Using C Language Extensions for Developing Embedded Software - A Case Study

Markus Völter voelter@acm.org
Arie van Deursen Arie.vanDeursen@tudelft.nl
Stephan Eberle stephan.eberle@itemis.com
Bernd Kolb Bernd.kolb@itemis.de

(How well) do domain-specific language extensions work?

And how can we find out?

Domain-Specific Extensions of C for Embedded Software

An Industrial Case Study

An Industrial Case Study

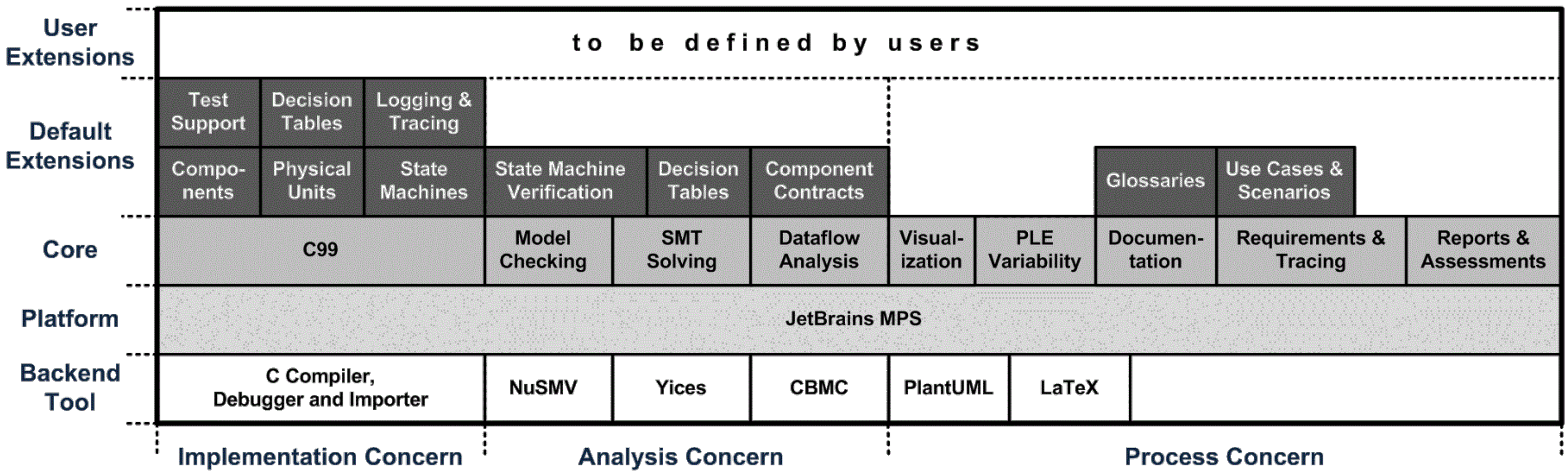
Smart Meter

Measures Voltage and Current
Computes Derived Values
Shows Data on LCD Display
Communicates through Networks

