Some concerns about safety for software-intensive systems , and an Introduction to STPA

Paul Sherwood 24 Oct 2018
@devcurmudgeon
www.devcurmudgeon.com

NB: Codethink-branded slides are mine, the rest are from MIT

Intro: @devcurmudgeon

- CEO Codethink (.com)
- Stealing Spitfires (Spotify)
- Shut Up And Shoot Me (IMDB)
- Software Commandments (github)
- YBD: Yaml Build Deploy (gitlab)
- www.devcurmudgeon.com

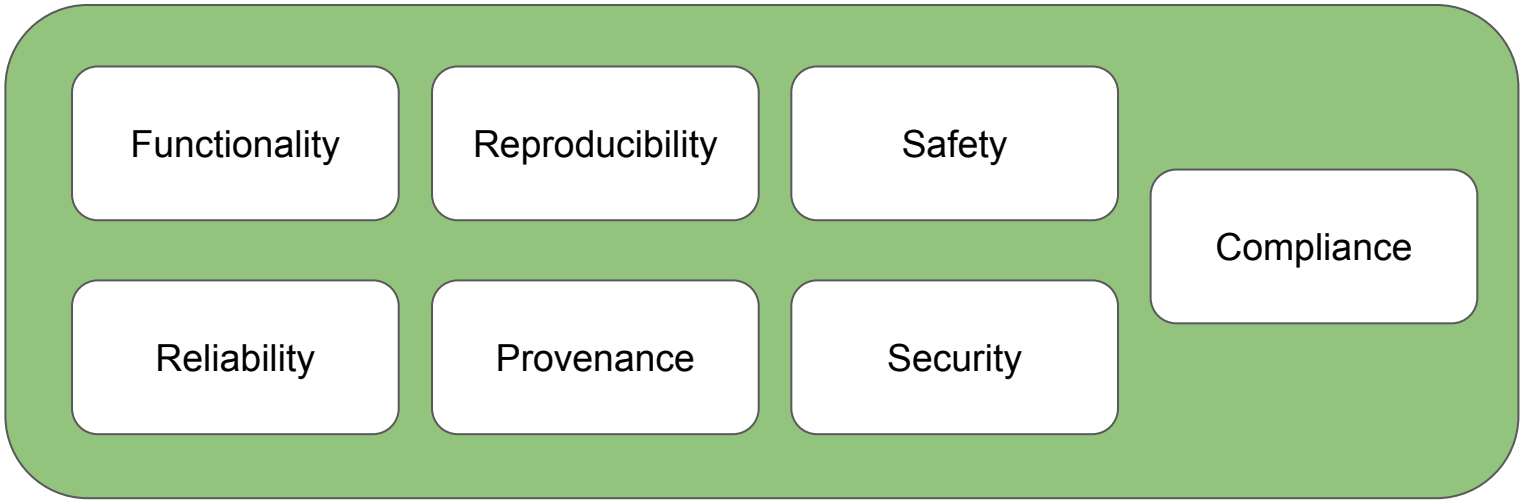
- python/ruby/git/C and vi
- skeptical, opinionated and grumpy
- with trust issues
- insisting on honesty

Intro: Codethink

Who has read any of the safety standards?

CodeThink

Working hypothesis: software trustability factors



we could base our trust on evidence for each/all of these

https://trustable.io

CodeThink

Patch me, if you can: Grave TCP/IP flaws in FreeRTOS leave IoT gear open to mass hijacking

AWS-stewarded net-connected platform has multiple remote code execution vulnerabilities

By Shaun Nichols in San Francisco 22 Oct 2018 at 20:05 13 SHARE ▼

Serious security flaws in FreeRTOS – an operating system kernel used in countless internet-connected devices and embedded electronics – can be potentially exploited over the network to commandeer kit.

Subaru Destroys 293 Ascent SUVs After Coding Error Leads to Unsafe Cars

A coding error led robots to miss welds on 293 of Subaru's Ascent 2019 SUVs.

By Jessica Miley

September, 27th 2018

462 shares

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

CodeThink

You thought Dieselgate was over? It's not.

