Using model-based design to implement the motor control logic of a fully electric downhole flow-control valve

Michel Gardes, Schlumberger

MATLAB Energy Conference 2021, November 16-17

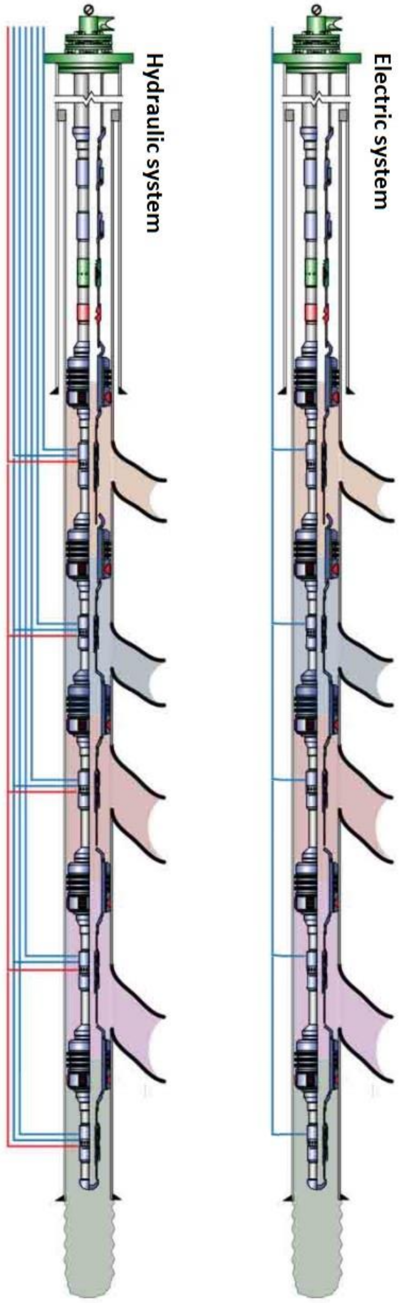
Agenda

- Project presentation
- Simulink model
- Motor control logic
- Test harness
- Code generation
- CPU load concern
- Improvements in the model and in the generated code
- Results
- Conclusion

Project presentation

Downhole flow control valves

- Control the flow of a producing zone
- High flow-rates: up to 60 000 bbl / day
- Hydraulic technology
  - Installation complexity: 1 hydraulic line per valve + 1 optional return line
    - Limits the number of valves in a well (max 5)
    - Several feedthroughs, potential leakage paths
  - Indexing valves: fully closing requires fully opening first
- Electric technology
  - Simpler installation: 1 single electrical line for all the valves in the well
    - Only one feedthrough running through the packers
    - Increased reliability
  - Versatility: the valve can be actuated from any position to any position



Schlumberger

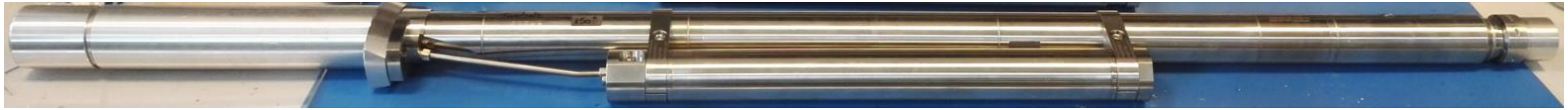
Electric downhole flow-control valve

Motor control electronics must control and drive an Electro-Mechanical Actuator (EMA):

- 3-phase Permanent Magnet Synchronous Motor (PMSM)
- Gear box
- Roller screw

Position measurement: resolver mounted on the motor shaft

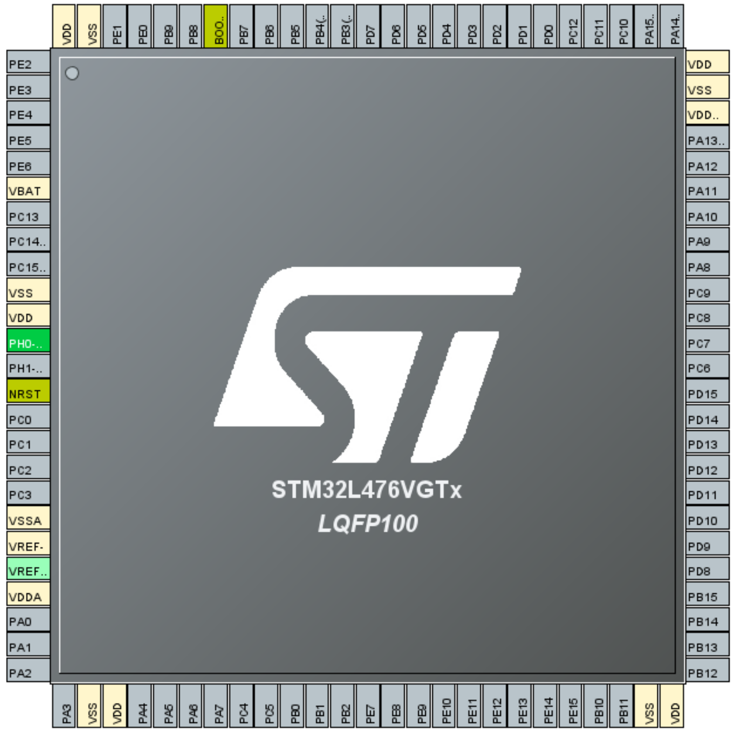
- The EMA must be able to apply the maximal force at the very beginning of an actuation
- Sensored control more suitable for applications requiring high torque at zero / low speed