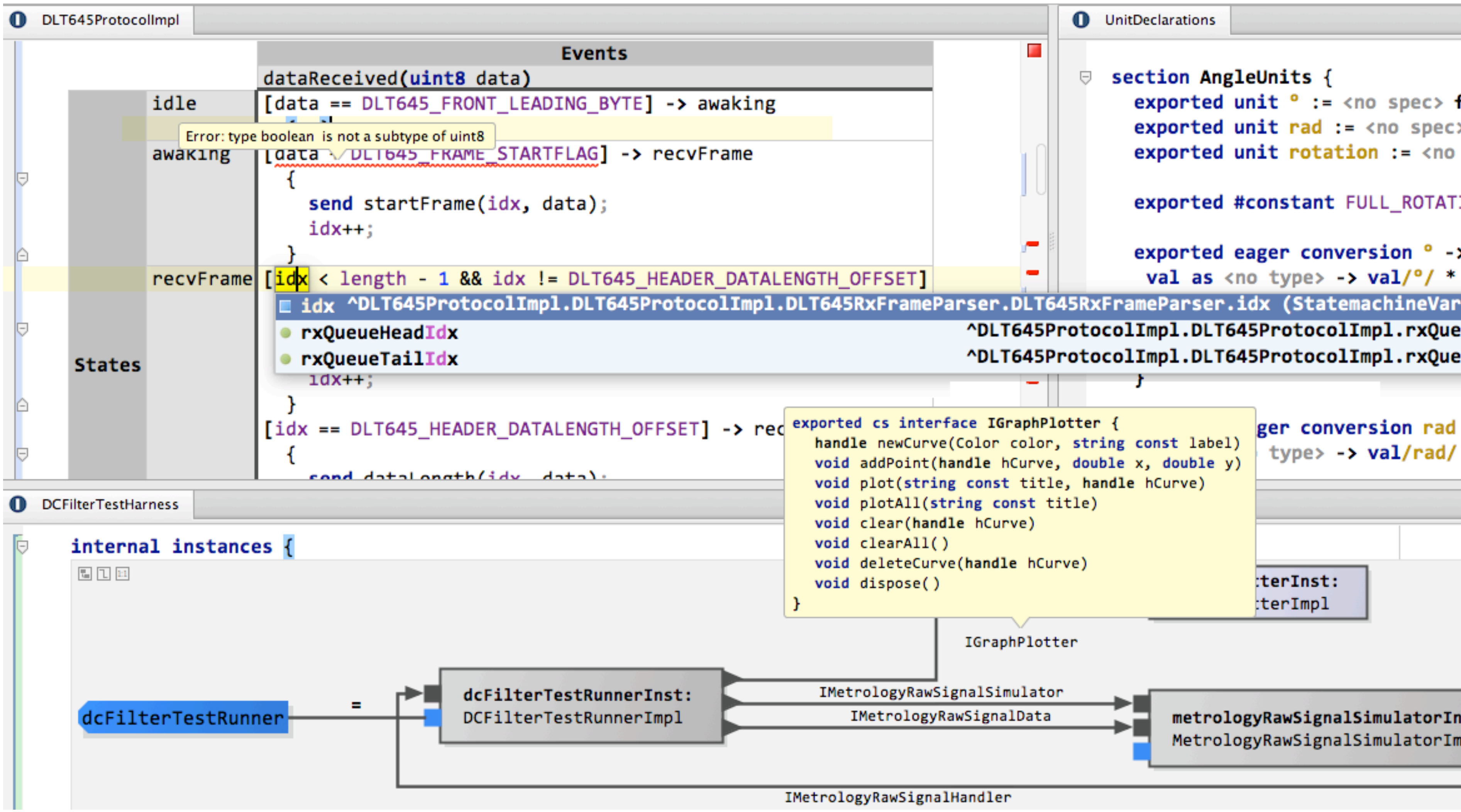
Precision is critical for Certification.
Evolvability is critical for it to be a viable business.

Developed with mbeddr, a set of domain-Specific extensions to C, plus an IDE.

An extensible set of integrated languages for embedded software engineering.



mbeddr



DLT645ProtocolImpl
Events
dataReceived(uint8 data)
idle
[data == DLT645_FRONT_LEADING_BYTE] -> awaking
Error: type boolean is not a subtype of uint8
awaking
DLT645_FRAME_STARTFLAG] -> recvFrame
send startFrame(idx, data);
idx++;
recvFrame
[idx < length - 1 && idx != DLT645_HEADER_DATALENGTH_OFFSET]
idx ^DLT645ProtocolImpl.DLT645ProtocolImpl.DLT645RxFrameParser.DLT645RxFrameParser.idx (StatemachineVar
rxQueueHeadIdx
rxQueueTailIdx
States
[idx == DLT645_HEADER_DATALENGTH_OFFSET] -> rec
UnitDeclarations
section AngleUnits {
exported unit ° := <no spec>
exported unit rad := <no spec
exported unit rotation := <no
exported #constant FULL_ROTAT
exported eager conversion °
val as <no type> -> val/°/ *
exported cs interface IGraphPlotter {
handle newCurve(Color color, string const label)
void addPoint(handle hCurve, double x, double y)
void plot(string const title, handle hCurve)
void plotAll(string const title)
void clear(handle hCurve)
void clearAll()
void deleteCurve(handle hCurve)
void dispose()
}
DCFilterTestHarness
internal instances {
IGraphPlotter
dcFilterTestRunner
dcFilterTestRunnerInst:
DCFilterTestRunnerImpl
IMetrologyRawSignalSimulator
IMetrologyRawSignalData
metrologyRawSignalSimulatorIn
MetrologyRawSignalSimulatorIm
IMetrologyRawSignalHandler

Setup

Context: Industry Project

very

Realistic

Real requirements, real size, real deadlines, representative developers

maybe not so

Reproducible

Not so easy to reproduce, because the source code of Smart Meter is not available. mbeddr itself is open source, though:

http://mbeddr.com/

Research Questions

Complexity	Are the abstractions provided by mbeddr beneficial for mastering the complexity encountered in a real-world embedded system? Which additional abstractions would be needed or useful?
Testing	Can the mbeddr extensions help with testing the system? In particular, is hardware-independent testing possible to support automated, continuous integration and build? Is incremental integration and commissioning supported?
Overhead	Is the low-level C code generated from the mbeddr extensions efficient enough for it to be deployable onto a real-world embedded device?
Effort	How much effort is required for developing embedded software with mbeddr?