The scandal of Volkswagen caused political turmoil in Germany

By Wolfgang Kerler | Sep 18, 2018, 5:46pm EDT

The number of recalls linked to electronic failures has risen by 30 per cent a year since 2012, compared with an average of 5 per cent a year between 2007 and 2012, according to data from consultancy AlixPartners.

Our current tools are all 40-65 years old but our technology is very different today

1940 1950 1960 1970 1980 1990 2000 2010 2020

FMEA

FTA HAZOP ETA

Bow Tie
(CCA)
FTA + ETA

- Introduction of computer control
- Exponential increases in complexity
- New technology
- Changes in human roles

Assumes accidents caused by component failures

safety standards (IEC 61508, ISO 26262, MISRA C etc)

Codethink

- expensive, not public, protected by strange EULAs
- mostly arose incrementally from mech eng reliability
- graduated to simple electronics, then microcontrollers
- ... and then defined rules for the software that could be trusted to run on microcontrollers (e.g. MISRA C)
- lots of special language (e.g. “...out of...”)

The underlying principles are:

- “make your components reliable”
- assure software by enforcing 90s style engineering process

safety standards

Some dangerous misunderstandings have arisen:

- treat microprocessors as big microcontrollers
- choose pre-certified software for its magical safety powers
- combine 2 ASIL B components to achieve ASIL D
- safety design can be achieved via component reliability

