Spark in an automotive context

Florian Schanda
June 3, 2019
Who is Zenuity?

• AD and ADAS software company owned by Veoneer and Volvo
• Customers:
  1. Volvo
  2. Through Veoneer: Geely, an American OEM, a German OEM, etc.
• Offices in Göteborg, München, Detroit, Santa Clara, and Shanghai
• Primary priority is safety
Why is AD so hard?

Technical issues

Safety critical, and:

- Complex environment (roads, road users, etc.)
- No obvious fail-safe architecture
- Nobody knows how to do it
- Large safety-critical code base (compared to other industries)
Why is AD so hard?

Technical issues

Safety critical, and:

- Complex environment (roads, road users, etc.)
- No obvious fail-safe architecture
- Nobody knows how to do it
- Large safety-critical code base (compared to other industries)
- Will contain neural networks
Why is AD so hard?

Technical issues

Safety critical, and:

- Complex environment (roads, road users, etc.)
- No obvious fail-safe architecture
- Nobody knows how to do it
- Large safety-critical code base (compared to other industries)
- Will contain neural networks
  - Effective validation approach unclear
  - Verification is a research problem
Why is AD so hard?

Process issues

• No experience with large safety-critical software in automotive
• Feature engineers, not programmers (in classical OEMs and suppliers)
• Unsuitable processes (what works in ADAS won’t work here)
ISO 26262

- Key safety standard in automotive
ISO 26262

- Key safety standard in automotive
- Defines integrity levels, precise mapping is up to interpretation

<table>
<thead>
<tr>
<th>IEC 61508</th>
<th>DO-178C</th>
<th>ISO 26262</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>DAL-E</td>
<td>QM</td>
</tr>
<tr>
<td>SIL-1</td>
<td>DAL-D</td>
<td>ASIL-A</td>
</tr>
<tr>
<td>SIL-2</td>
<td>DAL-C</td>
<td>ASIL-B/C</td>
</tr>
<tr>
<td>SIL-3</td>
<td>DAL-B</td>
<td>ASIL-D</td>
</tr>
<tr>
<td>SIL-4</td>
<td>DAL-A</td>
<td>–</td>
</tr>
</tbody>
</table>
ISO 26262

- Key safety standard in automotive
- Defines integrity levels, precise mapping is up to interpretation

<table>
<thead>
<tr>
<th>IEC 61508</th>
<th>DO-178C</th>
<th>ISO 26262</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>DAL-E</td>
<td>QM</td>
</tr>
<tr>
<td>SIL-1</td>
<td>DAL-D</td>
<td>ASIL-A</td>
</tr>
<tr>
<td>SIL-2</td>
<td>DAL-C</td>
<td>ASIL-B/C</td>
</tr>
<tr>
<td>SIL-3</td>
<td>DAL-B</td>
<td>ASIL-D</td>
</tr>
<tr>
<td>SIL-4</td>
<td>DAL-A</td>
<td>–</td>
</tr>
</tbody>
</table>

- Defines verification objectives
- Suggests minimal approach to meet them
Demonstrate that the software units achieve:

- compliance with the software unit design specification
- compliance with the specification of the hardware-software interface
- the specified functionality
- confidence in the absence of unintended functionality
- robustness
- sufficient resources to support their functionality
Demonstrate that the software units achieve:

- compliance with the software unit design specification
- compliance with the specification of the hardware-software interface
- the specified functionality
- confidence in the absence of unintended functionality
- robustness
- sufficient resources to support their functionality
It can mean a lot of things:

- no dead code?
- error detection effective?
- error handling effective?
- code does not crash?
- signal noise?
ISO 26262
Robustness - “obvious”

This objective is classically hard through testing:

- Trivial to find “bugs”

```c
result = (previous_speed + current_speed) * 0.5f;
```
This objective is classically hard through testing:

- Trivial to find “bugs”

```c
result = (previous_speed + current_speed) * 0.5f;
```

What about if our car goes at $1.13 \times 10^{30} \times c$?