Data Collected

Complexity

Qualitative impact of mbeddr and SM extensions on complexity

Testing

Measured Coverage
Test-Specific SMT Code
Commissioning of the system

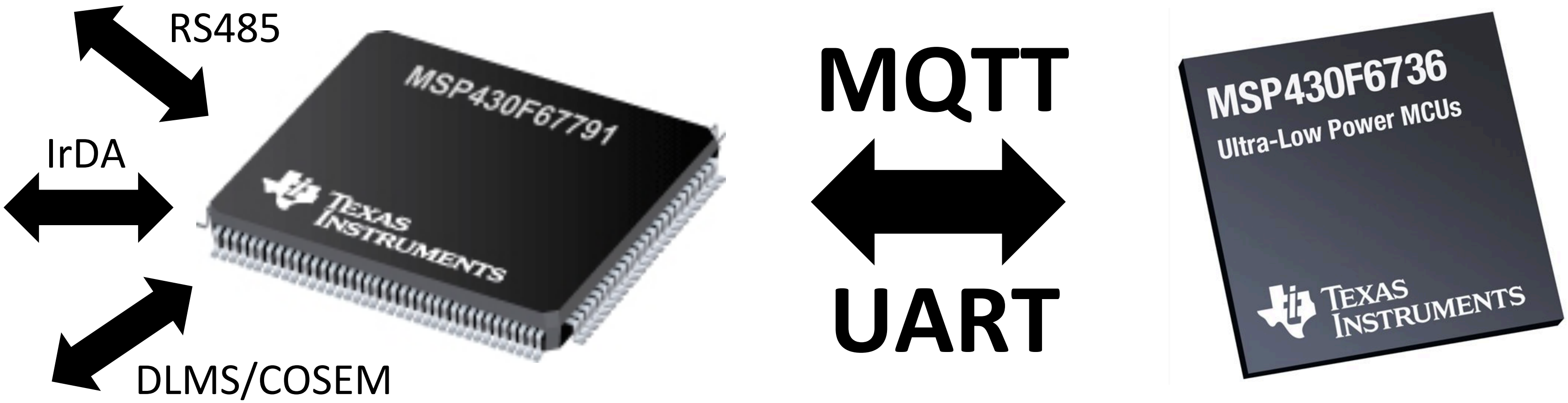
Overhead

Compared Size of Binary with Resources of Hardware
Analyzed/Measured Performance
Theoretical Discussion of the Overhead of Extensions