these are all fundamentally WRONG

Software for Safety: 80s/90s

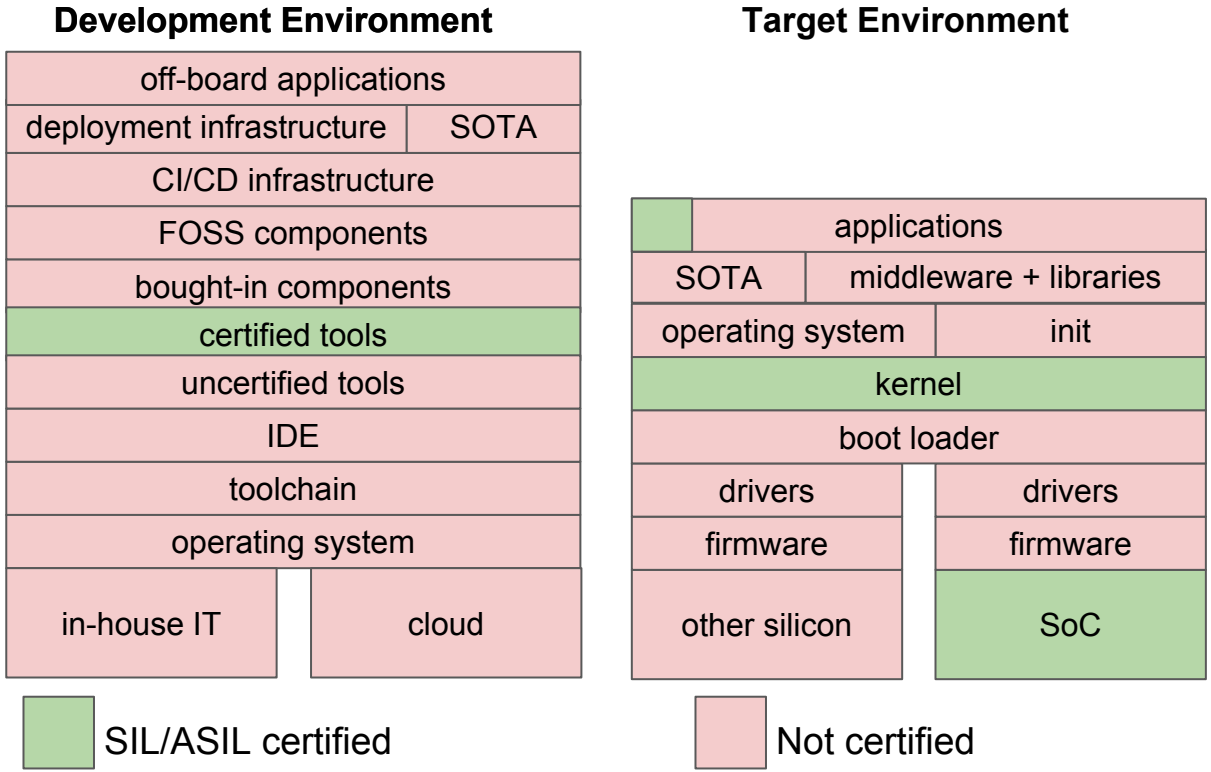
Development Environment

Target Environment

SIL/ASIL certified

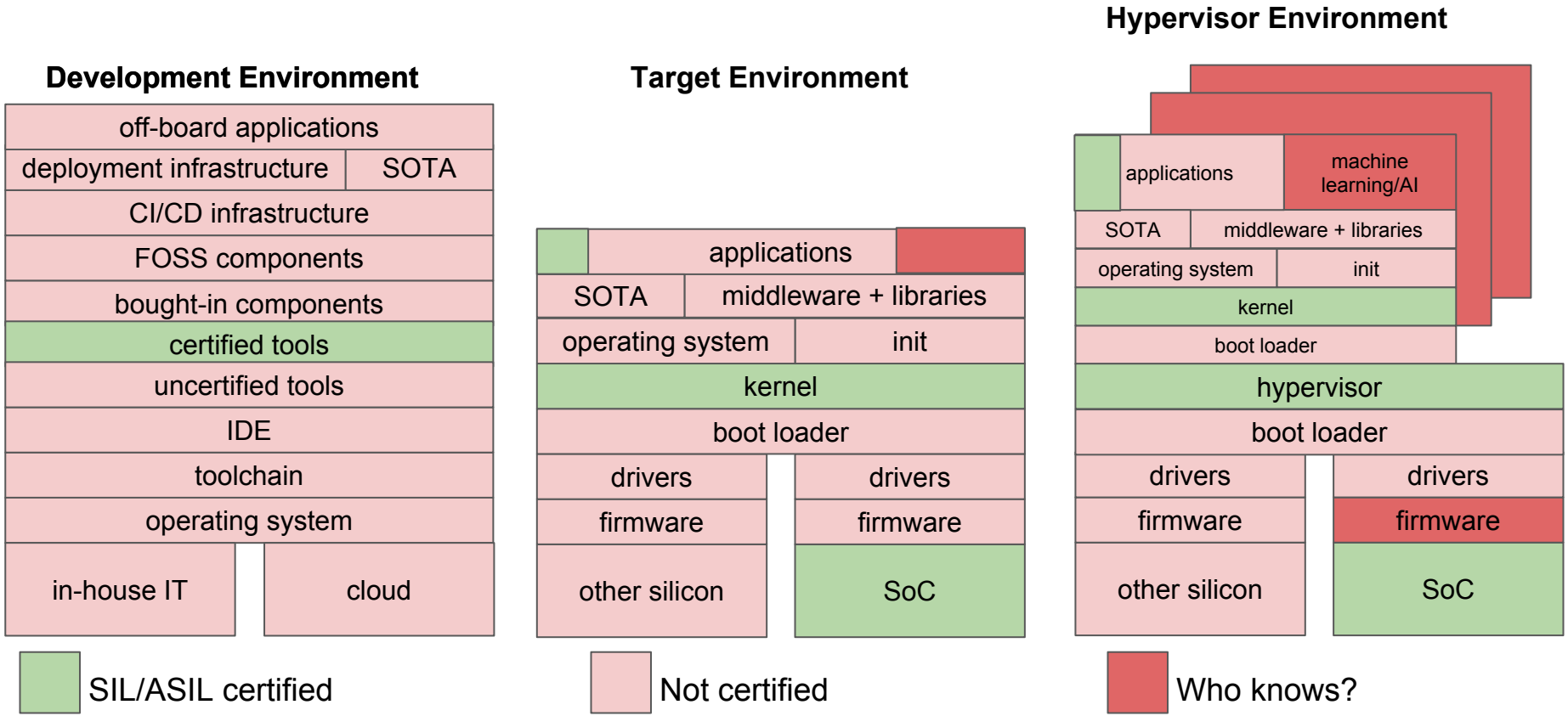
Software for Safety: as time goes by...

