



Model Interpretation for an AUTOSAR compliant Engine Control Function

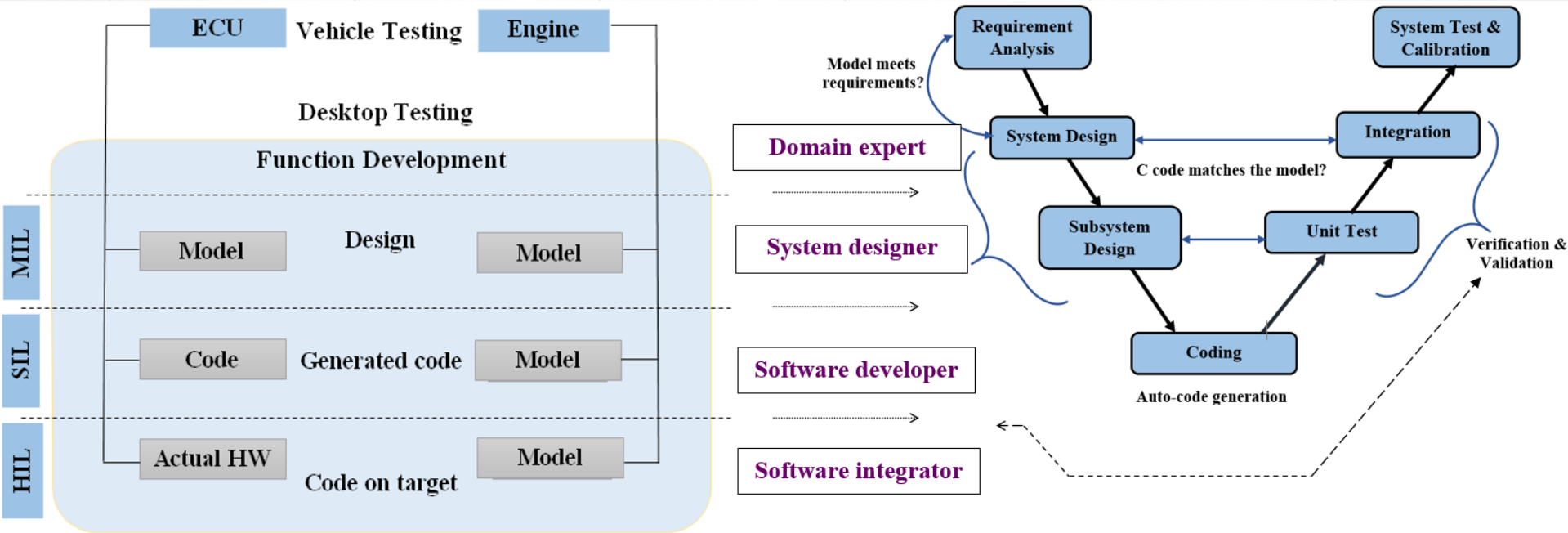
Sakthivel M. SUNDHARAM, Sebastian ALTMAYER, Nicolas NAVET,
University of Luxembourg



(WATERS 2016)
Toulouse, France, July 05, 2016



State of the art Practice



Model as the main artifact to develop the embedded software

Generative MBD – e.g., MLSL , ASCET-MD etc.



Code, other artifacts automatically generated from model
Code → binary → run on target hardware



Interpreted MBD

- Direct interpretation of design models using *interpretation engine* running on top of target
- No (optional) code, other artifacts generated
- No commercially available interpreted MBDs
- So not practiced in industrial embedded software development life cycle
- But interpretation-based runtime environments are proven (track-record) to be applied



Paper discusses...