STM32L4

STM32L476VG from ST microelectronics (ARM Cortex-M4 processor): most powerful microprocessor qualified for high-temperature permanent applications

- Low power
- Advanced-control timers dedicated to motor control applications
  - Complementary PWM outputs with programmable dead-time
  - Break inputs for safety purposes
- Dual-mode ADC (analog-to-digital converter) operation for simultaneous acquisition of phase current values
- Floating Point Unit for higher performance



Reasons for using model-based design

Late availability of the required motor (custom development for high temperature applications)

- Need to develop and test the control algorithm before receiving the motor

Different motor models to be tested

- Flexibility to adapt and tune the control algorithm for different motor types

MBD enables more effective troubleshooting

- The objective is to troubleshoot with the simulation instead of analyzing the failure on the hardware

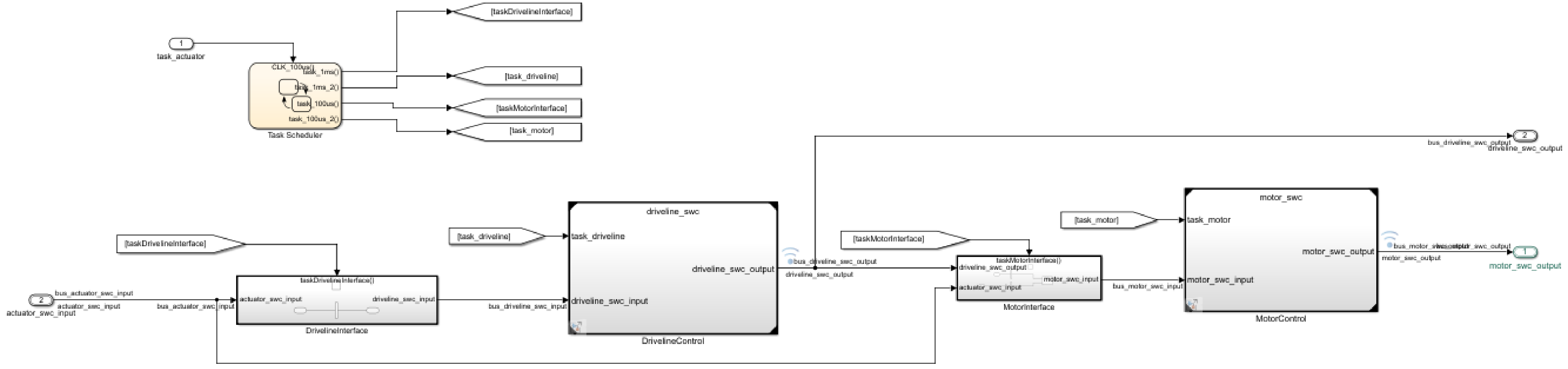
Ability to simulate corner cases

Motor control method

Field Oriented Control (FOC)

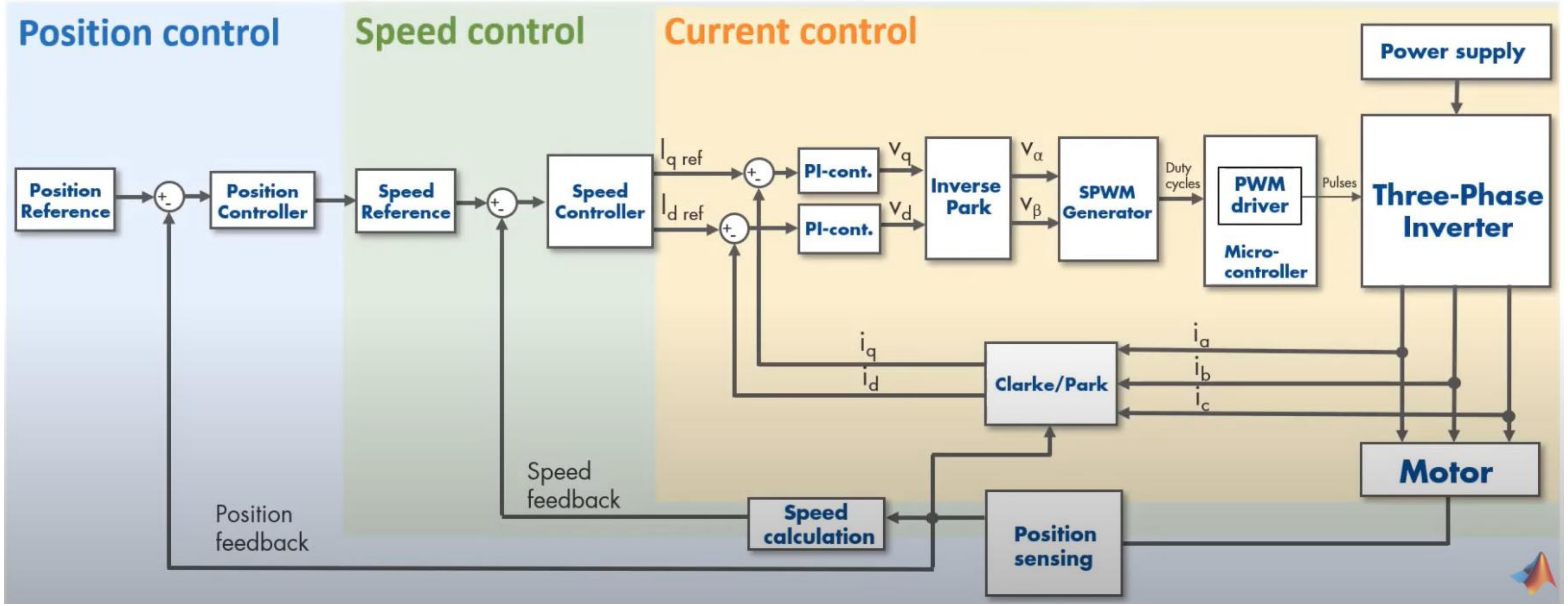
- High dynamic performance
- Full torque at zero speed

