
TSD

# *Towards Complete Embedded System Modeling and Generation*

**Dr.-Ing. Klaus Uhl**
**Dr.-Ing. Kay-Ulrich Scholl**
**Intel Corporation, GENIVI Charter Member**

**GENIVI 14th All Member Meeting, Paris, April, 28th, 2016**

# *Overview*

## The present

- Component modeling and code generation
- Intel demo system

## The future

- System modeling and code generation
- Data streaming

# *Component Modeling and Code Generation*

# *Motivation for a component generator*

## Most components share common patterns of code

- Configuration parameter processing
- IPC initialization
- Interface instantiation
- Consolidate information from multiple interfaces
- etc.

## Manually maintaining this code is tedious and error-prone

## Code generation helps to

- Find bugs in the generated code faster because the same code is used in many components
- Fix bugs simultaneously in many components by just updating the generator
- Roll out implementation improvements or adpations to API changes simultaneously to many components by just updating the generator

# *Intel Component Generator Features*

- Component **startup** and **shutdown** handling
- **Trigger other generators** depending on the component definition
- Configuration **parameter** handling
- **Communication** interface instantiation and initialization
  - Multiple channels
  - Multiple IPC mechanisms
- Event and worker **thread** initialization
- Forwarding of calls from multiple interfaces to a single **handler class**
- **DLT** application and context registration
- Link to **GENIVI lifecycle**

# Example

```
package calculator.example.server

import calculator.example.* from "Calculator.fidl"

module CalculatorServer {
  dlt context use parent;
  communication channel mainChannel;
  provides interface Calculator calculator
           on mainChannel;
}

component CalculatorServerApp {
  dlt app id ICES "Calculator Example Server";
  dlt context _CSC "Calculator Server Context";
  contains CalculatorServer calculatorServer
  communication channel MainBus {
    type UFIPC;
    maps to calculatorServer.mainChannel;
  }
  event thread mainThread {
    processes MainBus;
  }
  use genivi_lifecycle;
}
```

# Example

- Change only the type of a communication channel

```
package calculator.example.server

import calculator.example.* from "Calculator.fidl"

module CalculatorServer {
  dlt context use parent;
  communication channel mainChannel;
  provides interface Calculator calculator
           on mainChannel;
}

component CalculatorServerApp {
  dlt app id ICES "Calculator Example Server";
  dlt context _CSC "Calculator Server Context";
  contains CalculatorServer calculatorServer
  communication channel MainBus {
    type UFIPC;
    maps to calculatorServer.mainChannel;
  }
  event thread mainThread {
    processes MainBus;
  }
  use genivi_lifecycle;
}
```

# Example

- Change only the type of a communication channel
- Component generator instructs the build system to
  - Generate different proxy and stub adapters
  - Link against a different IPC binding runtime library

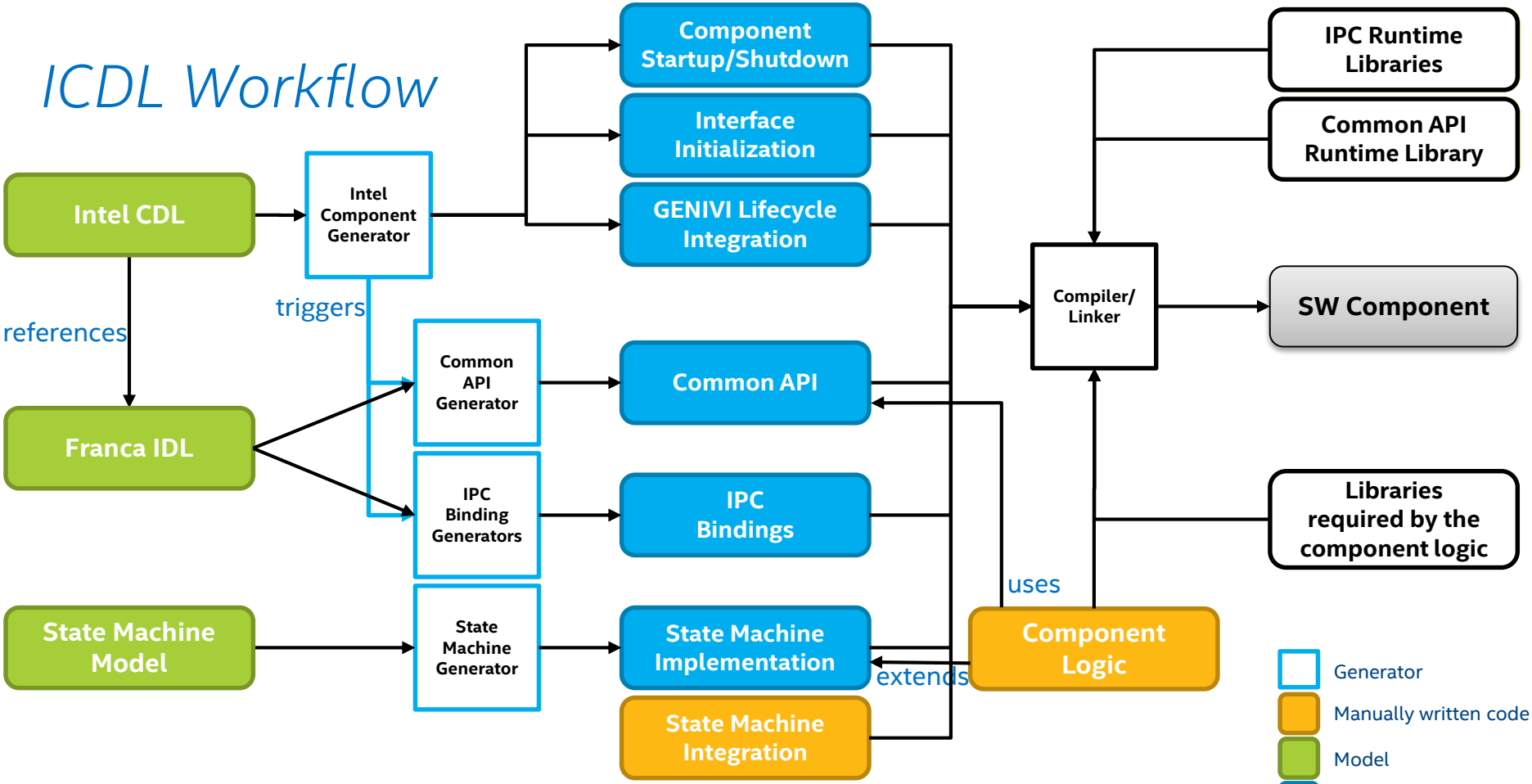
```
package calculator.example.server

import calculator.example.* from "Calculator.fidl"

module CalculatorServer {
  dlt context use parent;
  communication channel mainChannel;
  provides interface Calculator calculator
           on mainChannel;
}

component CalculatorServerApp {
  dlt app id ICES "Calculator Example Server";
  dlt context _CSC "Calculator Server Context";
  contains CalculatorServer calculatorServer;
  communication channel MainBus {
    type Dbus;
    maps to calculatorServer.mainChannel;
  }
  event thread mainThread {
    processes MainBus;
  }
  use genivi_lifecycle;
}
```