- ***Interpreted MBD*** to an industry case study to investigate it's applicability to embedded software development cycle



Interpreted MBD based embedded software development life cycle (proposal)

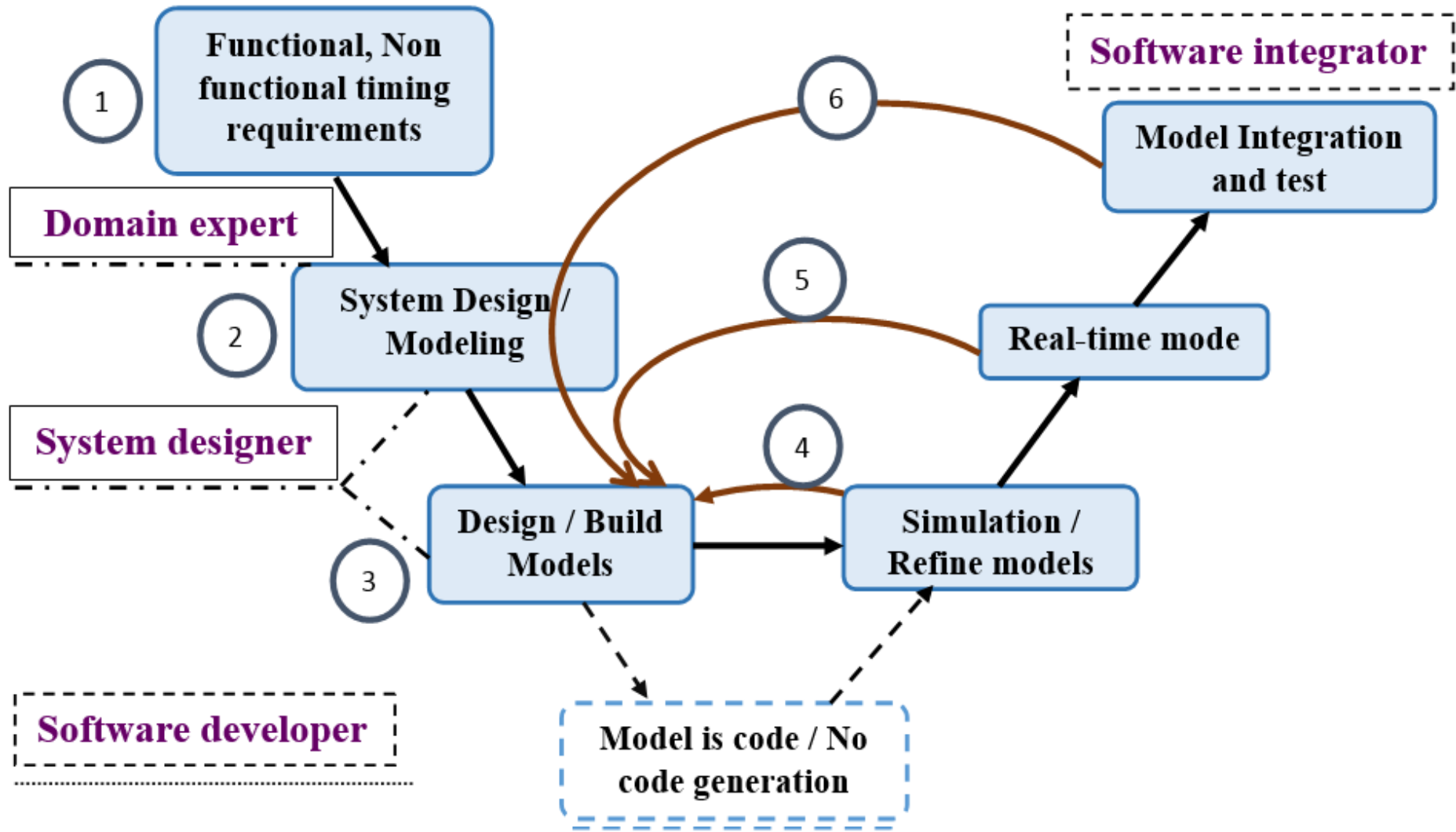
- Exploring the ***theoretical benefits*** of model interpretation with a industrial experiment