Developed in MATLAB / Simulink in collaboration with MathWorks

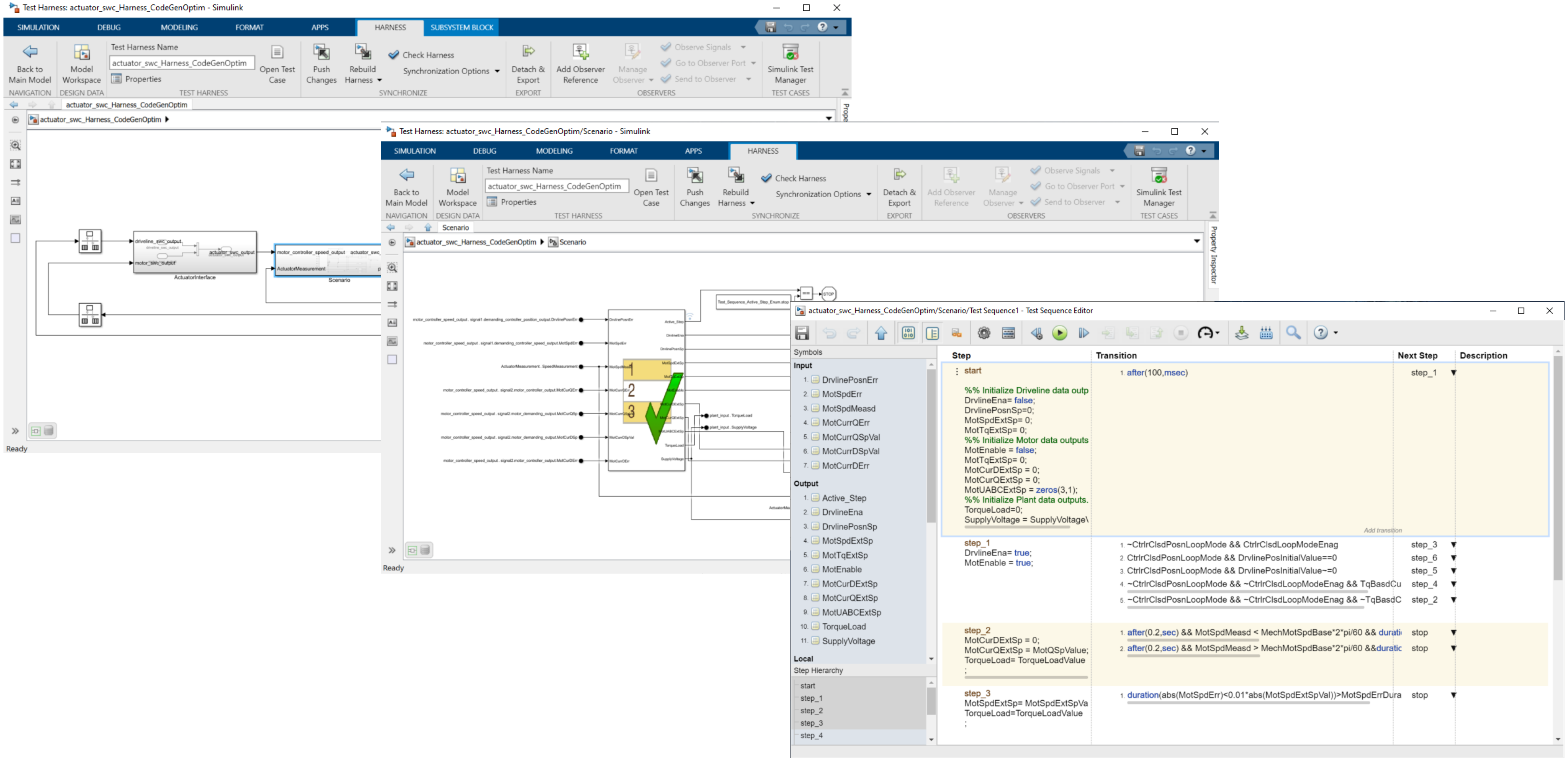


Control loops

- **Position control loop**
  - Position setpoint
  - Default operating mode
- **Speed control loop**
  - Speed OR torque setpoint
- **Current control loop**
  - PI controller for maximal Iq
  - PI controller for Id = 0