# ICDL Workflow

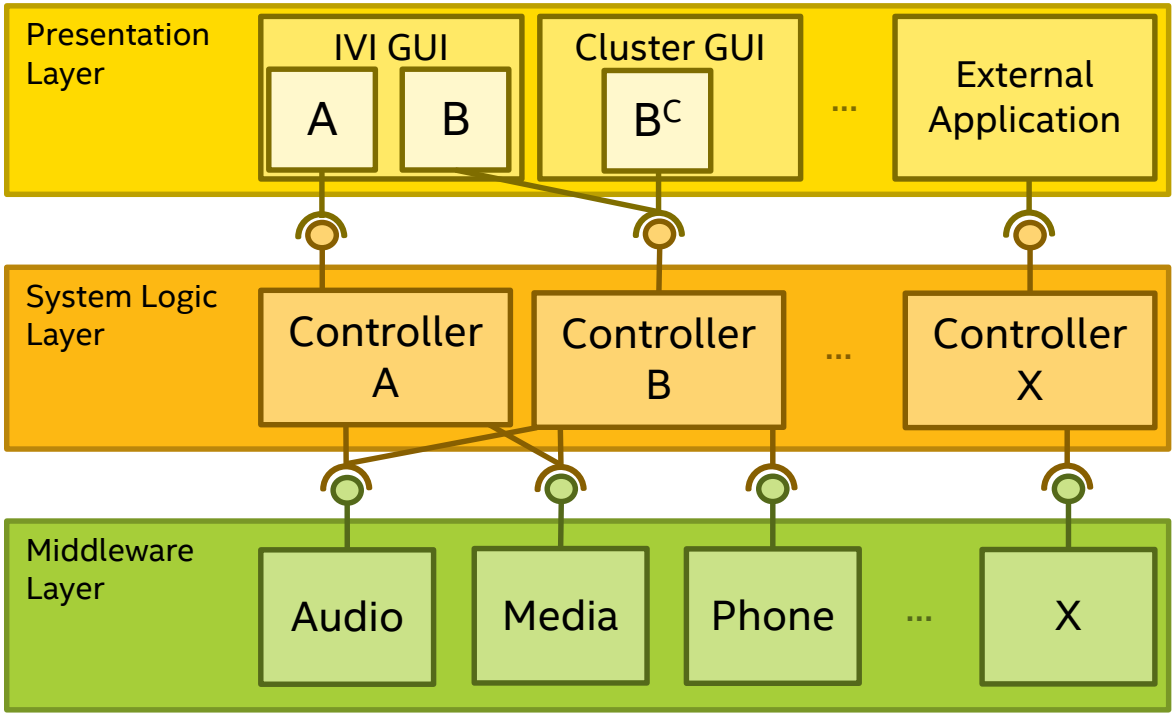



Reduce the amount of manually maintained code

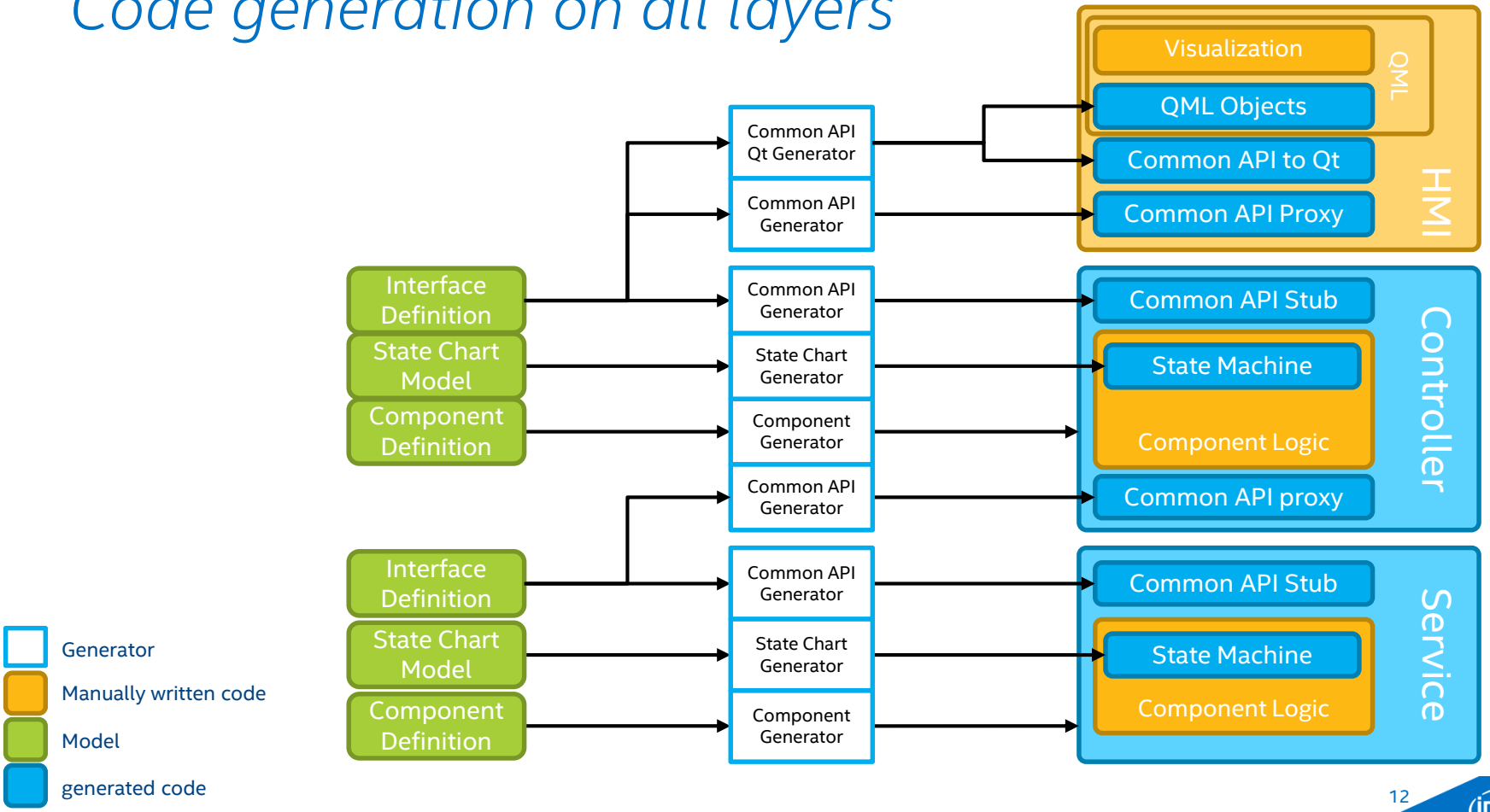
# *Intel Demo System*

# System Architecture

- Multiple UI runtimes possible (GUI, Cluster, Speech, etc.)
- Inter-process communication API defined by Franca IDLs
- API syntax is UI runtime specific