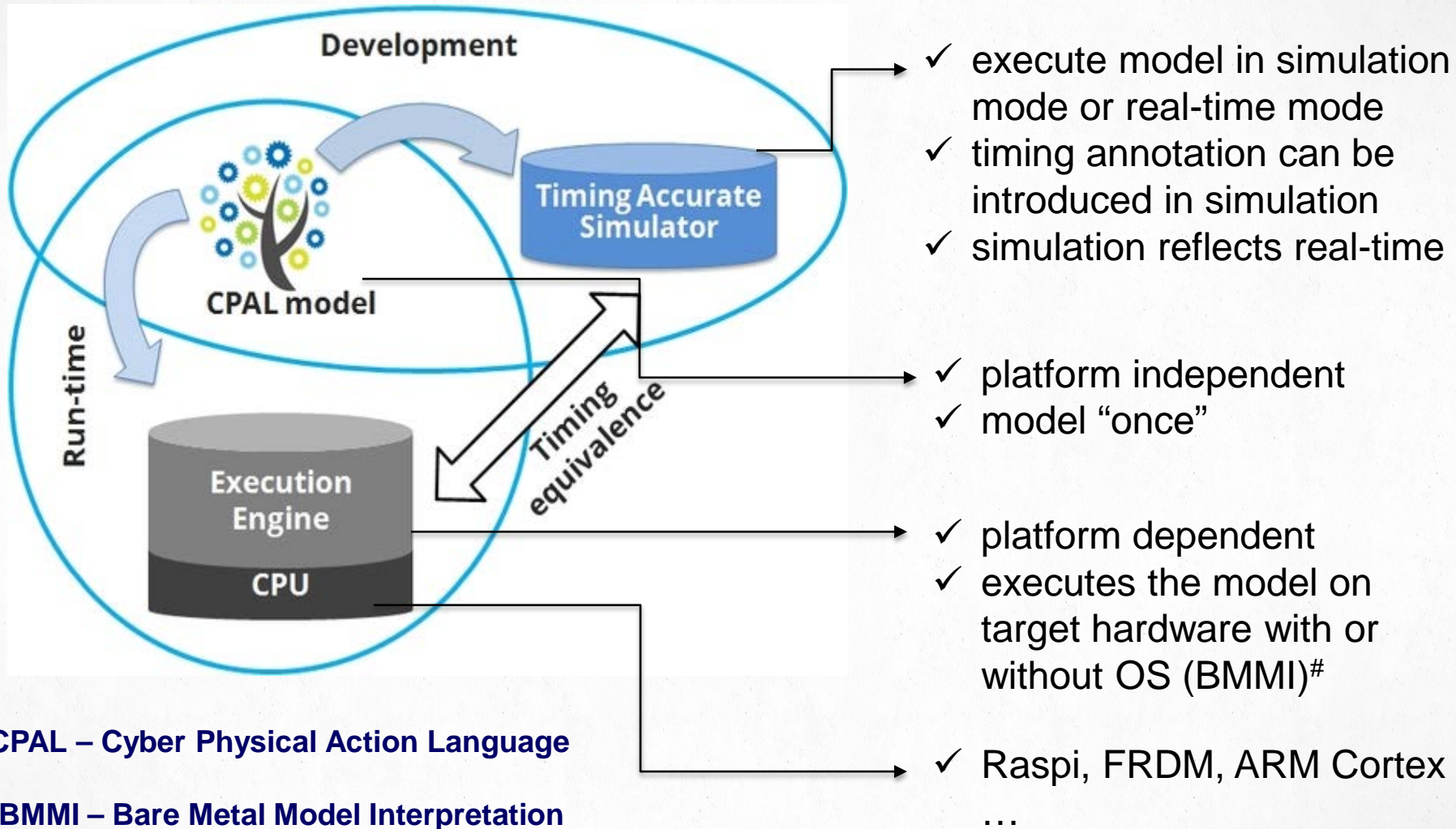
Observations on productivity, simplicity, and performance (discussions)



