *target*”
- Continous integration systems commonly just runs such commands automatically
- The command line use of matlab.exe turned out to be fairly limited:
  - The overhead of Matlab start-up can be significant
  - No output is printed to console -> hard to debug
- *Matlab API for Python* provided a more flexible solution

sum_ab.m

```
a=3;
b=4;
c = a + b;
save('result.txt','c','-ascii');
exit;
```



- Execution of this simple script takes 15 seconds on my PC
- Running a simple simulation or code generation would take some more since the Simulink start-up time is added

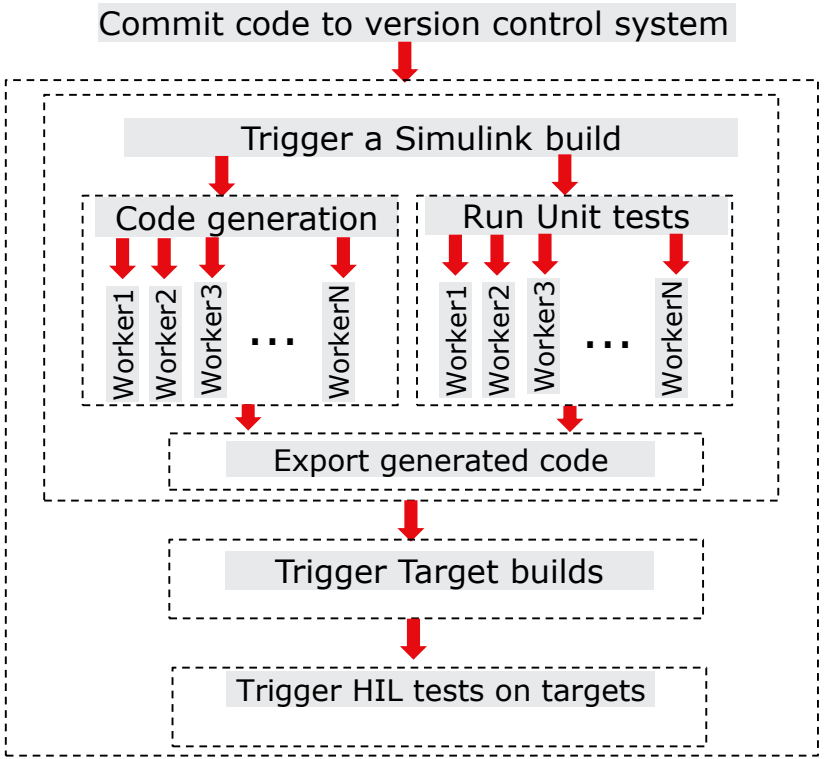
Continuous integration with Simulink models

- Launching the Matlab build jobs from Python had following advantages
  - Scheduling of build jobs was more flexible by using Python threads
    - Schedule the jobs to be done: run unit tests, generate code, generate configurations etc.
    - Parallelizing builds: run one job/cpu core to reduce total build times
- First we used a open-source Pyhon based build tool called Waf
- Later we implemented a custom build tool in Matlab that exactly matched our needs
  - Scheduling was also moved into Matlab/Parallel Computing Toolbox

```
BuildTool('Name1','Value1','Name2','Value2',...)

Input aguments ('Name' – Value):

    'Rule' – Build task to be done specified as
                        cell array of characters
              e.g. {'codegen','unittest',..}

    'Target' – Model names specified as cell array
                        of characters, using wild cards
                        *, **, ?, is supported
               e.g. {'ModelName1', 'ModelName2', ...}

    'NumWorkers' – Number of parallel workers
                        specified as double

    ...

Example:

BuilTool('Rule',{'codegen'},'Target',{'MainCtrlSystem'})
```

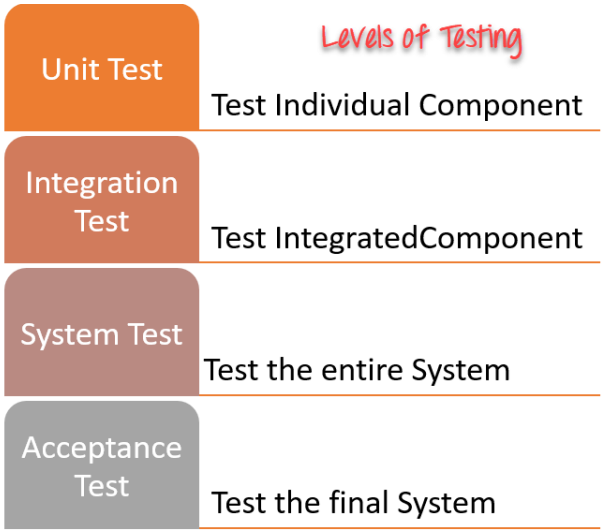
Continuous integration with Simulink models