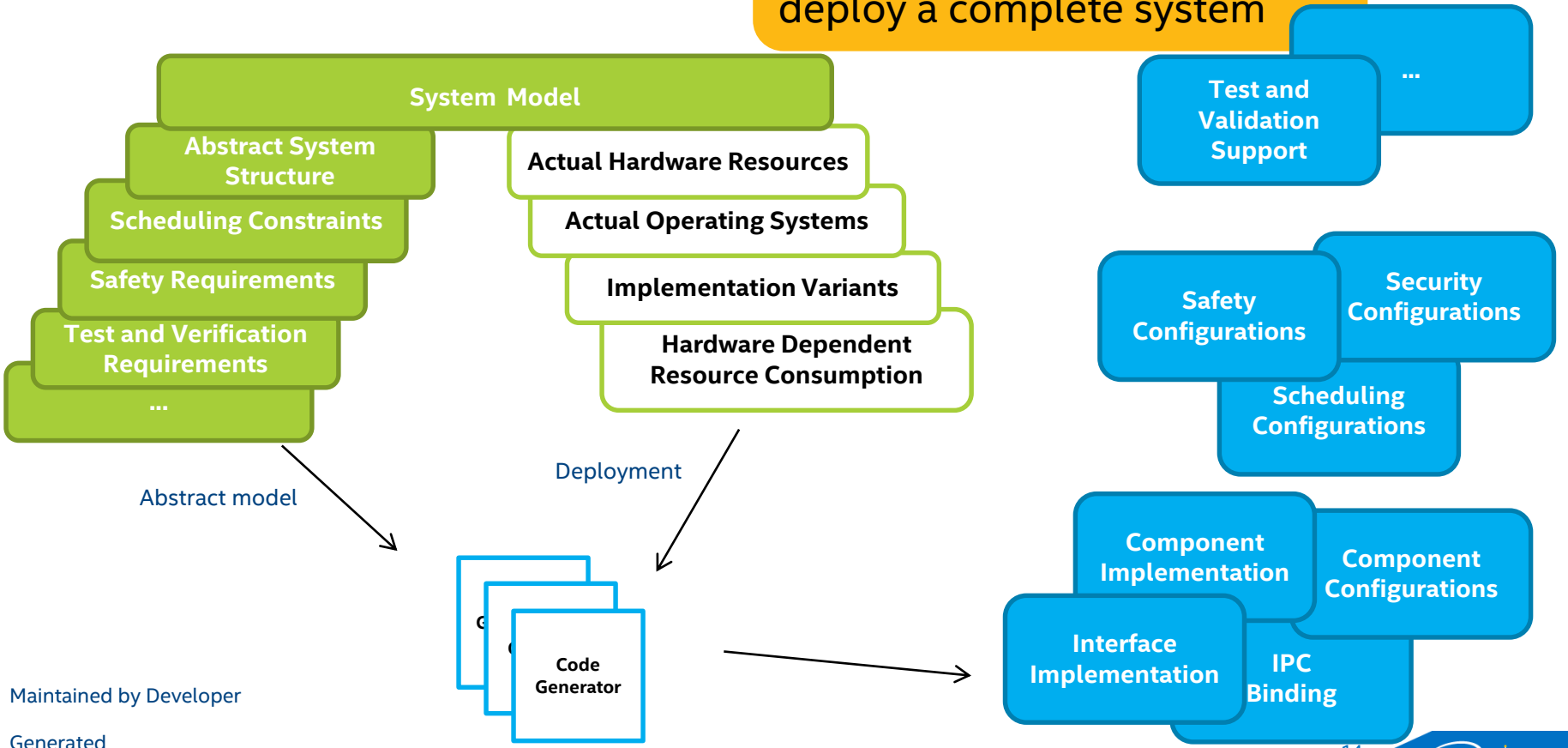
# Code generation on all layers



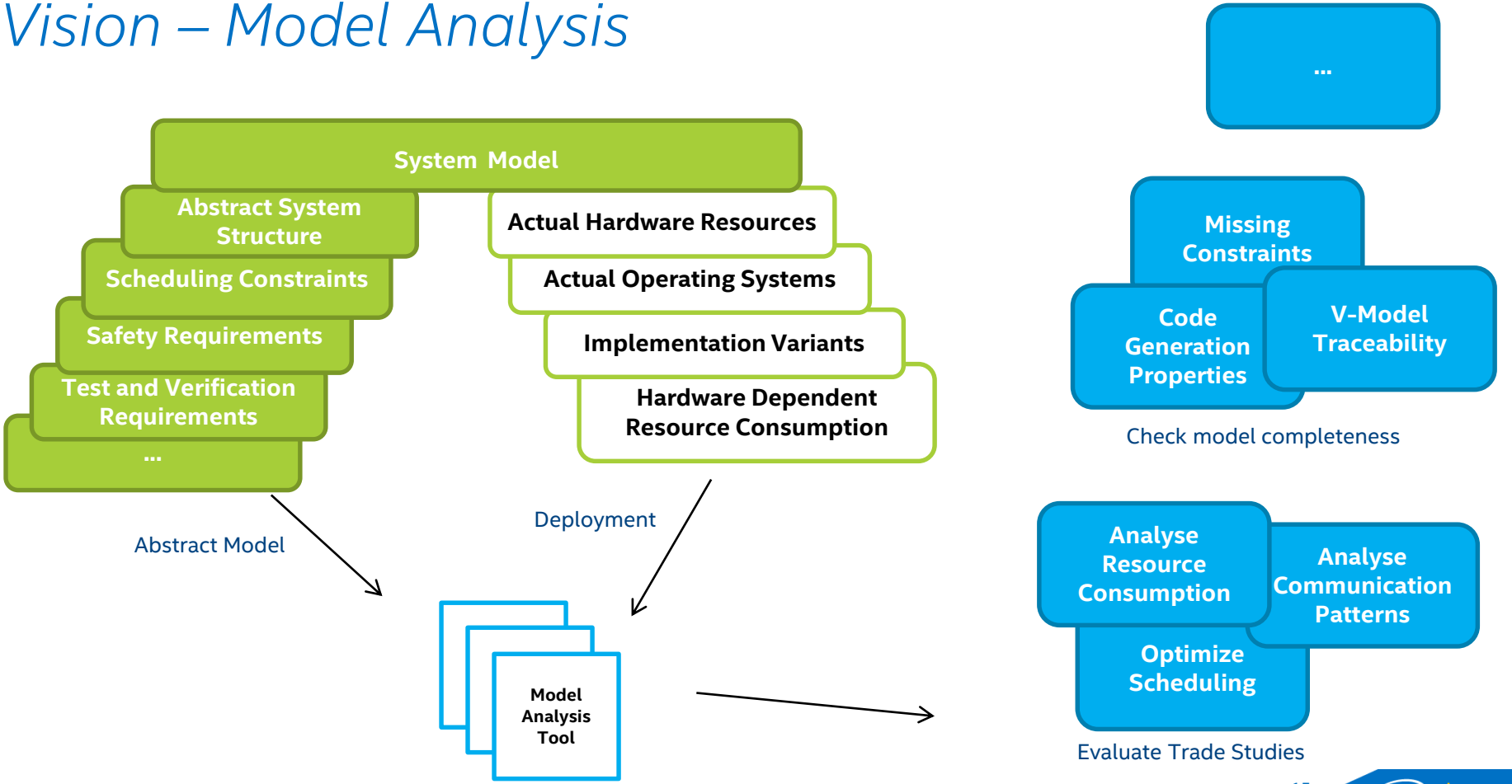
# *System Modeling and Code Generation*

# Vision – Code Generation

System architect can quickly re-configure, re-build and re-deploy a complete system