Test Harness



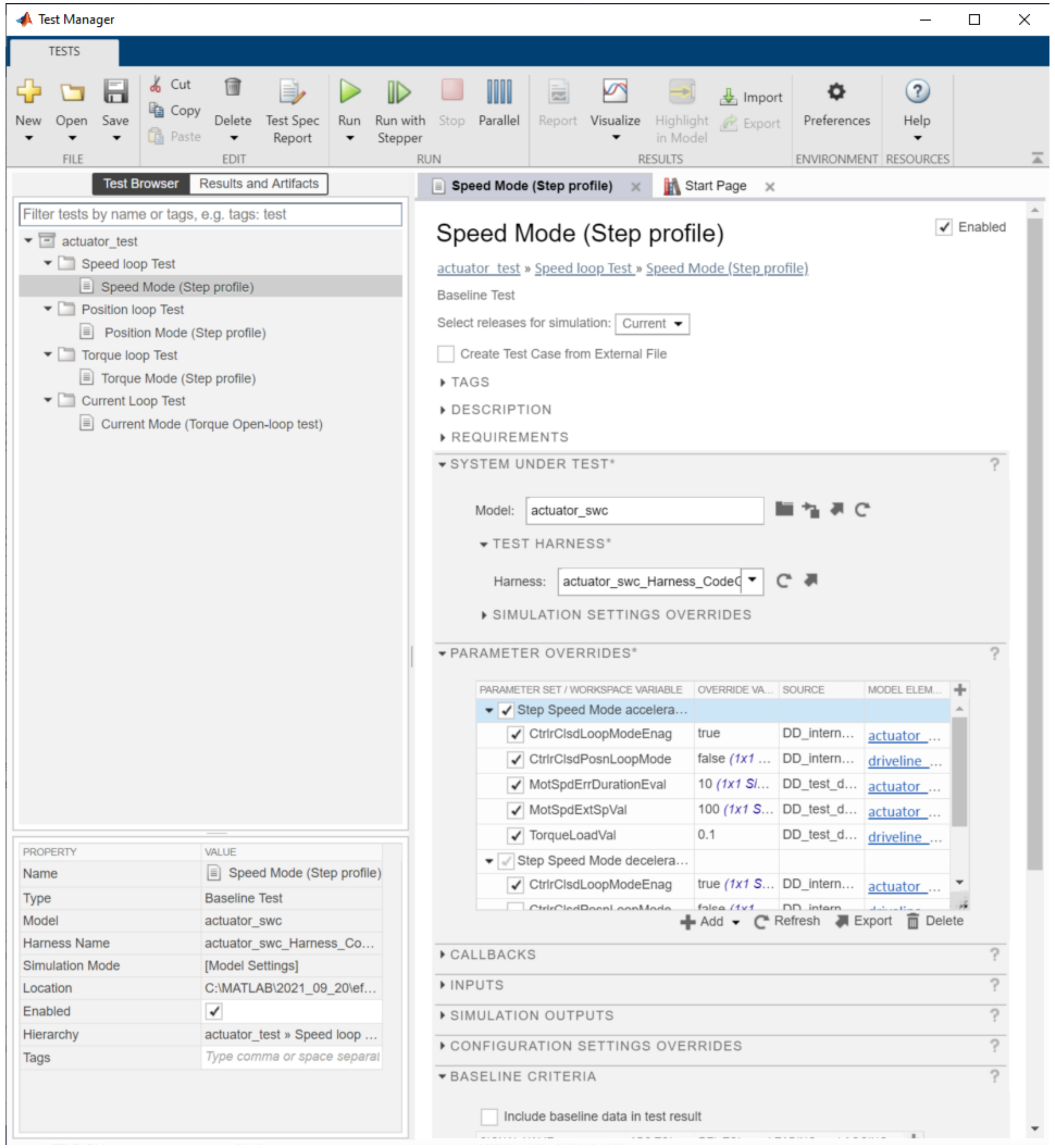
Schlumberger

Simulink Test Manager

New approach for the team

Much easier to launch tests and keep track of results

Erratic behaviour when exploring results ⚠



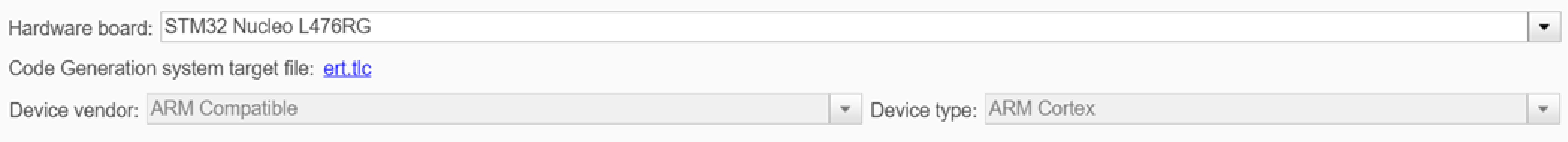
Schlumberger

Code generation

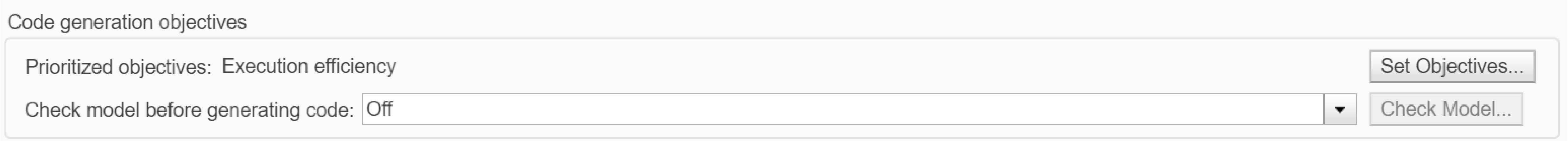
C code generation with MATLAB Embedded Coder

- Control algorithm only

Optimized for the STM32 microprocessor



Objective: execution efficiency = lowest CPU load



CPU load concern

Preliminary analysis based on data from ST, TI, Renesas

- Estimation: 1000 to 1300 cycles for sensored FOC at 10 kHz

PIL (Processor in the Loop) tests with Simulink

- Simulink report: 60% at 80 MHz, i.e. 4800 cycles at 10 kHz
- Lab measurements: 61.6% at 77.4 MHz, i.e. 4772 cycles at 10 kHz
  - With compiler optimization for speed

Unsatisfactory results

Lower CPU frequency targeted

Post-generation code enhancements for maximal performance

Model parameters and signals

Simulink parameters and signals defined as auto are converted to 64-bit values in the generated code

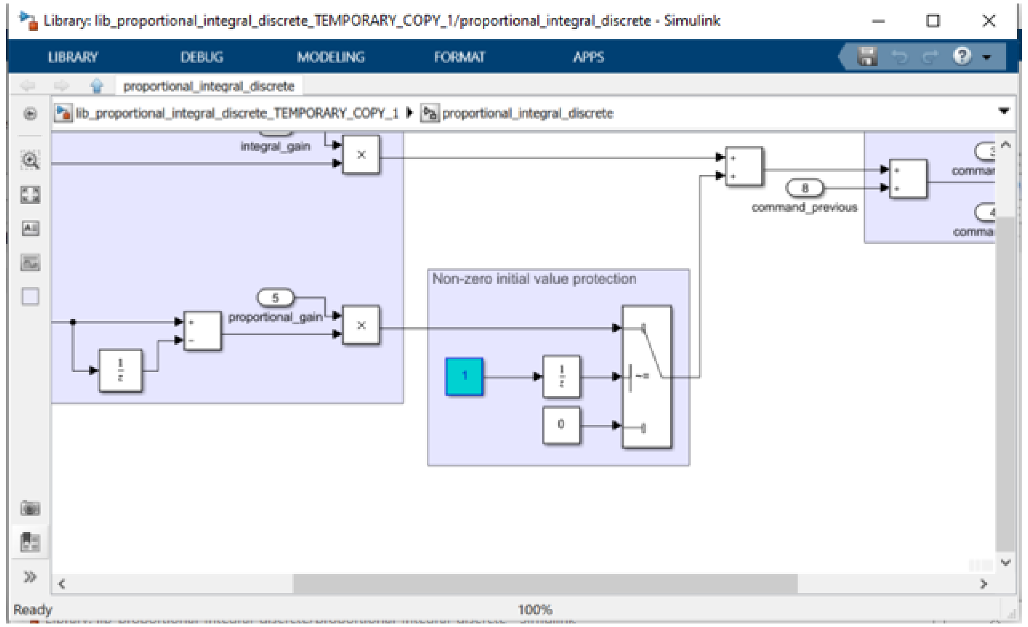
- Must be defined as *single* when using a 32-bit FPU