- A rough workflow of the CI server build pipeline is shown on the right
  - If one of the build steps fail the subsequent steps are not run
  - When the build pipeline is succesfully run, the code has passed the validation and is ready to be merged into the develop branch



Verification by Simulation, HIL and more

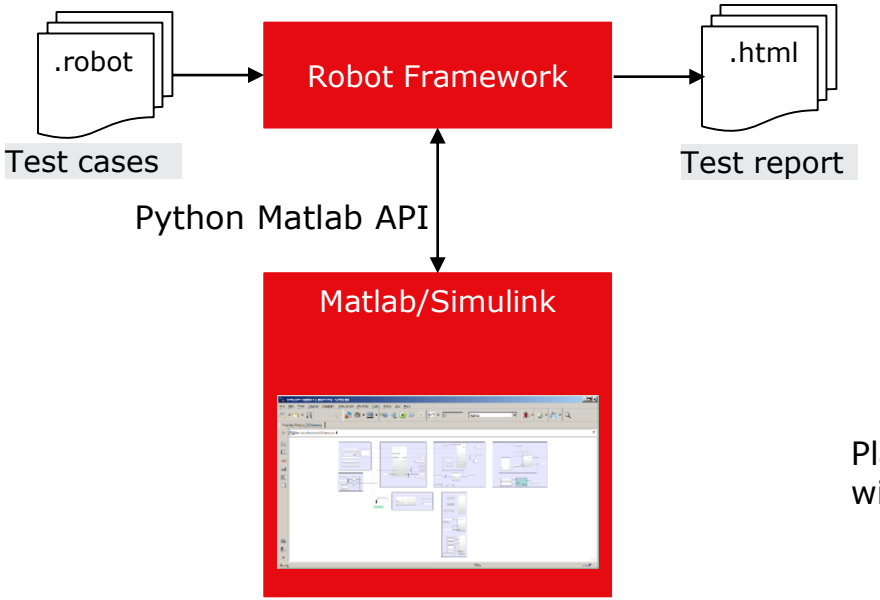
- Distinction of different testing levels is often challenging
  - In our testing terminology the Integration Testing and System Testing are used interchangeably
- System Testing is done on few different kind of setups
  - **Simulink System tests**: the production code is ran togeher with a plant simulation model. The whole simulation is ran in Simulink.
  - **Light HIL tests**: The generated production code runs on target HW that is coupled with a light weight hardware simulator (real-time). The hw simulator is capable of simulating a limited accuracy dynamic model.
  - **Heavy HIL tests**: The generated production code runs on target HW that is coupled with a fully featured hw simulator (real-time)
  - **Real Motor/Grid tests**: The production code runs on target hw that is connected to actual physical environment.



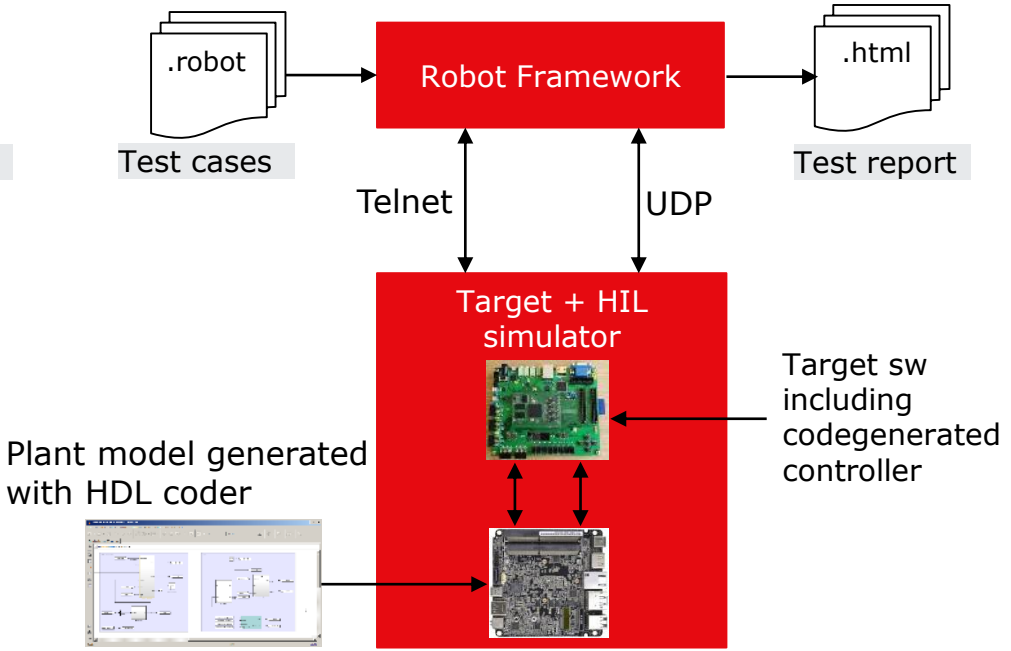
Source: https://www.guru99.com/levels-of-testing.html