(we need to think about all of parts, not just the kernel and some MISRA C)



Software for Safety: 2018

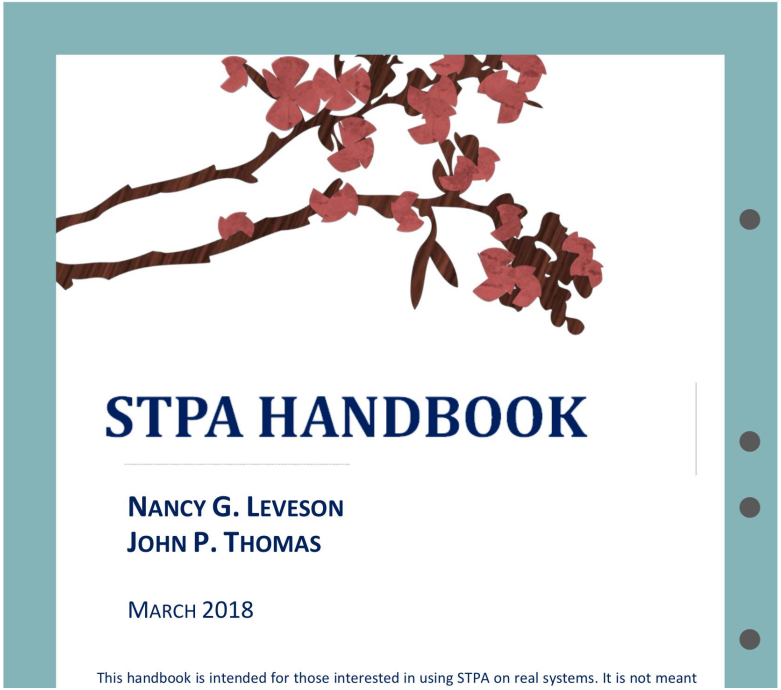
(safety for connected devices involves security, obviously...)



Safety has to evolve to handle complex software...

Electromechanical **safety and reliability** requirements (for seatbelts,airbags, brakes, steering, lights etc)	**Simple** electronics and software **safety and reliability** requirements	**Complex** electronics and software **safety and trustability** requirements

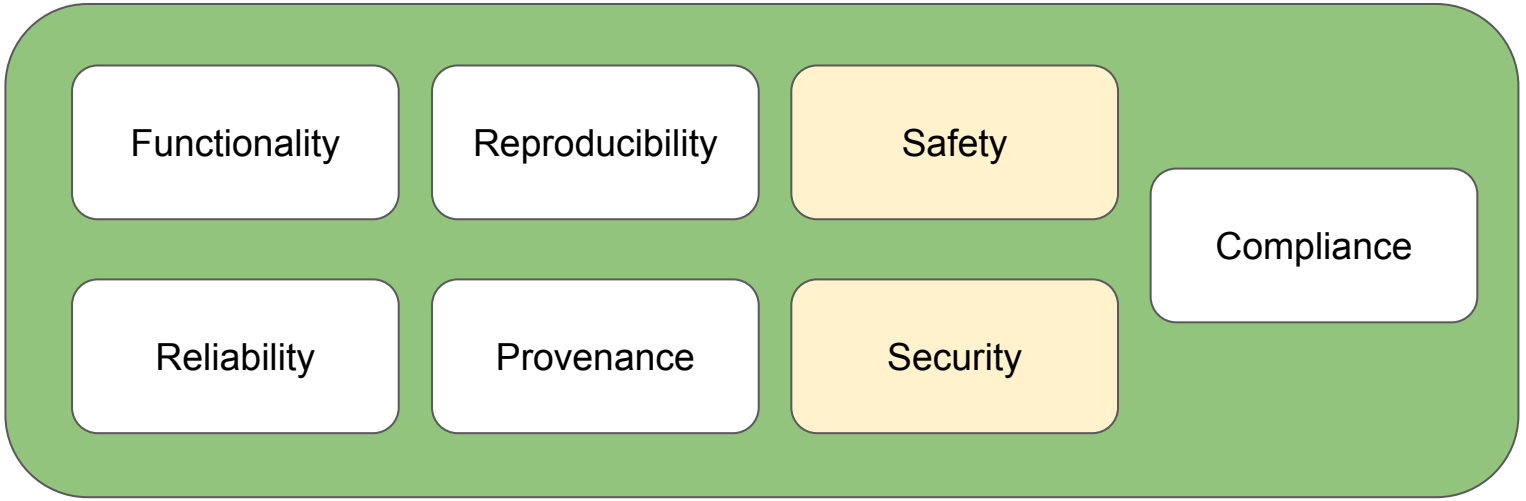
We **can't** guarantee behaviour of software at scale. So safety designs need to **expect** misbehaving software