Name	Status	Value	DataType	Dimensions	Complexity	Min	Max	Unit	StorageClass	
bus_motor_demanding_output										DD_interna
MechMotSpdBase		[ ]	single	[0 0]	real	[ ]	[ ]	rpm	Auto	DD_interna
MotCurDExtSpFiltd			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurDSp			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurDSpMax		[ ]	single	[0 0]	real	[ ]	[ ]	A	Auto	DD_interna
MotCurDSpMin		[ ]	single	[0 0]	real	[ ]	[ ]	A	Auto	DD_interna
MotCurDSpTqBasd			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurQExtSpFiltd			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurQSp			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurQSpMax		[ ]	single	[0 0]	real	[ ]	[ ]	A	Auto	DD_interna
MotCurQSpMin		[ ]	single	[0 0]	real	[ ]	[ ]	A	Auto	DD_interna
MotCurQSpTqBasd			single	-1	auto	[ ]	[ ]	A	Auto	DD_interna
MotCurSpMax		[ ]	single	[0 0]	real	[ ]	[ ]	A	Auto	DD_interna
motor_demanding_output			Bus: bus_motor_demanding_output	-1	auto	[ ]	[ ]		Auto	DD_interna
MotTqToMotCur		[ ]	single	[0 0]	real	[ ]	[ ]	A/N/m	Auto	DD_interna
PermanentFlxLnkg		[ ]	single	[0 0]	real	[ ]	[ ]	Wb	Auto	DD_interna
T_CURRENT_DEMANDING		0.0001	single	[1 1]	real	[ ]	[ ]		Auto	DD_interna
TqBasdCurSpEnag		[ ]	boolean	[0 0]	real	[ ]	[ ]		Model default	DD_interna

