

# IEC 61499 Applications in Embedded Systems

Partha S Roop

Precision Timed Systems Group

<http://www.ece.auckland.ac.nz/~pretzel/>

• University of Auckland, New Zealand •

# Outline

- Specific requirements of Embedded Systems
- TimeMe – a tool-kit for safety-critical embedded systems using IEC 61499
- Case Study 1: Fertigators
- Case Study 2: Animal Weighing Scales
- Extensions to IEC 61499

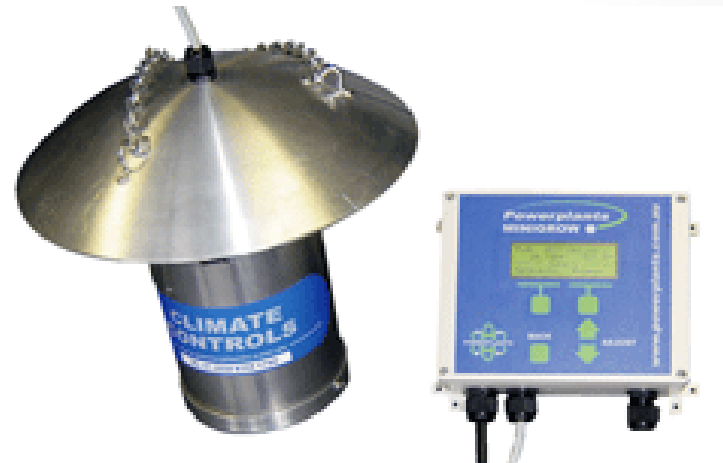
# Embedded Systems

- Unique requirements of Embedded systems
  - Smaller code footprint.
  - No OS or runtime.
  - Custom communication protocols.
- Requirement for code generation
  - C Code generation.
  - Re-use of existing code base.
- Verification and Validation
  - Simulation prior to deployment
  - Formal verification
- Hard Real-Time Systems
  - Static timing analysis.
  - Distributed deployment without any middleware.



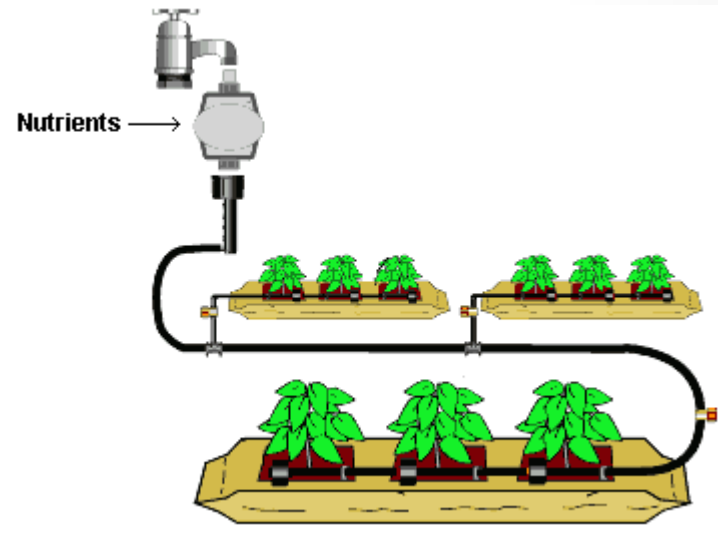
# Powerplants Ltd.

- Provide greenhouse growers throughout Australia with the latest horticultural technology and supplies.
- Embedded Systems:
  - Set point control (non-PID controlled)
  - C code