Lean Development Cycle



CPAL* - an Interpreted MBD

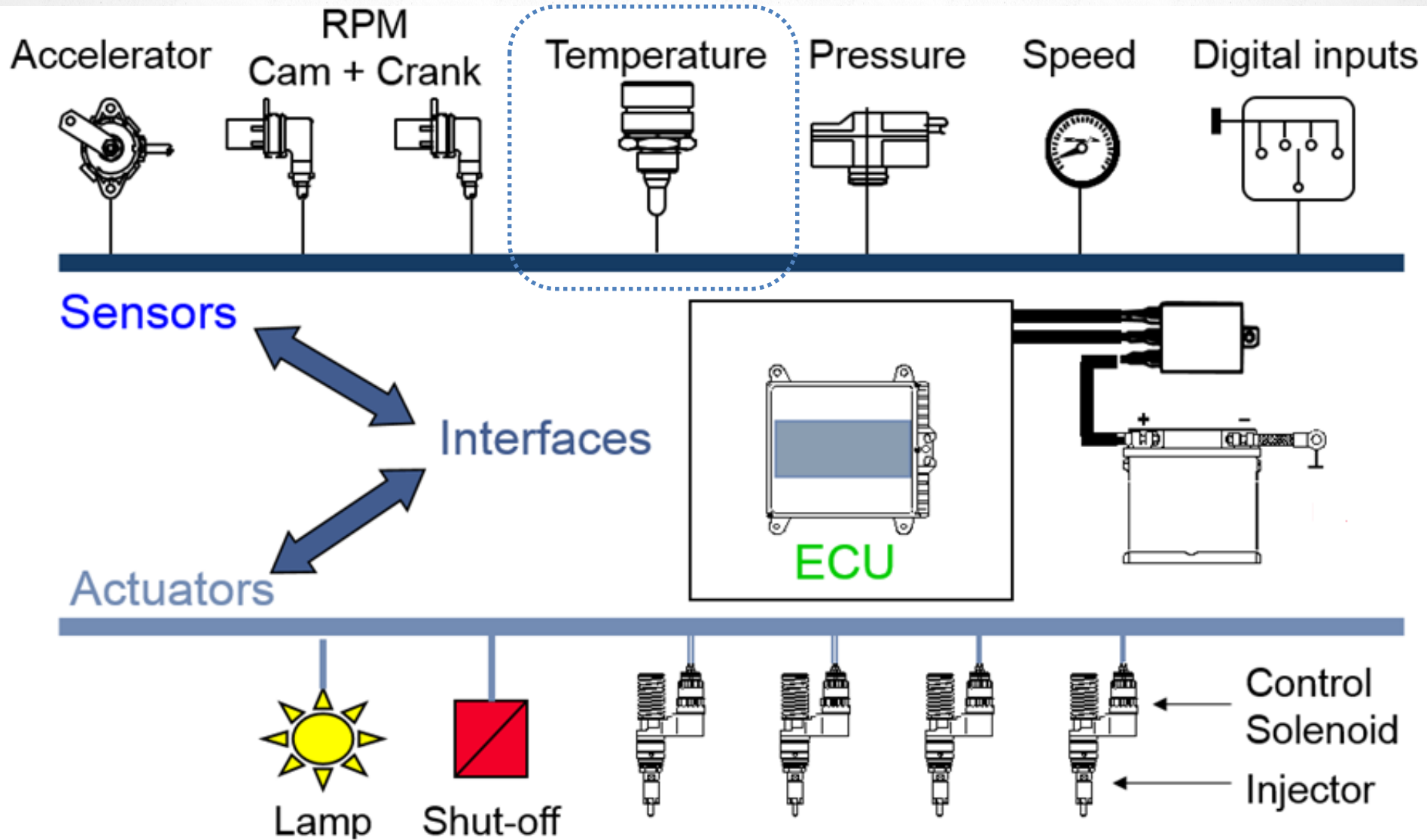


* CPAL – Cyber Physical Action Language

BMMI – Bare Metal Model Interpretation



Engine Control System



Engine Subsystems

Mechanical & ECU

→ Air System

Air-Filter, Intake Manifold, Turbo-Charger / Super-Charger
Air Mass Sensor, Manifold Pressure/Temperature Sensor, Electronic Throttle

→ Fuel Injection System

Fuel Tank, Fuel Filter, Fuel Pump, Injector
Fuel Tank Pressure Sensor, Fuel Pump, Electrical Injector, Canister Purge Valve, Fuel Rail Pressure Sensor, Rail Pressure control valve