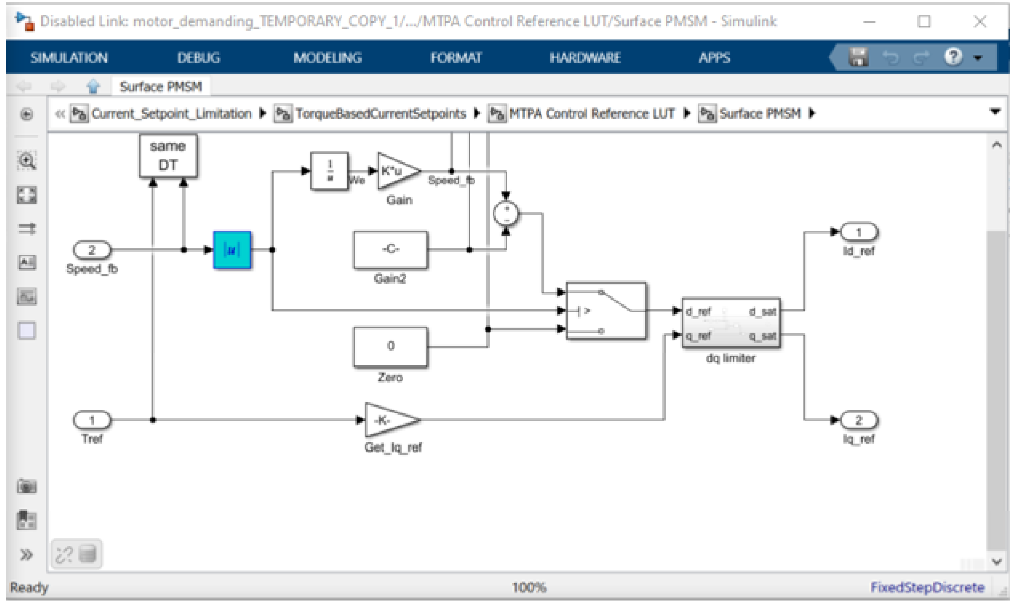
Schlumberger

Simulink blocks – *constant* and *abs*

The data type of the *constant* block output must be set to *single*

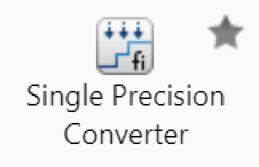


The *abs* block output data type must be set to *single*