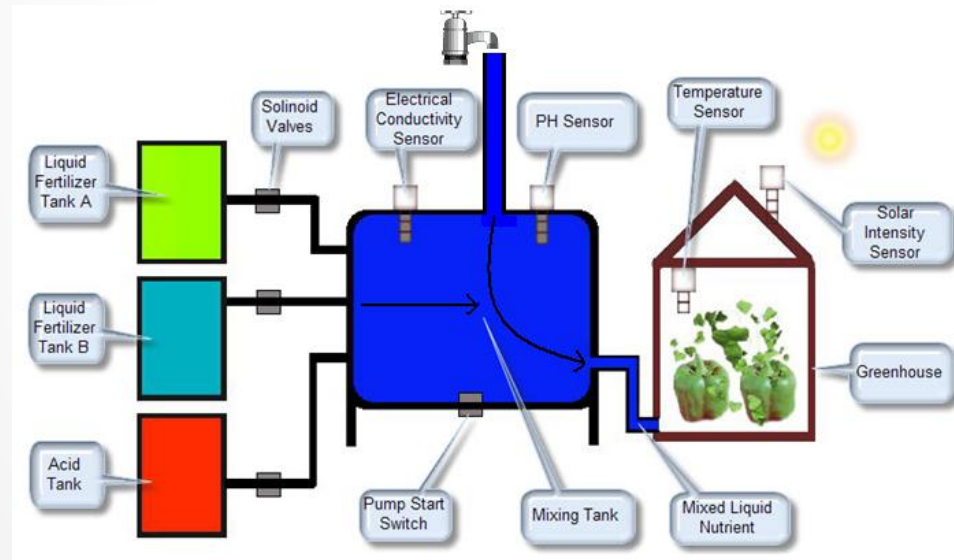
# Fertigator

- Modern greenhouses are hydroponic.
- Fertigation is the process of injecting liquid nutrients into water.
- Nutrients and water are mixed inside a tank.
- Instrumentation based control for the optimal release of nutrients.



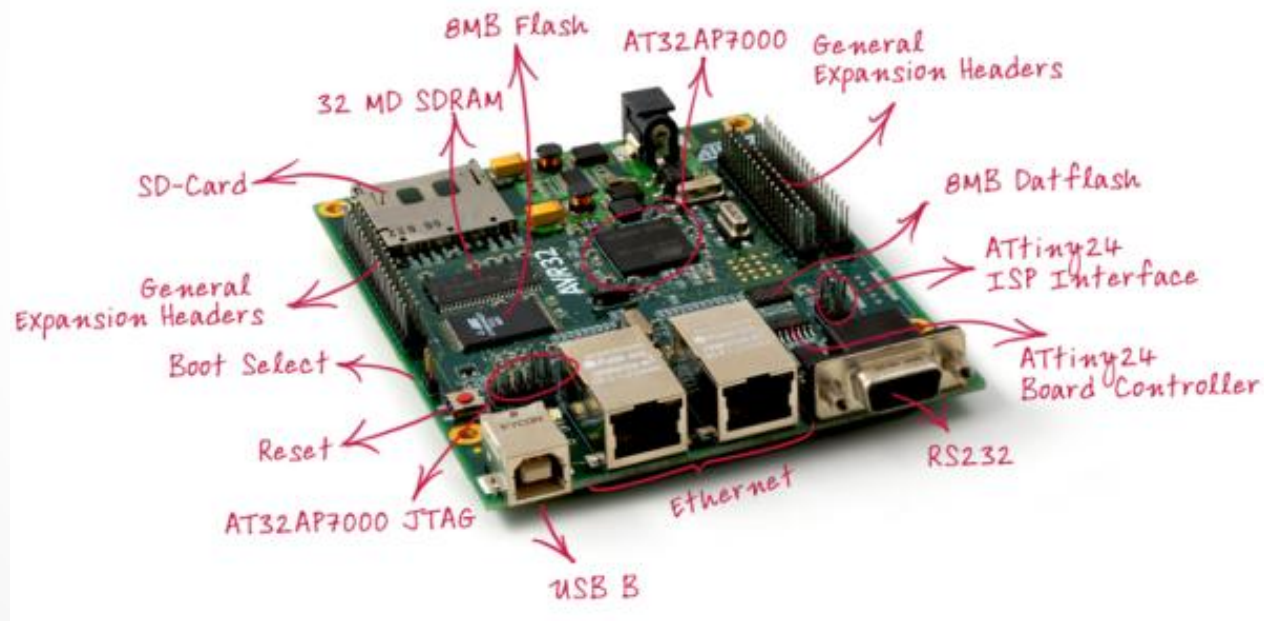
# Fertigation Control

- Measure electrical conductivity, temperature, solar intensity and PH level.
- Control pumps and valves to reach desired values.