http://psas.scripts.mit.edu/home/

- Increasingly recalls/accidents are due to:
  - specification/requirements errors
  - interactions between components
- Safety is not the same as reliability
- Safety is a system property, not a component property
- A system composed of reliable components is not necessarily safe

Working hypothesis: software trustability factors



safety and security are (emergent) system properties, not just software

STAMP

Model framework

STPA

requirements analysis

CAST

accident investigation

STPA: systematic **top-down** analysis

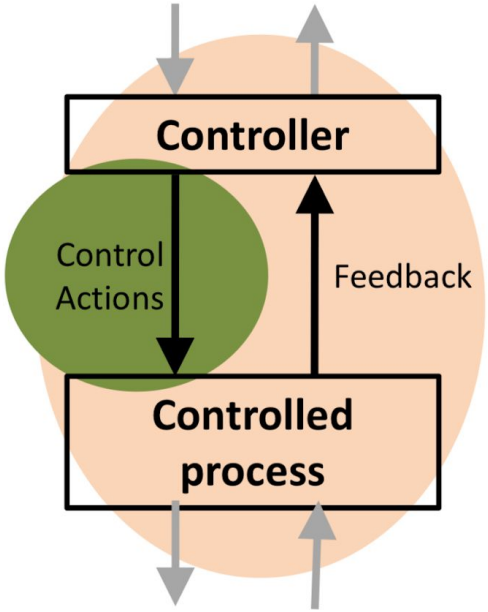
- Applicable for both safety and security design
- Led by MIT, increasingly adopted in automotive and other industries
- Some standards are now taking this approach STAMP/STPA

Massachusetts Institute of Technology

STPA Hazard Analysis

STAMP Model

- System engineering foundation
  - Define accidents, system hazards,
  - Control structure
- Step 1: Identify unsafe control actions
- Step 2: Identify accident causal scenarios