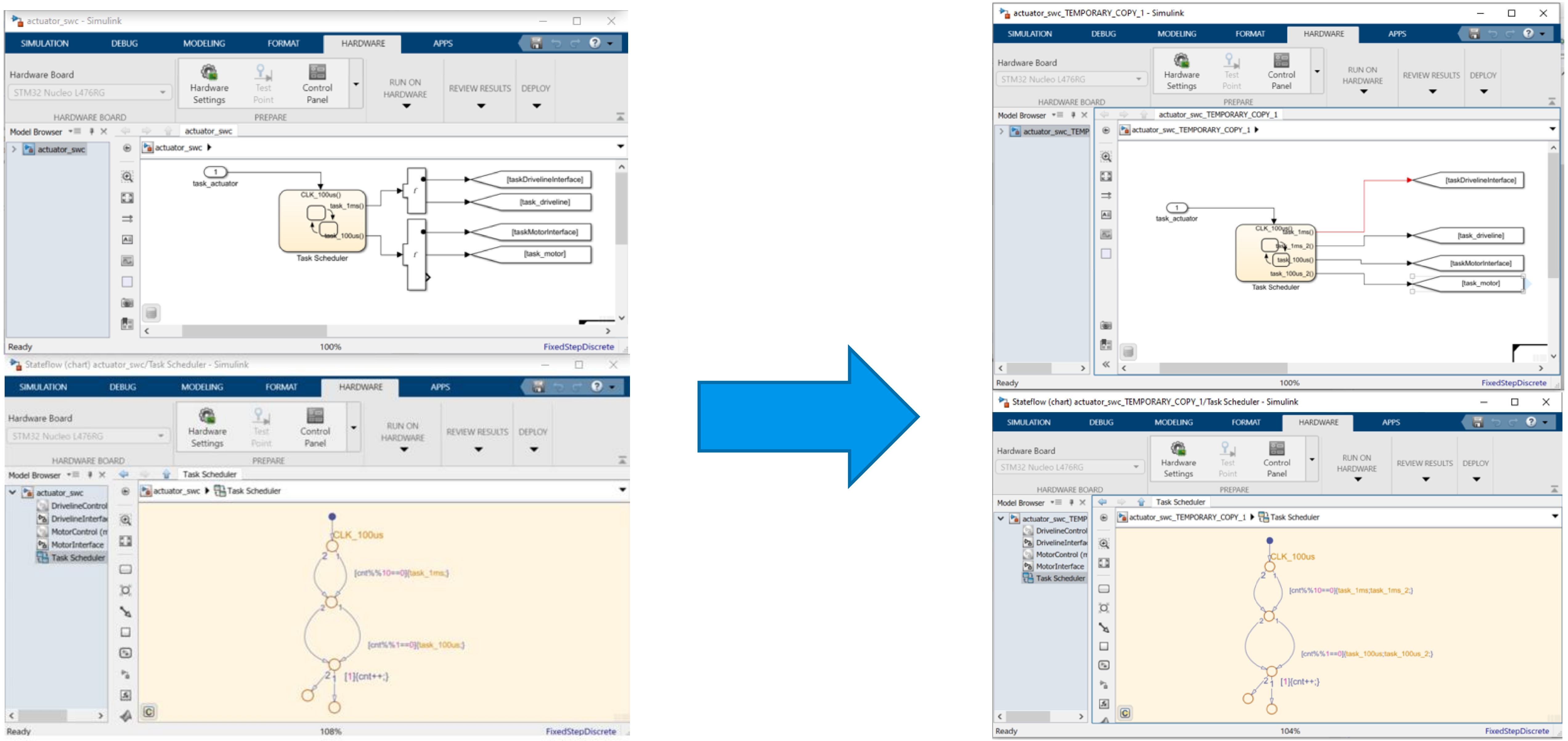


Schlumberger

Simulink blocks – *function call split*



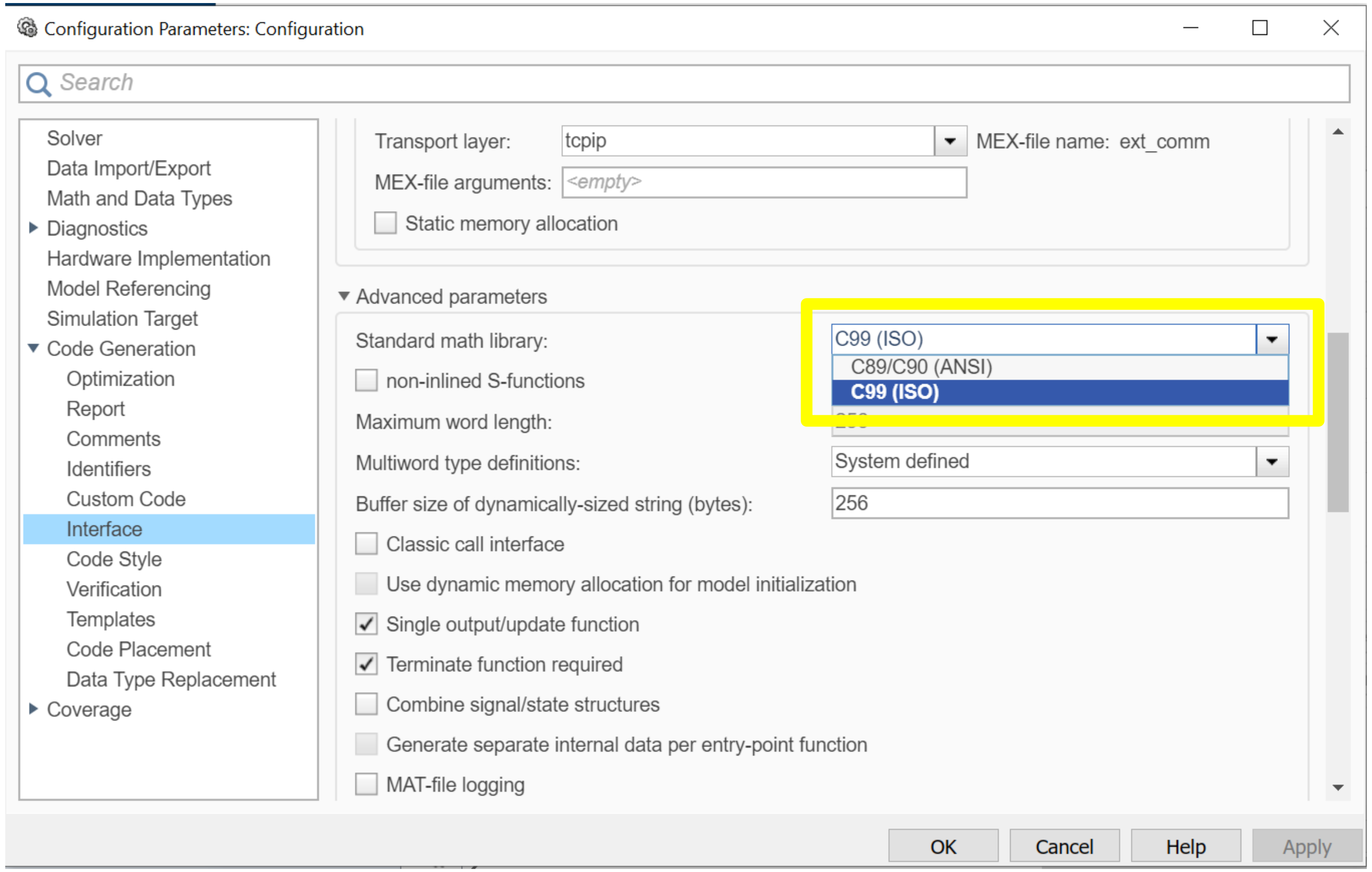
Function call splits are not 32-bit compatible and must be replaced by as many individual function calls as required