- These bugs are “not helpful” and “obviously” irrelevant
- Often the only way to fix them is defensive code or justification
  - Defensive code further increases testing effort (coverage)
  - Justifications are often wrong, outdated, or fail to grasp the big picture
ISO 26262
Robustness - testing is hard

- Coverage metrics helps you complete your test-suite
- Various levels exist

<table>
<thead>
<tr>
<th>Level of assurance</th>
<th>Coverage criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIL-A</td>
<td>Statement</td>
</tr>
<tr>
<td>ASIL-B</td>
<td>Statement or Branch</td>
</tr>
<tr>
<td>ASIL-C</td>
<td>Branch</td>
</tr>
<tr>
<td>ASIL-D</td>
<td>MC/DC</td>
</tr>
</tbody>
</table>
ISO 26262
Robustness - testing is hard

- Coverage metrics helps you complete your test-suite
- Various levels exist

<table>
<thead>
<tr>
<th>Level of assurance</th>
<th>Coverage criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIL-A</td>
<td>Statement</td>
</tr>
<tr>
<td>ASIL-B</td>
<td>Statement or Branch</td>
</tr>
<tr>
<td>ASIL-C</td>
<td>Branch</td>
</tr>
<tr>
<td>ASIL-D</td>
<td>MC/DC</td>
</tr>
</tbody>
</table>

Coverage is *not* a sufficient metric for robustness tests!
function Absolute_Value (N : Integer) return Integer is begin
  if N >= 0 then
    return N;
  else
    return -N;
  end if;
end Absolute_Value;

Tests for MC/DC:

- -42, 123
function Absolute_Value (N : Integer) return Integer is begin
    if N >= 0 then
        return N;
    else
        return -N;
    end if;
end Absolute_Value;

Tests for MC/DC:
• -42, 123

Robustness around discontinuity:
• -1, 0, 1
function Absolute_Value (N : Integer) return Integer is begin
  if N >= 0 then
    return N;
  else
    return -N;
  end if;
end Absolute_Value;

Tests for MC/DC:
  • -42, 123

Robustness around discontinuity:
  • -1, 0, 1

Robustness around boundaries:
  • $-2^{31}, -2^{31} + 1$
  • $2^{31} - 1, 2^{31} - 2$
function Absolute_Value (N : Integer) return Integer
is begin
if N >= 0 then
  return N;
else
  return -N;
end if;
end Absolute_Value;

Tests for MC/DC:
• -42, 123

Robustness around discontinuity:
• -1, 0, 1

Robustness around boundaries:
• $-2^{31}$, $-2^{31} + 1$
• $2^{31} - 1$, $2^{31} - 2$
ISO 26262

Robustness - escalating complexity

foo
It is not plausible that humans can maintain this without making mistakes!
ISO 26262
Robustness - escalating complexity

\[0 \leq b < 128\]

\[a > 0\]
ISO 26262

Robustness - escalating complexity

It is not plausible that humans can maintain this without making mistakes!
ISO 26262
Robustness - escalating complexity

\[ c = 0 \lor c > 1 \]

\( \text{potato} \)

\( T \)

\[ 0 \leq b < 128 \]

\( \text{kitten} \)

\( T \)

\[ c > 0 \land a > 0 \]

\( \text{cat} \)

\( T \)

\[ a \neq b \]

\( \text{puppy} \)

\[ a < -1 \lor a > 1 \]

\[ -12 \leq b \leq 12 + a \]

\( \text{wibble} \)

\[ a > 0 \]

\( \text{foo} \)

\[ c \neq 0 \]

\[ 0 \leq b < 128 \]

\( \text{bar} \)

\[ x < y \]

\( \text{baz} \)

\[ x < y \]

It is not plausible that humans can maintain this without making mistakes!
ISO 26262
Robustness - escalating complexity

\[ c = 0 \lor c > 1 \]
\[ a > 0 \]
\[ c \neq 0 \]
\[ 0 \leq b < 128 \]
\[ 0 \leq b \leq 128 \]
\[ x < y \]
\[ x < y \]
\[ a < -1 \lor a > 1 \]
\[ -12 \leq b \leq 12 + a \]

It is not plausible that humans can maintain this without making mistakes!
ISO 26262

Robustness - escalating complexity

It is not plausible that humans can maintain this without making mistakes!
It is not plausible that humans can maintain this without making mistakes!
Software failure modes are non-obvious and system complexity is vast!
Spark can help here, even if you just do the absolute basics:

- Absence of run-time error proof
- Guarantee for type ranges
- Preconditions replace defensive code
Challenges

What makes adopting SPARK difficult?

- C and C++ is very established
- Emphasis on validation over verification
- Model-based design is sometimes seen as a magic bullet
- Platform issues (i.e. SPARK compiler availability)
How to make it work in automotive?
Generate C code!