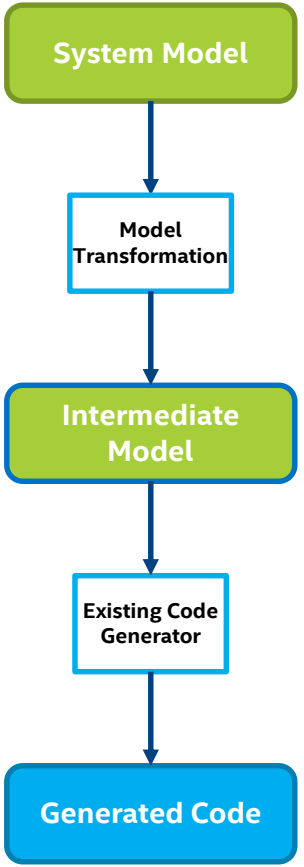


# Vision – Model Analysis

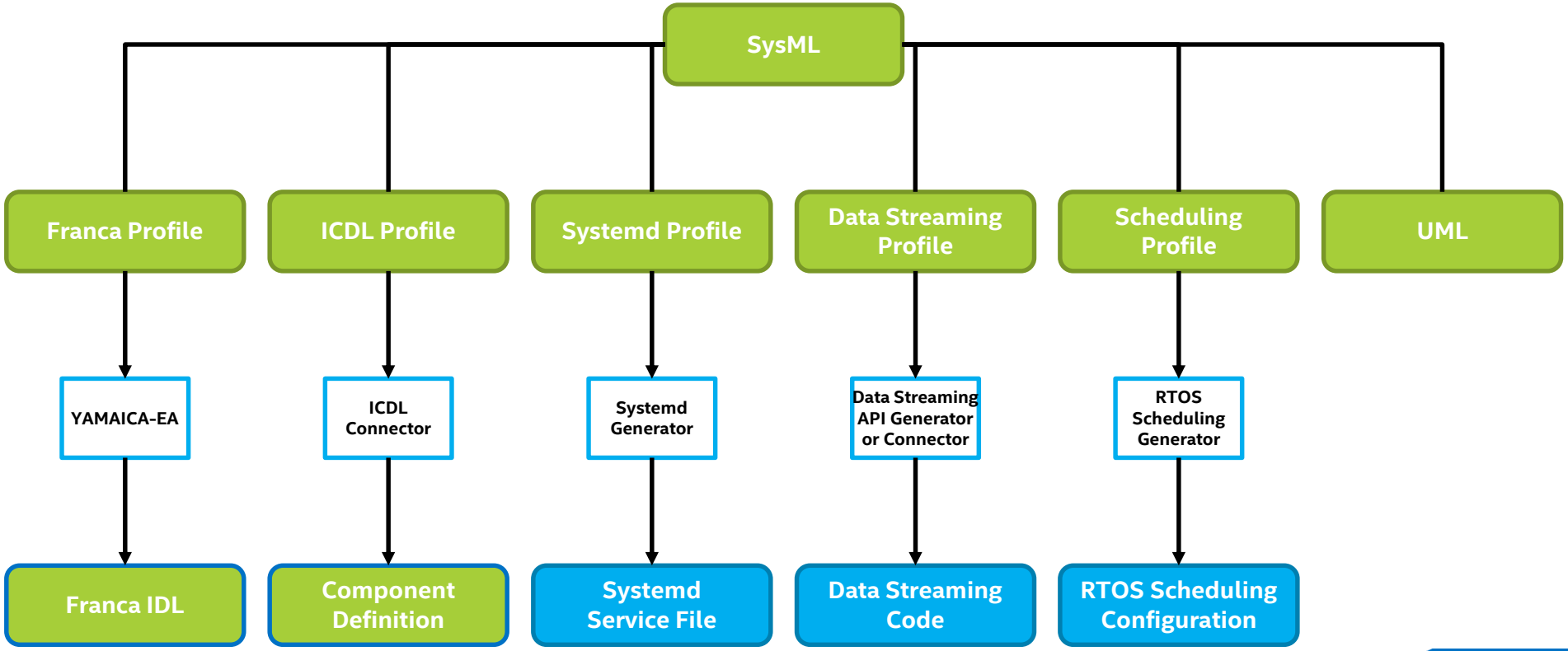


# *Facilitate Technology Reuse*

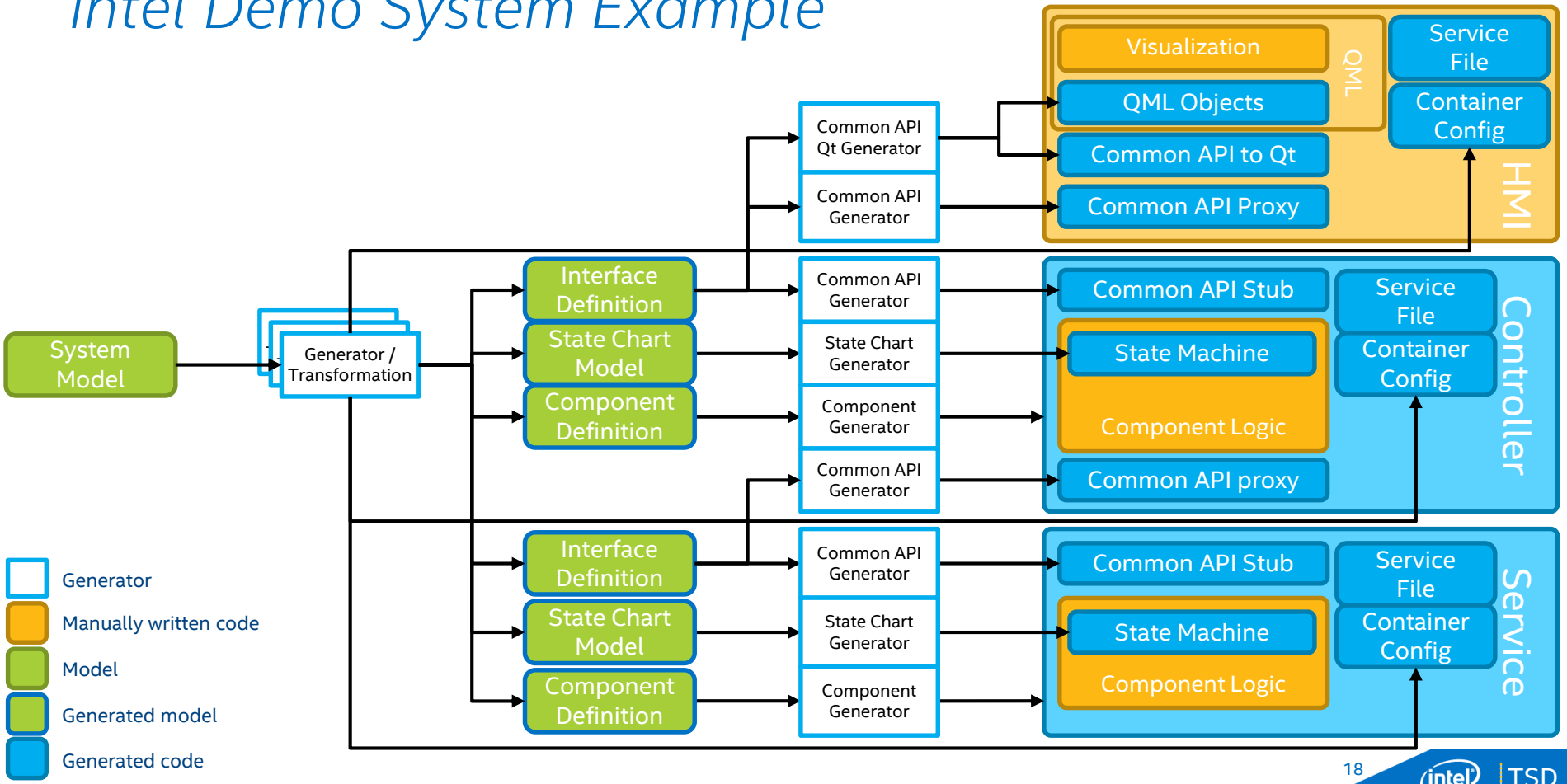
- Link system model entities to models that are defined in other languages
- Extend the system model to subsume the external models
- Re-use existing code generators
  - Export external models from system model on the fly during code generation if possible
  - Use import/export only if needed
- Only implement new code generators if no usable alternative exists