Effort

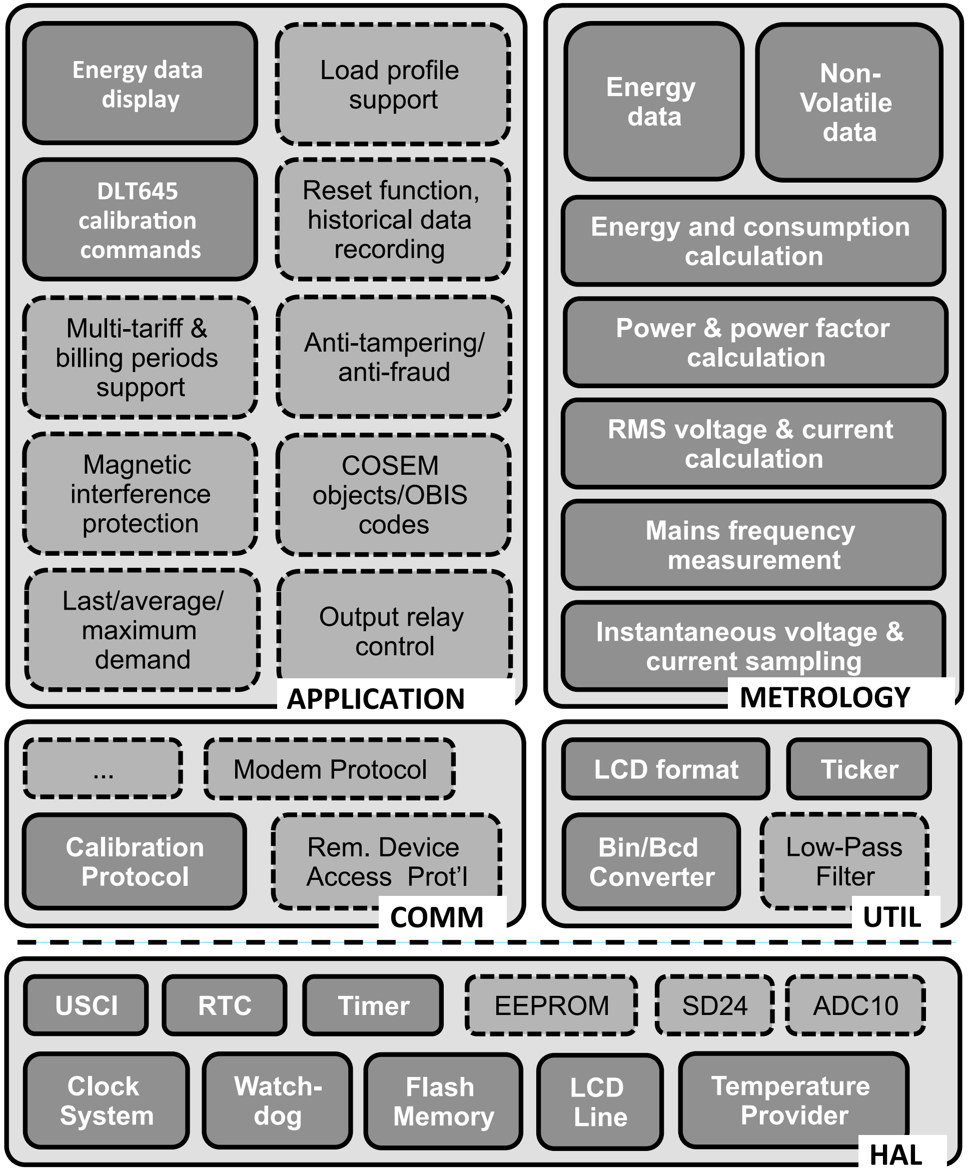
Report and discuss Effort required to build SM separated by implementation, testing, commissioning and extension development

Hardware Architecture



Application Logic	Metrology
MSP430 F67791	**MSP430** F6736
25 MHz	25 MHz
256K Flash ROM	128K Flash ROM
32K RAM	8K RAM

Software Architecture



No RTOS

Interrupt-Driven

One-Threaded Programming

Required Precision leads to 4096 Hz Sampling Rate

Interrupt-Triggered:

Measurement

Foreground Tasks:

App Logic, RTC

Example Smart Meter Code

From the processor vendor. But: no tests, bad structure, buggy, not all features.

Hence:

Phase 1 Reimplement with mbeddr

Phase 2

Two Processors,
Communication between the two processors,
Improved comms infrastructure (multiplexing,
    two comm stacksRS485 and IrDa)
an I2C Bus driver
an EEPROM controller
a subset of the required DLMS/COSEM messages
additional application functionality (historical data rec, reset)

Size of the System

Criterion	Common	Metro	App	Total
# of Files	134	101	105	340
Total LOC	8,209	10,447	10,908	29,564
Code LOC	4,397	5,900	5,510	15,807
Comment LOC	950	2,402	2,620	5,972
Whitespace LOC	2,852	2,145	2,778	7,775

Common code runs on both processors, **Metro** runs on the metrology processor and **App** runs on the application / communication processor.

+ roughly the same amount again for **tests**.

Use of Extensions

Category	Concept	Count
Chunks (≈ Files)	Implementation Modules	382
	Other (Req, Units, etc.)	46
C Constructs	Functions	310
	Structs / Members	144 / 270
	Enums / Literals	150 / 1,211
	Global Variables	334
	Constants	8,500
Components	Interfaces / Operations	80 / 197
	Atomic Components	140
	Ports / Runnables	630 / 640
	Parameters / Values	84 / 324
	Composite Components	27
	Component Config Code	1,222
State Machines	Machines	2
	States/Transitions/Actions	14 / 17 / 23
Physical Units	Unit Declarations	122
	Conversion Rules	181
	Types / Literals with Units	593 / 1,294

Category	Concept	Count
Product Line Variability	Feature Models / Features	4 / 18
	Configuration Models	10
	Presence Condition	117
Custom Extensions	Register Definition	387
	Interrupt Definitions	21
	Protocol Messages	42
Statements	Statements total	16,840
	Statements in components	6,812
	Statements in test cases	5,802
	Statements in functions	3,636
Testing	Test Cases / Suites	107 / 35
	Test-Specific Components	56
	Stub / Mock Components	9 / 8
	`assert` Statements	2,408

All mbeddr C extensions used a lot.
Some extensions built specifically for SM.