SIMULINK

MATLAB

Embedded coder

SPARK

New

SPARK2C

C

C++

Your favourite compiler

Binary
Interfacing

• Interfacing SPARK / C is trivial
Interfacing

- Interfacing SPARK / C is trivial
- Interfacing SPARK / C++ is harder
  - C wrappers often required
Interfacing

• Interfacing SPARK / C is trivial
• Interfacing SPARK / C++ is harder
  • C wrappers often required
• Interfacing SPARK / SIMULINK?
What we have:

```plaintext
procedure Potato (Thing : in out Potato_Bus_T;
    A   : in Integer;
    B   : in Kittens);
```
Interfacing
SIMULINK

What we have:

```plaintext
procedure Potato (Thing : in out Potato_Bus_T;
    A : in Integer;
    B : in Kittens);
```

What we want:
Interfacing
Where to start?

Read the docs...

- There used to be an Ada binding (deprecated and dead)
Interfacing
Where to start?

Read the docs...

- There used to be an Ada binding (deprecated and dead)
- Bind to SIMULINK C API from Ada?
- Generate C code, bind to that from SIMULINK?
Interfacing
Where to start?

Read the docs...

- There used to be an Ada binding (deprecated and dead)
- Bind to SIMULINK C API from Ada?
- Generate C code, bind to that from SIMULINK?
- Try something else...
Interfacing

Compatible header files

Annotate interface spec:

```haskell
type Potato_Bus_T is record
   Arr : Int_Array_T;
   Num : Integer;
   Bus_Arr : Bus_Array_T;
   Kitten : Kittens;
end record
with Convention => C_Pass_By_Copy;
```

```c
typedef struct {
   int arr[25];
   int num;
   point bus_arr[25];
   kittens kitten;
} potato_bus;
```

Ideally generate both from a neutral format.
Interfacing
A thin binding

Create a small C wrapper for SIMULINK:

```c
void my_fun(potato_bus *u1, int u2, kittens u3, potato_bus *y1)
{
    memcpy(y1, u1, sizeof(potato_bus));
    ada__potato(y1, u2, u3);
}

void my_init()
{
    potatoinit();
}

void my_finish()
{
    potatofinal();
}
```
Interfacing

Glue code

Use the SIMULINK legacy code binding generator setup glue code:

```matlab
Simulink.importExternalCTypes('src/potato_bus.h');

def = legacy_code('initialize');
def.Options.language = 'C';
def.Options.isVolatile = false;
def.SFunctionName = 'potato';
def.SourceFiles = {'my_fun.c'};
def.HeaderFiles = {'potato_bus.h'};
def.OutputFcnSpec = ['void my_fun(potato_bus u1[1], uint8 u2, ' ...
    'kittens u3, potato_bus y1[1])'];
def.StartFcnSpec = ['void my_init()'];
def.TerminateFcnSpec = ['void my_finish()'];

legacy_code('sfcn_cmex_generate', def);
movefile('potato.c', 'src/potato.c');
```
Build it all via gprbuild:

- Don’t use the mex utility function
- Works on Windows and Linux
- Build DLL / SO
  - Without auto init section
  - With a special linker script
  - With special treatment of interrupts (on Linux)
Interfacing

Building

Build it **all** via gprbuild:

- Don’t use the mex utility function
- Works on Windows and Linux
- Build DLL / SO
  - Without auto init section
  - With a special linker script
  - With special treatment of interrupts (on Linux)
- Result is a mixed language .mexw64 / .mexa64 plugin that “just works”.

Spark in an automotive context
June 3, 2019
Build it all via gprbuild:

• Don’t use the mex utility function
• Works on Windows and Linux
• Build DLL / SO
  • Without auto init section
  • With a special linker script
  • With special treatment of interrupts (on Linux)
• Result is a mixed language .mexw64 / .mexa64 plugin that “just works”.
• A lot of this can be automated!
Now all you need to do is this:

```ada
procedure Potato (Thing : in out Potato_Bus_T;
    A : in Integer;
    B : in Kittens)
with
    Export,
    Convention => C,
    External_Name => "ada__potato",
    Annotate => Simulink_Block;
```

- Python script based on gnat2xml works out data types and a list of subprogram to build bindings for
- Other script generate both C header and Ada type definitions
- No manual work involved!
Conclusion

**Spark** in the automotive industry:

- Great fit with ISO 26262 (especially robustness)
- Interfacing with existing tools and environments is possible
- Incremental adoption possible: integrates well with existing environments and systems
Conclusion

**Spark in the automotive industry:**

- Great fit with ISO 26262 (especially robustness)
- Interfacing with existing tools and environments is possible
- Incremental adoption possible: integrates well with existing environments and systems

---

Thank you for listening.

*Questions?*