→ Cooling System

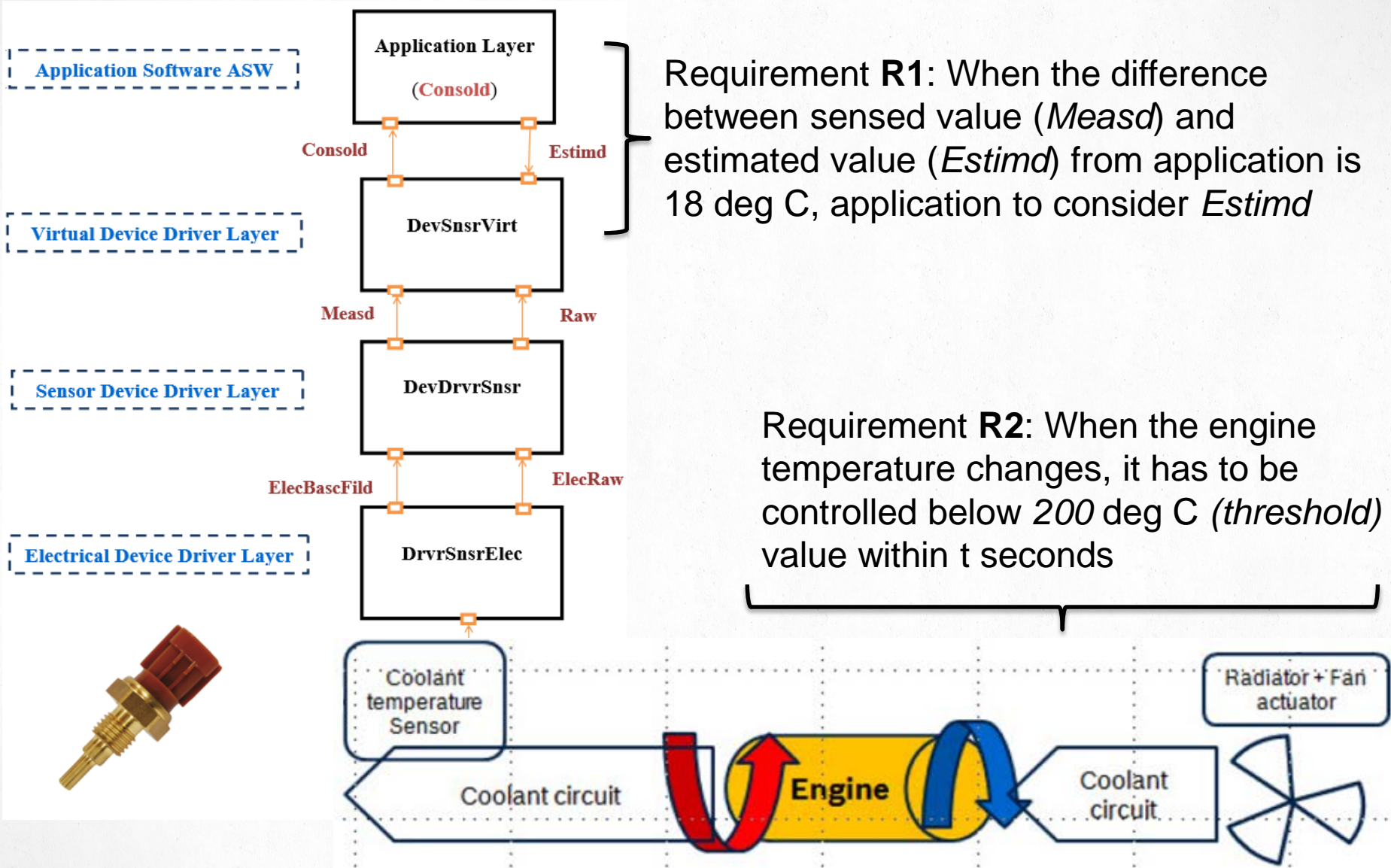
Coolant (Water) Reservoir, Water Pump, Radiator, Fan
Electrical Water Pump, Electrical Fan, Water Temperature Sensor, Flow Control Valves

→ Exhaust System

Exhaust Manifold, Exhaust Pipe, Exhaust Muffler, Catalytic Converter
Exhaust Temperature Sensor, Lambda Sensor, NOx Sensor, EGR Valve, Secondary Air Pump, Secondary Air Valve