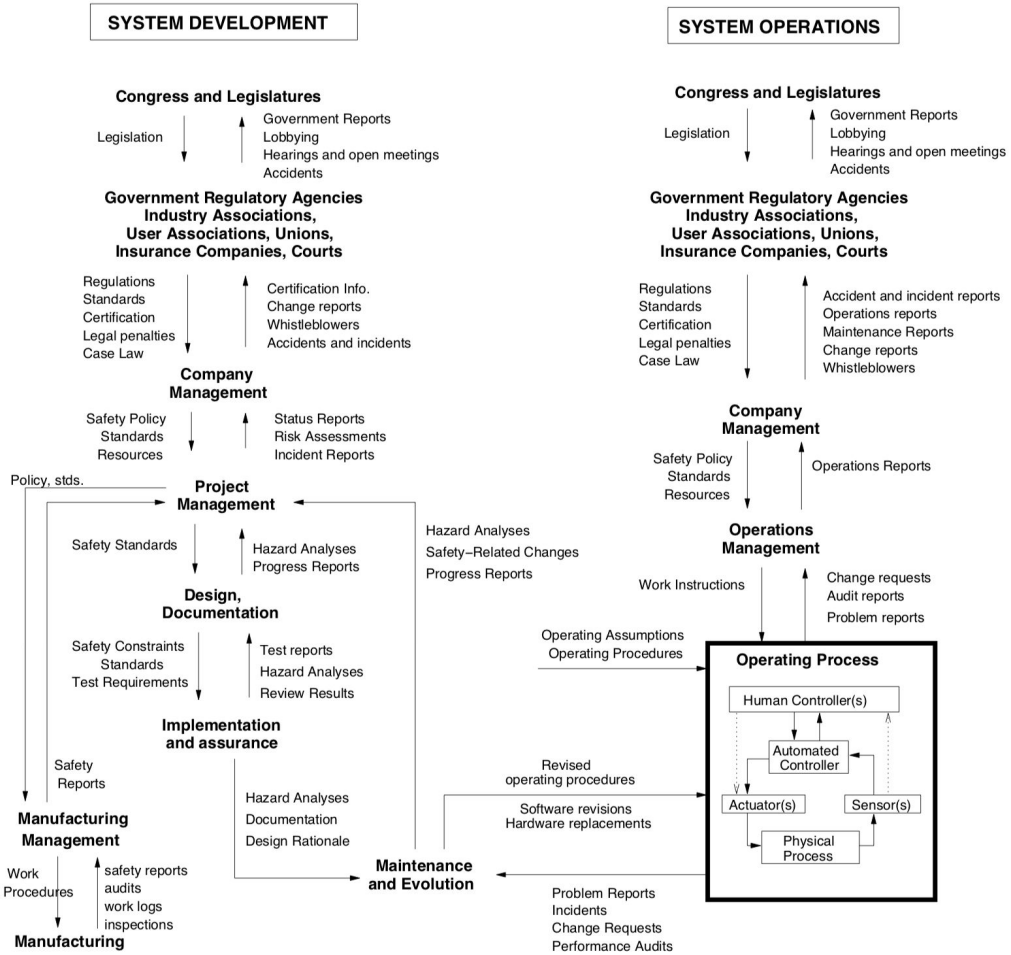
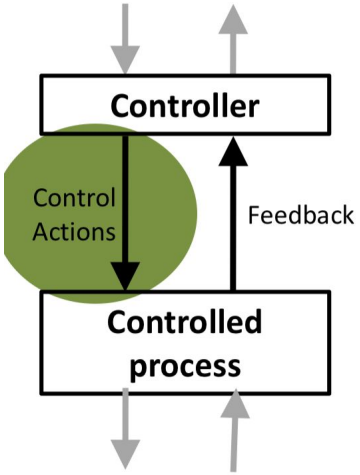


Figure 4.4: General Form of a Model of Socio-Technical Control

STPA Step 1: Unsafe Control Actions (UCA)

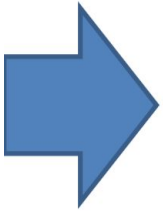


4 ways unsafe control may occur:

- A control action required for safety is not provided or is not followed
- An unsafe control action is provided that leads to a hazard
- A potentially safe control action provided too late, too early, or out of sequence
- A safe control action is stopped too soon or applied too long (for a continuous or non-discrete control action)

	Not providing causes hazard	Providing causes hazard	Incorrect Timing/ Order	Stopped Too Soon / Applied too long
Shifter Command	?	?	?	?

STPA Step 2: Identify Causal Factors



- Select an Unsafe Control Action

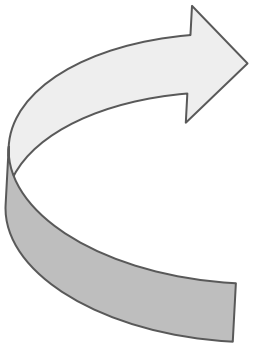
A. Identify what might cause it to happen
   - Develop accident scenarios
   - Identify controls and mitigations

B. Identify how control actions may not be followed or executed properly
   - Develop causal accident scenarios
   - Identify controls and mitigations

STPA Method: applicable before, during, after design

Losses => Hazards => Control Diagram => Controllers, Signals, Feedback

For each controller, signal, feedback:



Identify Unsafe Control Actions:

Controller + Action + Type + Context

Establish Requirements:

Negate the UCAs

And then iterate to refine the details from control diagram to requirements

STPA Method: example

Losses

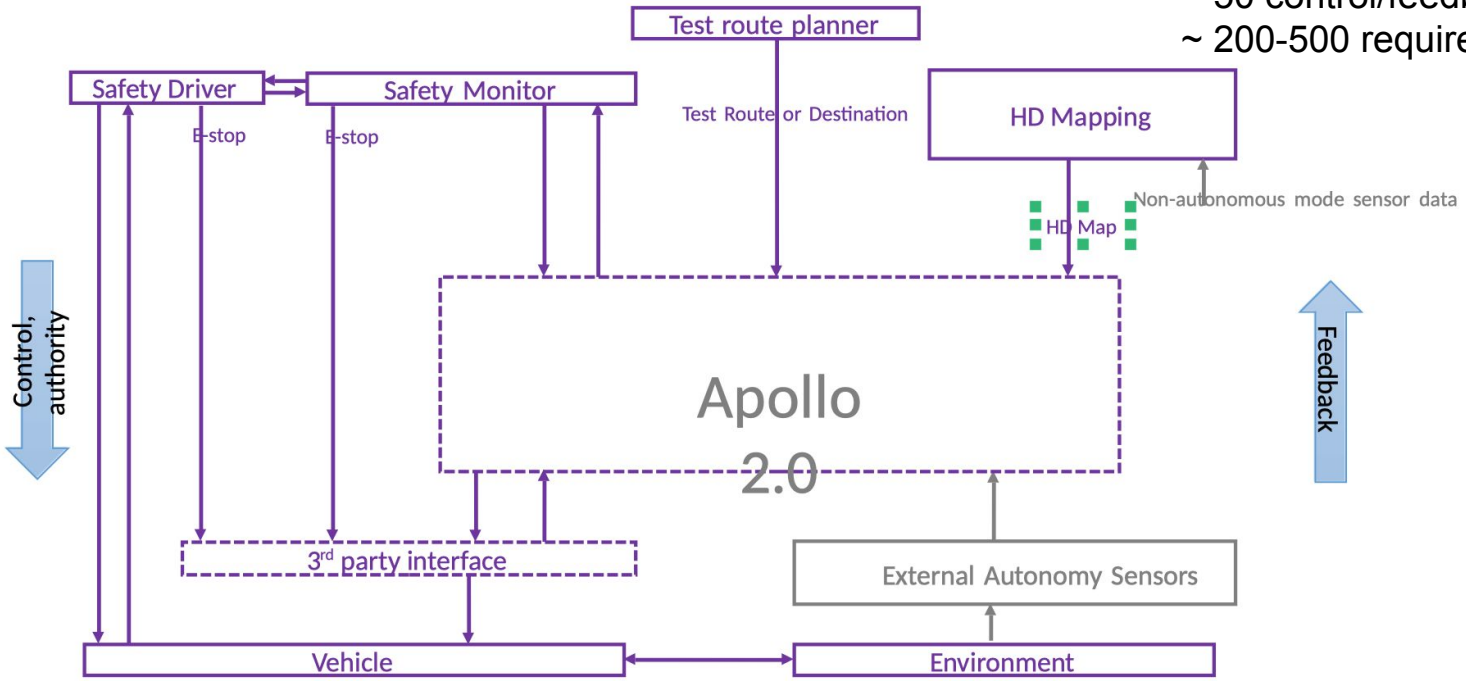
- L-1 : Loss of life or injury to people
- L-2 : Loss of or damage to vehicle
- L-3 : Loss of or damage to objects outside the vehicle
- L-4 : Loss of transportation mission
- L-5 : Loss of traffic flow (road blockages etc.)
- L-6 : Loss of customer satisfaction
- L-7 : Environmental impact

Hazards

- H-1 : Vehicle does not maintain safe distance from terrain and other obstacles [L-1, L-2, L-3, L-4, L-5, L-6]
- H-2 : Vehicle drives too fast [L-1, L-2, L-3, L-4, L-5, L-6, L-7]
- H-3 : Excessive braking [L-1, L-2, L-3, L-4, L-5, L-6, L-7]
- H-4 : Vehicle does not follow traffic flow e.g. jumps red lights, drives on wrong side of the road [L-1, L-2, L-3, L-4, L-5, L-6, L-7]
- H-5 : Vehicle is unpredictable to others e.g. no indicators, drives on wrong side of road [L-1, L-2, L-3, L-4, L-5, L-6]