The Code

Components (mbeddr)

```
// ADC is the analog-digital converter
interface IADC {
  int16 read(uint8 addr)
}
```

```
component ADCDriver {
  provides IADC adc
  int16 adc_read(uint8 addr) <= op adc.read {
    int16 val = // low level code to read from addr
    return val;
} }
```

```
component CurrentMeasurer {
  requires IADC currentADC
  internal void measureCurrent() {
    int16 current = currentADC.read(CURR_SENSOR_ADDR);
    // do something with the measured current value
} }
```

State Machines (mbeddr)

```
statemachine FrameParser initial = idle {
  var uint8 idx = 0
  in event dataReceived(uint8 data)
  state idle {
    entry { idx = 0; }
    on dataReceived [data == LEADING_BYTE] -> wakeup
  }
  state wakeup {
    on dataReceived [data == START_FLAG]
       -> receivingFrame { idx++; }
  }
  state receivingFrame { .. }
}
```

```
// create and initialize state machine
FrameParser parser;
parser.init;
// trigger dataReceived event for each byte
for (int i=0; i<data_size; i++) {
  parser.trigger(dataReceived|data[i]);
}
```

Testing & State M. (mbeddr)

```
testcase testFrameParser1 {
  FrameParser p;
  assert(0) p.isInState(idle);
  // invalid byte; stay in idle
  parser.trigger(dataReceived|42);
  assert(0) p.isInState(idle);
  // LEADING_BYTE, go to awakening
  parser.trigger(dataReceived|LEADING_BYTE);
  assert(0) p.isInState(awakening);
}

testcase testFrameParser2 { ... }
testcase testFrameParser3 { ... }

int32 main(int32 argc, char* argv) {
  return test[testFrameParser1,
              testFrameParser2,
              testFrameParser3];
}
```

Mocks & Units (mbeddr)

```
mock component USCIReceiveHandlerMock {
  provides ISerialReceiveHandler handler
  Handle* hnd;
  sequence {
    step 0: handler.open { } do { hnd = handle; }
    step 0: handler.dataReceived {
        assert 0: parameter data: data == 1 }
    step 1: handler.dataReceived {
        assert 1: parameter data: data == 2 }
    step 2: handler.dataReceived { .. }
    step 3: handler.dataReceived { .. }
    step 4: handler.finished { } do { close(hnd); }
} }
```

```
unit V :=        for voltage
unit A :=        for Amps
unit Ω := V·A^-1 for resistance

uint16/Ω/ resistance(uint16/V/ u, uint16/A/[] i, uint8 ilen) {
  uint16/A/ avg_i = (Σ_{p = 0}^{ilen} i[p]) / ilen;
  return avg_i / u;   Error: type uint16 /V^(-1) · A/ is not a subtype of uint16 /Ω/
} resistance (function)
```

Product Lines (mbeddr)

```
feature model SMTFeatures
  root opt
    Data_LEDs opt
      DataReadLED
      DataWriteLED [DigitalIOPortPin pin]
    DISPLAY xor
      DISPLAY_V10
      DISPLAY_V22
    WRITABLE_FLASH_MEMORIES
```

```
exported composite component MetrologyPlatformLayer {
  provides IWatchdogTimer watchdogTimer
  ? {DataReadLED && WRITABLE_FLASH_MEMORIES}
  ?provides IDigitalOutputPin pin1
  ? {DataWriteLED}
  ?provides IDigitalOutputPin pin2
```

Registers (smart meter)

```
exported register8 ADC10CTL0 compute as val * 1000

void calculateAndStore( int8 value ) {
   int8 result = // some calculation with value
   ADC10CTL0 = result; // stores result * 1000 in reg.
}
```

Interrupts (smart meter)

```
module USCIProcessor {
  exported interrupt USCI_A1
  exported interrupt RTC

  exported component RTCImpl {
    void interruptHandler() <- interrupt {
      hw->pRTCPS1CTL &= ~RT1PSIFG;
} } }
```

```
instances usciSubsystem {
  instance RTCImpl rtc;
  bind RTC -> rtc.interruptHandler
  connect ... // ports
}
```