# System Modeling and Generation Approach



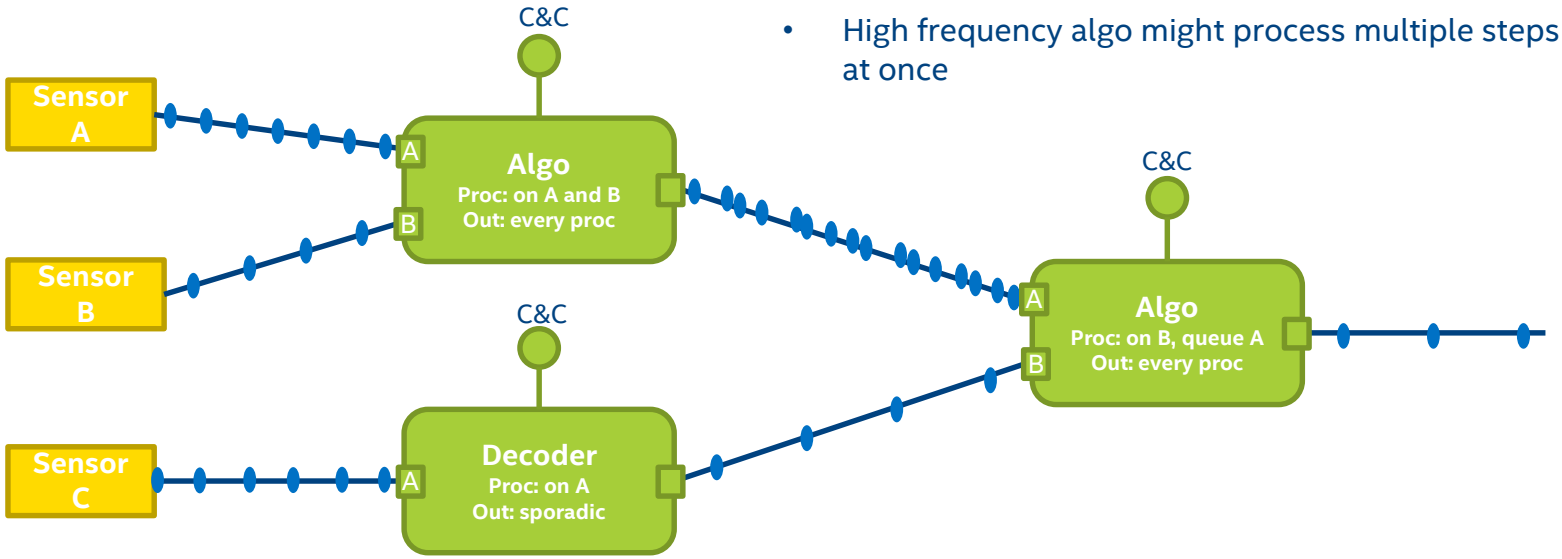
# Intel Demo System Example



# *Data Streaming*

# Data Streaming

- Data synchronization
  - input queuing
  - processing trigger
- Algorithm command & control
- Scheduling
  - Real-time constraints
  - High frequency algo might process multiple steps at once