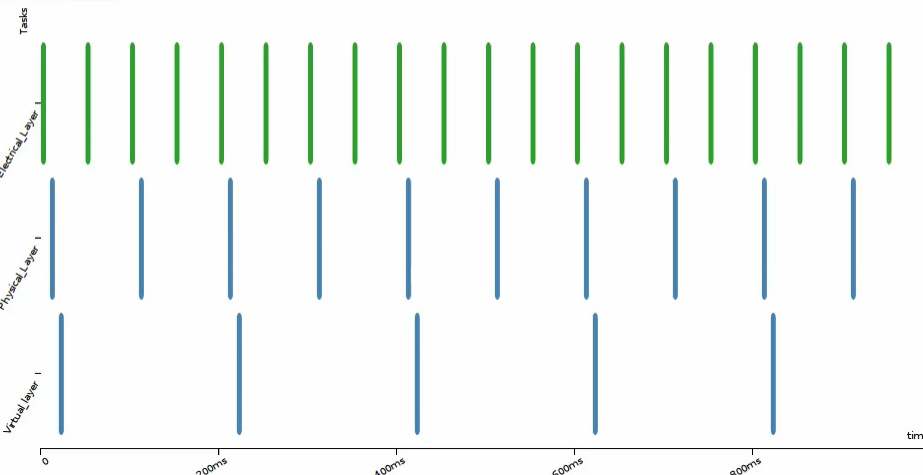
AUTOSAR Case Study



Let's have a look

coolantgit.cpal (/home/pi/CPAL-Editor-raspberry) - CPAL Editor @raspberrypi

File Edition Display Run ?
Architecture Tasks



```
coolantgit.cpal
110     }
111     else
112         raw=0.0;
113     }
114 }
115 processdef virtuallayer(in uint32: d,in float32: raw,out float32: T)
116 {
117     state main
118     {
119         var float32: raw_float;
120         var uint32: rpm;
121         raw_float=raw*(9.0/5.0)+32.0;
122         IO.println("raw_float:%f",raw_float);
123         IO.println("d:%u",d);
124         rpm=uint32.as((((float32.as(d)*3.3)/1024.0)/(10.0/1000.0))*1.8+32.0)*40.0;
125         IO.println("rpm:%u",rpm);
126         if(float32.as(model[rpm/600])-18.0<raw_float and float32.as(model[rpm/600])
127         {
128             pin0_out=true;
129             pin1_out=false;
130             IO.println("Engine Coolant Temperature in Celsius:%f",raw);
131             IO.println("Real value");
132         }
133         else
134         {
135             pin0_out=false;
136             pin1_out=true;
137             Temperature=float32.as(model[rpm/600]);
138             IO.println("Engine Coolant Temperature in Celsius:%f",Temperature);
139             IO.println("Modeled value");
140         }
141     }
142 }
143 }
144 var queue<uint32>: ttyTemperature_in[2000];
145 process electricalayer: Electrical_Layer[50ms](ttyTemperature_in,ElecRaw);
146 process physicallayer: Physical_Layer[100ms,10ms](ElecRaw,Raw);
147 process virtuallayer: Virtual_Layer[200ms,20ms](d,value,Raw,Temperature);
```

Parse success !
AST generated in file "/tmp/cpal_editor5472746733412900432.ast"



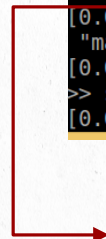
Observation # 1 – Early stage execution

Timing accurate simulation
and real-time execution



```
pi@raspberrypi ~/cpal/coolant $ sudo ./cpal_interpreter_raspberry -r git.ast
=> Digital pin found: 0, output
=> Digital pin found: 1, output
=> a serial read only /dev/ttyTemperature
[0.000000000000:ASSIGN] Assign pin0_out new value: false
[0.000000000000:ASSIGN] Assign pin1_out new value: false
[0.000000000000:ASSIGN] Assign flag_new value: 0
[0.000000000000:ASSIGN] Assign adcvalue new value: 0
[0.000000000000:ASSIGN] Assign modelvalue new value: 0
[0.000000000000:ASSIGN] Assign digit new value: 0
[0.000000000000:ASSIGN] Assign mode new value: 0
[0.000000000000:ASSIGN] Assign ElecRaw new value: 0.000000
[0.000000000000:ASSIGN] Assign faw new value: 0.000000
[0.000000000000:ASSIGN] Assign Temperature new value: 0.000000
[0.000000000000:ASSIGN] Assign model new value: {60,65,70,75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,180,185,190,195,200,205,210,215,220,225,230,235}
[0.000000000000:ASSIGN] Assign ttyTemperature_in new value: {}
>> step
[0.000000000000:STATE] process "electricallayer", instance "Electrical_Layer", state "Main"
>> step
[0.010000000000:STATE] process "physicallayer", instance "Physical_Layer", state "main"
[0.010000000000:ASSIGN] Assign Raw new value: 0.000000
>> step
[0.020000000000:STATE] process "virtuallayer", instance "Virtual layer", state "main"
```