Messages (smart meter)

```
// a field representing a timestamp for 10:20:00
uint8[6] f_time = {0x00A,  // field type identifier
                   UNIT_TIME24, // unit used: time
                   3, // 3 payload bytes follow
                   10, 20, 00 // the time itself
                  };

// a field repre
uint8[4] f_value

// a message that uses the two fields
uint8[5] message
```

```
message CurrentMeasuredValue:42 {
  int32      timestamp; // time of measurement
  uint16/A/ value;      // measured value in Amps
  uint16    accuracy;  // accuracy in 1/100 %
}
message ... { ... }
...
```

```
atomic component CoreMeasurer {
  field uint16/A/ lastValue = 0;
  message data 42 {:currentTime, &lastValue, 100};
  void measure() {
    lastValue = // perform actual measurement
} }
```

Answers to RQs

RQ Complexity

The developers **naturally think** in terms of extensions, and suggested additional ones during the project.

mbeddr **components help structure** the overall architecture and enable reuse and configurability.

mbeddr extensions facilitate strong **static checking**, improve **readability** and help avoid **low-level mistakes**.

RQ Testing

mbeddr components are instrumental in **improving testability** through clear interfaces and small units, leading to 80% test coverage for core components.

The custom extensions and the components **facilitate hardware-independent testing**, continuous integration and automated dry runs of the certification process.

The modularization facilitated by components **helps track down** problems during commissioning.

RQ Overhead

The **memory** requirements of SMT are **low enough** for it to run on the intended hardware, with room for growth.

Componentization enables **deployment of only the functionality necessary** for a variant, conserving resources.

The **performance** overhead is low enough to achieve the required **4,096 Hz sample rate** on the given hardware.

RQ Effort

Development Tasks	Effort	% Total
Implementation	200 PD	66%
Reimplementation	145 PD	48%
Additional Functionality	55 PD	18%
Tests, Simulators	48 PD	16%
Integration & Commissioning	38 PD	13%
Custom Language Extensions	14 PD	5%

RQ Effort

The **effort** for the additional functionality, integration and commissioning is **lower than what is common** in embedded software.

The effort for **building the extensions** is low enough for it to be absorbed in a real project.

Overall, using **mbeddr does not lead to significant effort overrun**, while resulting in better-structured software.

Discussion

Validity

Internal Bias, Team Expertise
Example Smart Meter Code

Conclusion Design of mbeddr
Cognitive Dimensions of N.
Concepts vs. Language
Language vs. Tool

External Beyond SM
Beyond the Team
Beyond the mbeddr Extensions
Beyond mbeddr's MPS Implementation

Discussion

Debugging on the DSL Level
an on the generated level

Code Quality Readable to build Trust
Readable for Debugging
MISRA Compliant: 25% automatic

Maintainability No long term experience
But good indications:
additional functionality

Drawbacks and Challenges

Limited Generator optimizations

same execution paradigm, not a problem yet.

2.5 X Longer Build Times

Tool Lock in: no way without MPS

Diff/Merge in MPS only

Learning Curve

Language Engineering Skills to build new L

Other Approaches

How is it different from...

Model-Driven-*

Fully open and exensible

Multiple paradigms, not one-size-fits-all

Mix of „Model and Code“

How is it different from...

Macros

More syntactic flexibility

Higher Expressivity (do more than with Macros)

Type Checking

Generally better IDE support

How is it different from...

C++

Requires no C++ Compiler

Components more suitable for Embedded

Different Features: units, state machines

TMP: Better IDE support

Better Error Messages

LE better done in LWB

Conclusions

Specific:
mbeddr & Smart Meter

The extensions help **master complexity** and lead to software that is more **testable**, easier to **integrate and commission** and is more evolvable.

Specific:
mbeddr & Smart Meter

Despite the abstractions introduced by mbeddr, the additional **overhead is very low** and acceptable in practice.

Specific:
mbeddr & Smart Meter

The development **effort is re-duced**, particularly regarding evolution and commissioning.

Generic:

Language Extensions

Based on mbeddr and Smart Meter, we consider language extension a very fruitful approach.

We have also used it in other domains, including robot control, engine management and insurance product definition.

Generic:

Case Study Research

Using real industry projects as case studies yields practically meaningful results, despite the drawbacks.

Language Extension Works!

Using C Language Extensions for Developing Embedded Software - A Case Study

Markus Völter	voelter@acm.org
Arie van Deursen	Arie.vanDeursen@tudelft.nl
Stephan Eberle	stephan.eberle@itemis.com
Bernd Kolb	Bernd.kolb@itemis.de