Code generation configuration – C standard

C standards: C89/90 versus C99

May have considerable impact on the implementation of floating-point operations



Code generation configuration – C standard

C89/90

```
    /* End of Switch: '<S6>/Switch2' */
  } else {
    /* Abs: '<S8>/Abs' */
    rtb_Abs = (real32_T)fabs((real_T)rtu_motor_swc_input->MechMotSpdMeasd);
```

```
if (rtb_convert_pu < 0.0F) {
  /* Outputs for IfAction SubSystem: '<S7>/If Action Subsystem' incorporates:
   *  ActionPort: '<S9>/Action Port'
   */
  rtb_convert_pu -=
(real32_T)(int16_T)(real32_T)floor((real_T)rtb_convert_pu);
```

C99

```
    /* End of Switch: '<S6>/Switch2' */
  } else {
    /* Abs: '<S8>/Abs' */
    rtb_Abs = fabsf(rtu_motor_swc_input->MechMotSpdMeasd);
```

```
if (rtb_convert_pu < 0.0F) {
  /* Outputs for IfAction SubSystem: '<S7>/If Action Subsystem' incorporates:
   *  ActionPort: '<S9>/Action Port'
   */
  rtb_convert_pu -= floorf(rtb_convert_pu);
```


Schlumberger

Code generation configuration – C standard

C89/90

```
/* MinMax: '<S16>/Max1' */
if (rtb_MotUABCSpRaw[0] < rtb_MotUABCSpRaw[1]) {
  rtb_Add3 = rtb_MotUABCSpRaw[0];
} else {
  rtb_Add3 = rtb_MotUABCSpRaw[1];
}

/* MinMax: '<S16>/Max' */
if (rtb_MotUABCSpRaw[0] > rtb_MotUABCSpRaw[1]) {
  rtb_MotEDObsrvd = rtb_MotUABCSpRaw[0];
} else {
  rtb_MotEDObsrvd = rtb_MotUABCSpRaw[1];
}

/* MinMax: '<S16>/Max1' */
if (rtb_Add3 >= rtb_MotUABCSpRaw[2]) {
  rtb_Add3 = rtb_MotUABCSpRaw[2];
}

/* MinMax: '<S16>/Max' */
if (rtb_MotEDObsrvd <= rtb_MotUABCSpRaw[2]) {
  rtb_MotEDObsrvd = rtb_MotUABCSpRaw[2];
}
/* Product: '<S16>/Product' incorporates:
 *  Constant: '<S16>/Constant'
 *  MinMax: '<S16>/Max'
 *  MinMax: '<S16>/Max1'
 *  Sum: '<S16>/Add'
 */
rtb_Add3 = (rtb_MotEDObsrvd + rtb_Add3) * 0.5F;

rtb_MotUABCSpRaw[0] -= rtb_Add3;
rtb_MotUABCSpRaw[1] -= rtb_Add3;
rtb_MotUABCSpRaw[2] -= rtb_Add3;
}
```

C99

```
/* Product: '<S16>/Product' incorporates:
 *  Constant: '<S16>/Constant'
 *  MinMax: '<S16>/Max'
 *  MinMax: '<S16>/Max1'
 *  Sum: '<S16>/Add'
 */
rtb_Add3 = (fmaxf(fmaxf(rtb_MotUABCSpRaw[0], rtb_MotUABCSpRaw[1]),
         rtb_MotUABCSpRaw[2]) + fminf(fminf(rtb_MotUABCSpRaw[0],
  rtb_MotUABCSpRaw[1]), rtb_MotUABCSpRaw[2])) * 0.5F;
rtb_MotUABCSpRaw[0] -= rtb_Add3;
rtb_MotUABCSpRaw[1] -= rtb_Add3;
rtb_MotUABCSpRaw[2] -= rtb_Add3;
}
```

Code generation configuration – C standard

Generation of code for both standards: C89/90 and C99

Best CPU load is achieved with a blend of C89/90 and C99

Results

CPU load of generated code for the control logic after improvement

4772 cycles → 1498 cycles

No compiler optimization

Lower clock frequency

Conclusion

Implementing Model-Based Design with the help of MathWorks consulting team allowed us to achieve challenging deadlines

Code generation is very useful, but might require manual intervention for optimal performance

Acknowledgements

Wided Zine, MathWorks

Zakaria Mahi, MathWorks

Denis Heliot, Schlumberger

Q&A