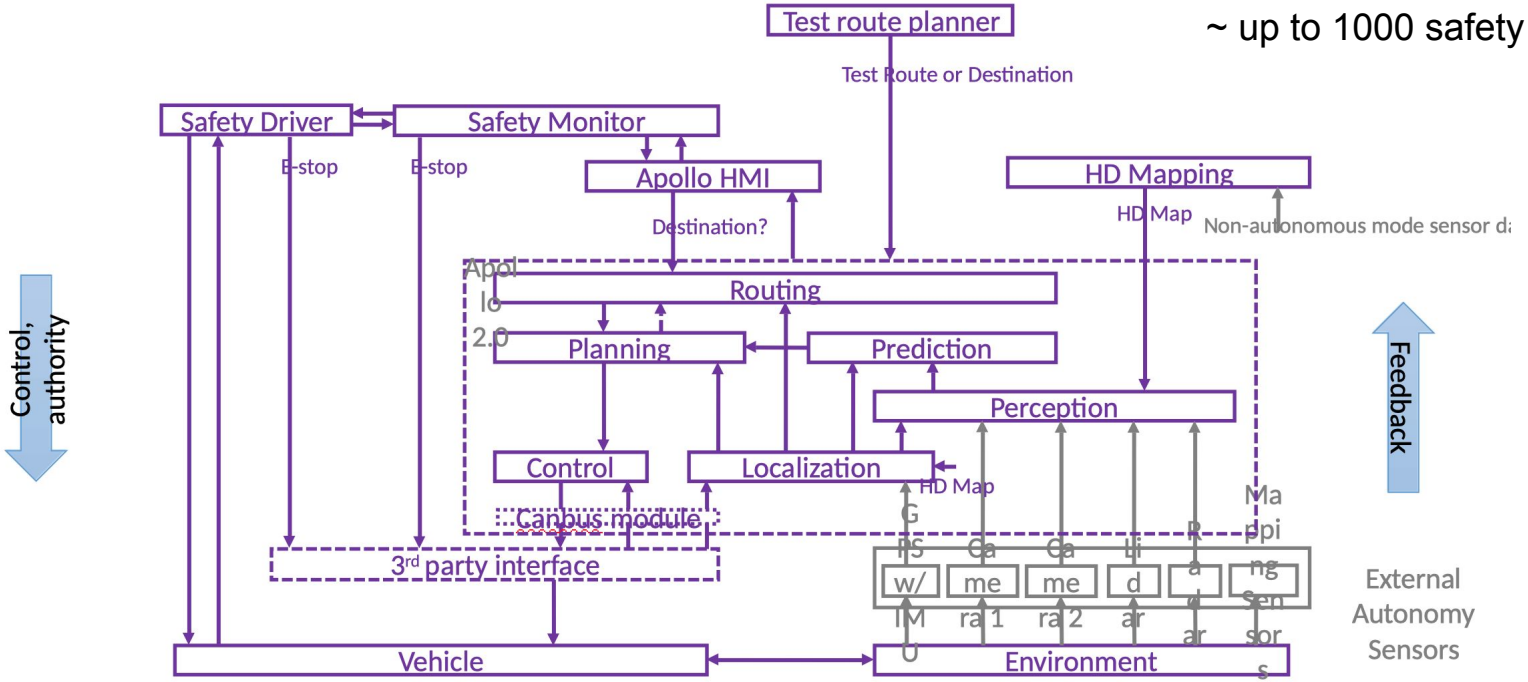
STPA Method: example continued

Scale/complexity
9 boxes
~ 20 arrows
~ 50 control/feedback signals
~ 200-500 requirements?



STPA Method: example continued

Scale/complexity
20 boxes
~ 40 arrows
~ 100 control/feedback signals
~ up to 1000 safety requirements?



STPA Method ... thoughts so far

- control architecture is easier to analyse than physical/logical
- in theory we can get to a complete set of safety requirements
- this is systems engineering, not just software
- must involve analysis and mapping of losses => requirements => design
- iteration is involved: we need tooling with version control, reviews etc
- the current foss-applicable tools are not great (so folks use visio, excel, word)
- not enough actual analyses have been made public

- there is no magic
- but top down is IMO the only sensible startpoint