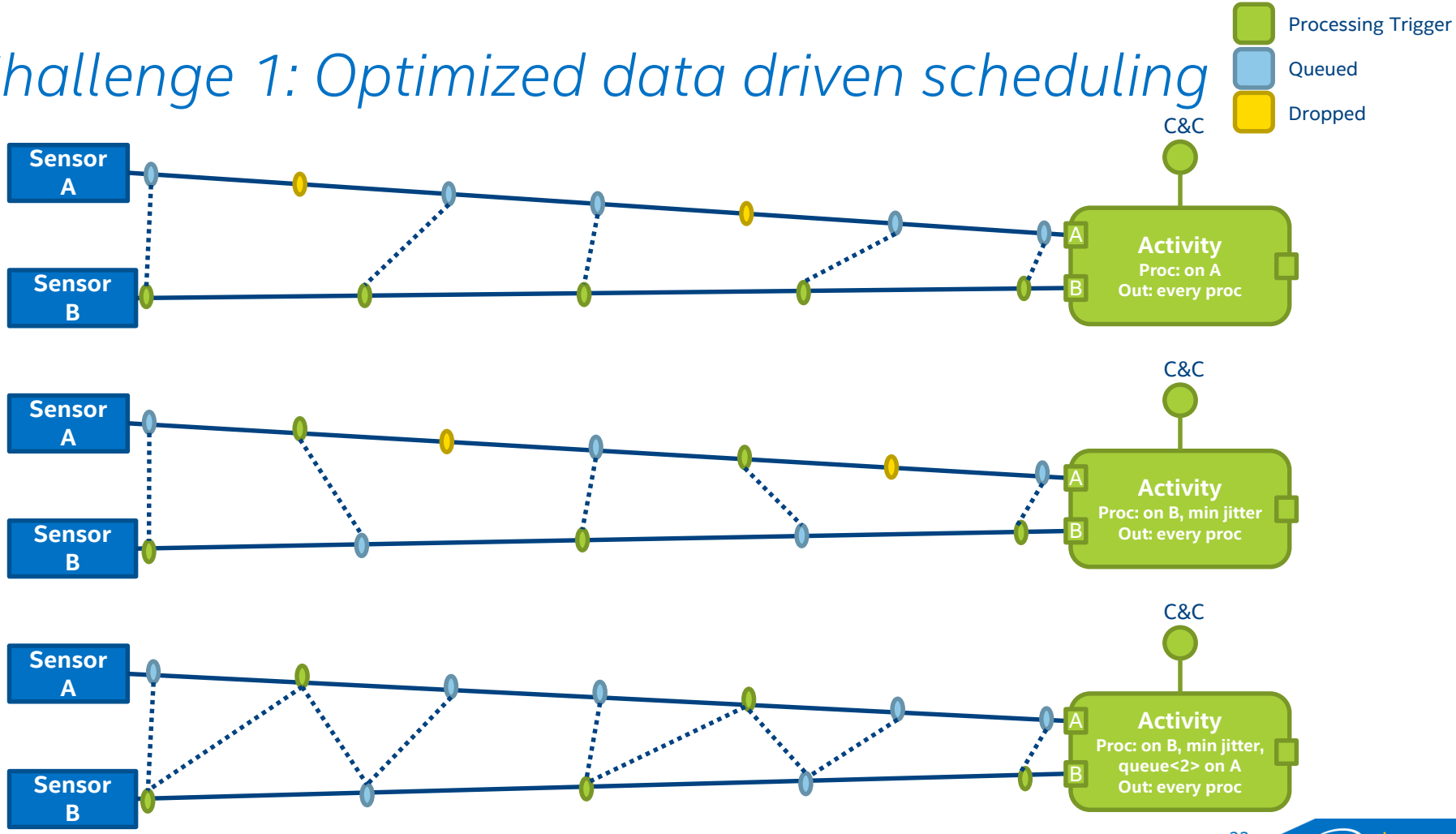



Far more complex than command & control

# *Differences to CommonAPI*

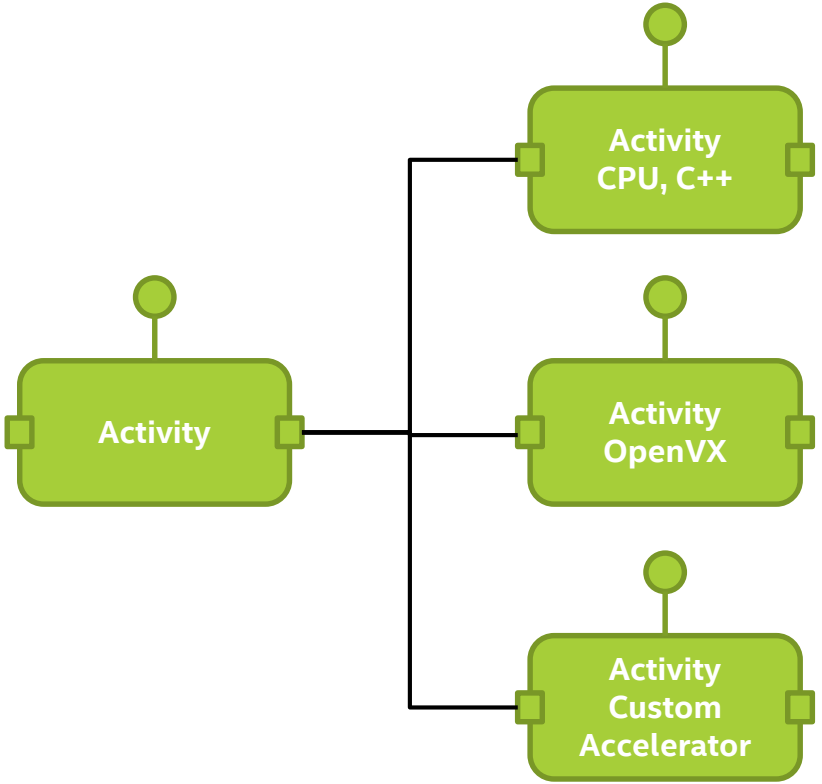
| Franca/CommonAPI | Data Streaming API |
|---|---|
| Client/server communication model<br>• Attributes<br>• Remote function calls with return value<br>• Events with subscription | Publish/subscribe communication model<br>• No attributes required<br>• No remote function calls required |
| Only one IPC mechanism per interface supported | Multiple IPC mechanisms per interface required |
| Mostly low-frequency transfer of small data chunks | High-frequency mass data streaming |
| Scheduling policy definition left to the IPC binding implementation | Scheduling policy definition API required |

# Challenge 1: Optimized data driven scheduling



# Challenge 2: Efficient multi-binding support

- Activities / algorithms can be implemented using different technologies, e.g.
  - C++ on CPU for decision steps
  - OpenVX graph on CPU, GPU, image processing accelerator or a mix of these for massively parallel image processing
  - Custom accelerator code for highly optimized accelerator usage
- Streaming data from a single producer has to be forwarded to multiple consumers inside the same process using different transport technologies




Reduce copy operations to the technical limit (zero-copy if possible)