Verification by Simulation, HIL and more

Simulink System tests: PC (not real time)



Light HIL tests: PC + simulation target (real-time)



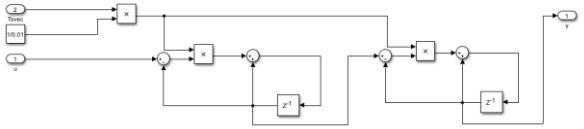
Asynchronous events in Simulink – advanced scheduling

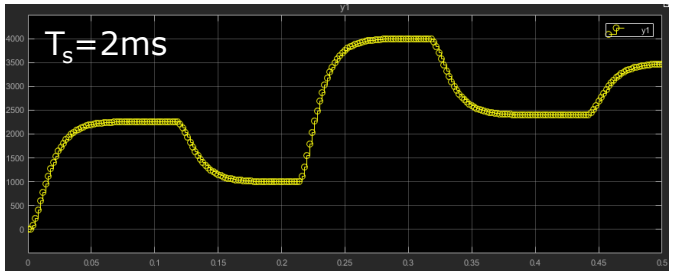
- Power electronics systems operate at fairly high frequencies
- Measurements sampling and control of semiconductor switches at rate of 10-1000MHz
- Control algorithms executed at rate of 1-20kHz
  - The execution rate varies steplessly
- Being able to simulate such system is essential to capture e.g. fast events and harmonics
- How to achieve a sufficient simulation accuracy and performance?

Asynchronous events in Simulink – advanced scheduling

- The discrete-time systems in Simulink are executed either
  - at fixed sample interval: T_s
  - at multiple of a fixed base sample interval: $n \cdot T_s$
- The sample time T_s cannot be changed during simulation!

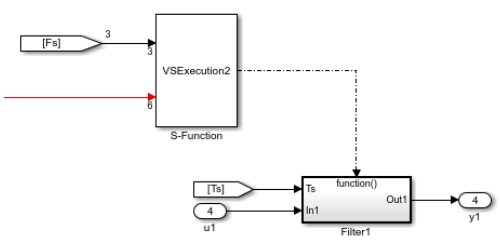
- Consider e.g. a discrete-time filter that should execute with a varying sample-rate
  - Quite common and trivial case in embedded systems
  - But how to simulate such event-based systems in Simulink?
  - One solution is to use an s-function with varying sample-time feature

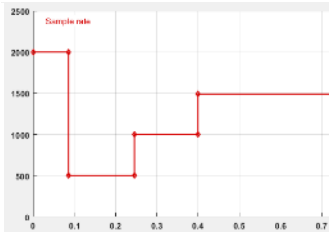


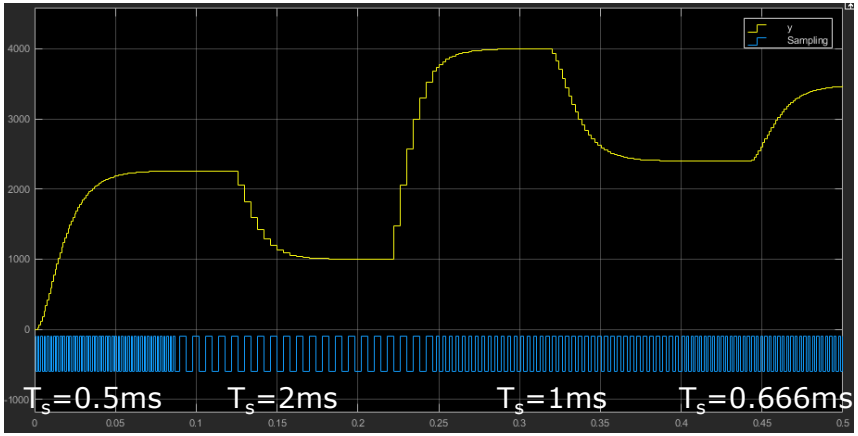


Asynchronous events in Simulink – advanced scheduling

- The s-function executes asynchronously and defines its next execution time instant based on some of its inputs (Fs)
  - The actual algorithm is executed as function call subsystem
  - For example filter algorithms need to take into account the changing sample time Ts
- Code generation from such model is not straight-forward (doesn't happen with a push of the build button)
- Requires separate code generation for each function call subsystem
  - Requires proper modularization to manage this







Danfoss

ENGINEERING
TOMORROW