No need of tracing from code to model when failure occurs



Step by step execution – functional verification and model debugging

Finding failure in model is easier (No code)



2 – Requirement change is easier

say R1 (slide #9) is requested to be changed 18 to 12 deg C

No code
No compilation
No linking
No binary
Executable model is readily available with change for R1

adapt the model to R1

```
110 }
111 else
112   raw=0.0;
113 }
114 }
115 processdef virtuallayer(in uint32: d,in float32:
116 {
117   state main
118   {
119     var float32: raw_float;
120     var uint32: rpm;
121     raw_float=raw*(9.0/5.0)+32.0;
122     IO.println("raw_float:%f", raw_float);
123     IO.println("d:%u", d);
124     rpm=uint32.as((((float32.as(d)*3.3)/1024.0)
125     IO.println("rpm:%u", rpm);
126     if(float32.as(model[rpm/600])-18.0<raw_float
127     {
128       pin0_out=true;
129       pin1_out=false;
130       IO.println("Engine Coolant Temperature in
131       IO.println("Real value");
132     }
133     else
134     {
135       pin0_out=false;
136       pin1_out=true;
137       Temperature=float32.as(model[rpm/600]);
138       IO.println("Engine Coolant Temperature in
139       IO.println("Modeled value");
140     }
141   }
142 }
143 }
144 var queue<uint32>: ttyTemperature_in[2000];
145 process electricallayer: Electrical_Layer[50ms](
146 process physicallayer: Physical_Layer[100ms,10ms]
```

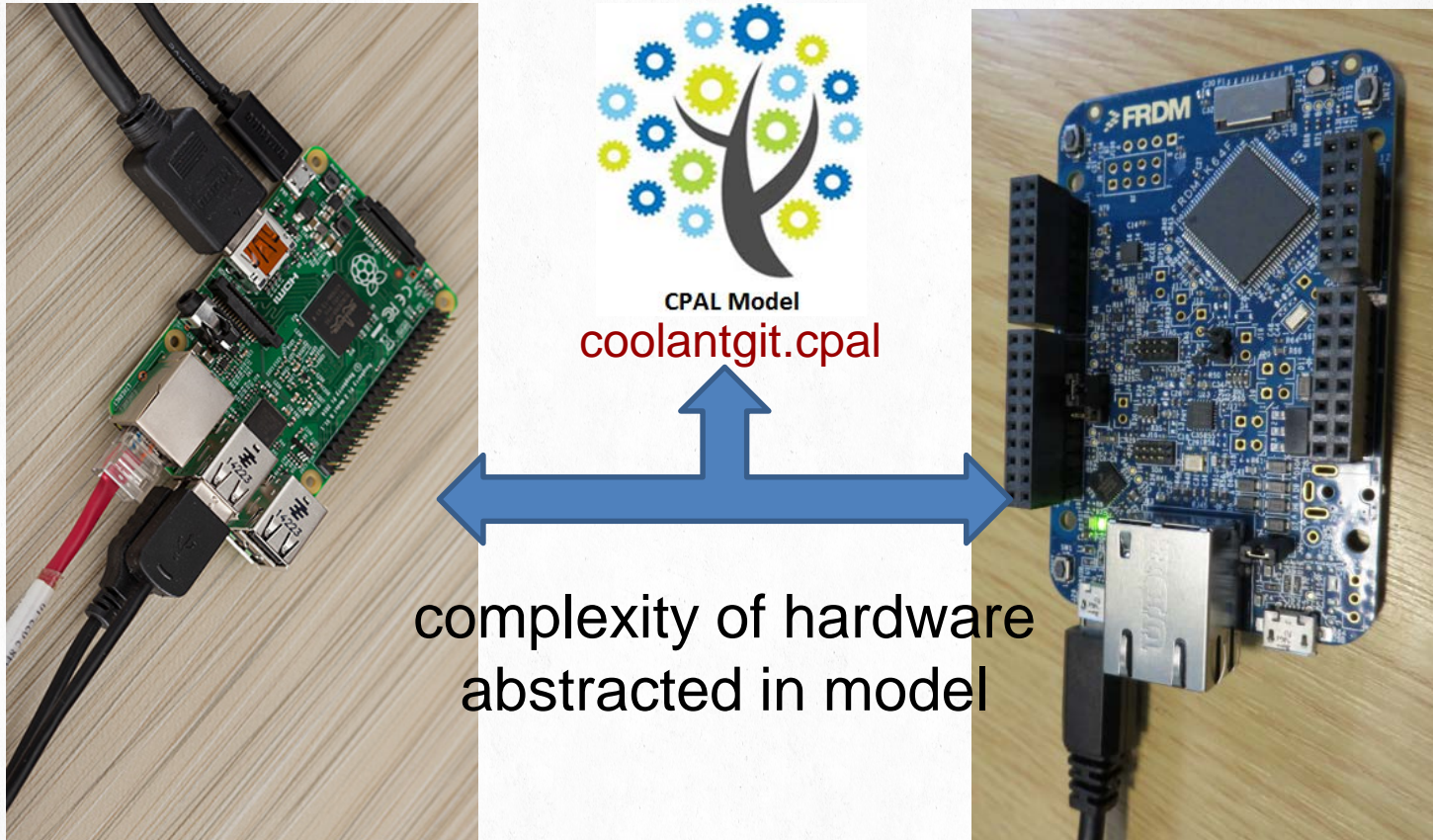
Parse success !
AST generated in file "/tmp/cpal_editor3046313860291156010.ast"

Instantaneous execution of change requested



3 – Hardware Independence

CPAL model is readily portable to any hardware



Interpretation engine to be adapted to HW - similar to code-generator switch to a new HW



4 – Design exploration

Functional architecture of the system – Domain expert view

The screenshot displays the CPAL Editor interface. On the left, the 'Domain expert view' shows a 'Gantt-chart of tasks activations' with three layers: 'Electrical_Layer' (green bars), 'Physical_Layer' (blue bars), and 'Virtual_Layer' (blue bars). The x-axis represents time from 0 to 600ms. A 'Modeling environment' window is open below the chart. On the right, the 'Real-time / Simulation' view shows the 'modes of execution' and a code editor with the following code:

```
64   adcvalue=data-320;
65   data=0;
66   flag=0;
67 }
68 }
69 }
70 }
71 }
72 processdef physicallayer(in float32: eraw,out float32:
73 {
74   state main
75   {
76     if(uint32.as(eraw)!=0)
77     {
78       var float32: volts; var float32: ohms; var
79       var float32: a = 0.002197222470870;
80       var float32: b = 0.000161097632222;
81       var float32: c = 0.000000125008328;
82       var float32: t1; var float32: c2; var fl
83       var float32: tempc;
84       var time64: time1; var time64: time2;
85       /*log variables*/
86       var float32: taylor_s=0.0;
87       var float32: temp=0.0;
88       var uint32: i=0;
89       var float32: temp_pow=0.0;
90       volts=(eraw*3.7)/1024.0; /*3.3*/
91       ohms=((1.0/volts)*2900.0)-1000.0; /*3300*
92       time1 = time64.time();
93       IO.println("NTC Thermistor resistance:%f"
94       /*log calculation*/
95       while(true)
96       {
97         taylor_s=taylor_s+t
```

At the bottom, a 'Compiled code/Parse errors window' shows the message: 'Parse success ! AST generated in file "/tmp/cpal_editor1902628727656799295.ast"'. A blue cloud contains the text: 'All stake-holders are connected seamlessly'.

Model

Developer view

Scheduling of processes during simulation – timing analysis view



Our thoughts on Low-lights / Next steps

- Code generation is standard practice
- Model interpretation is **slower** than code executed – **Still...**

Calling binary code (computation-intensive portions) from interpreted code

design phase – model interpretation to benefit productivity / easier verifiability aspects

- Production phase – Code generation to benefit faster execution capability
- Interpretation and code generation are often seen as two alternatives, not as a **continuum**