# The Hardware

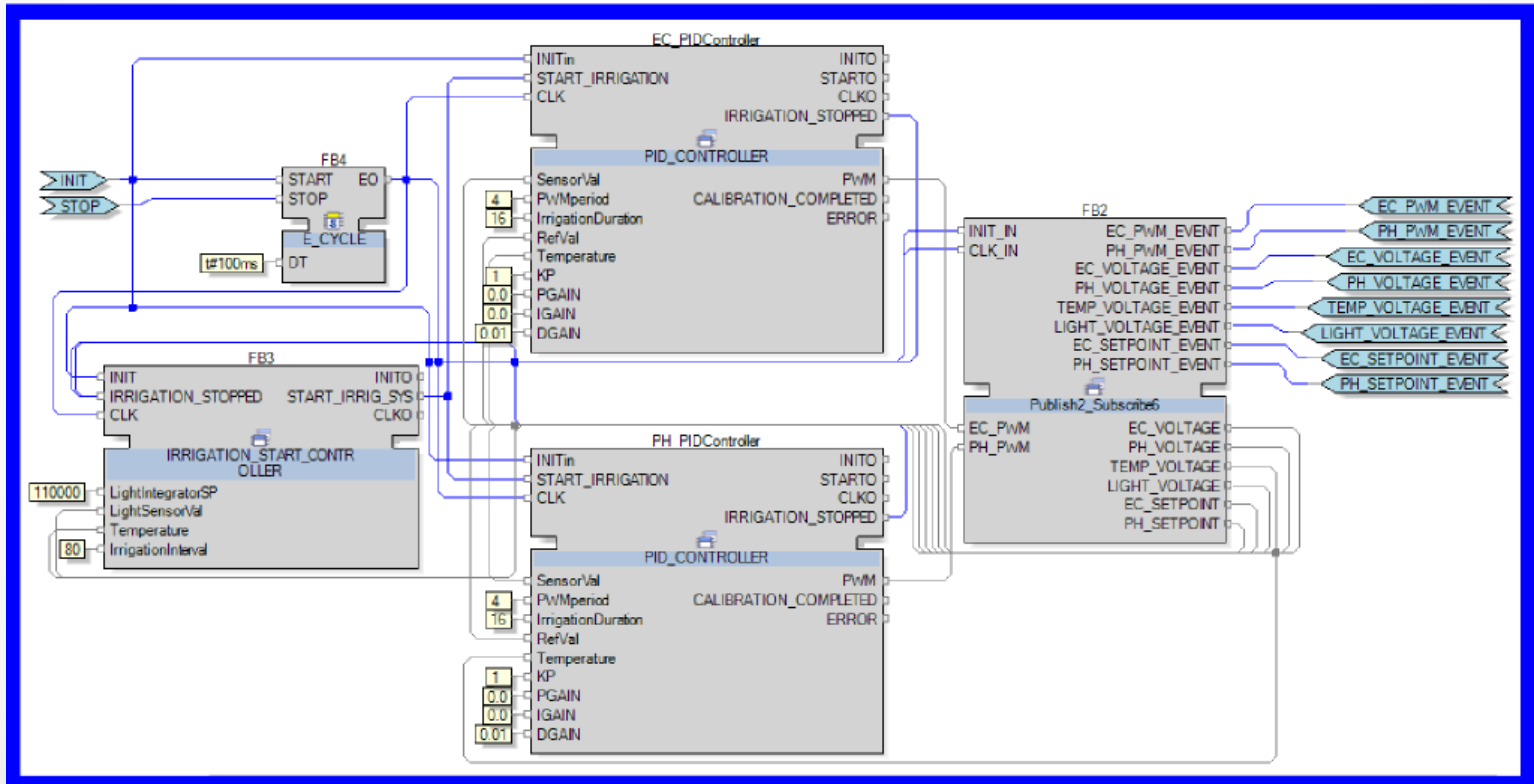
- The AVR NGW100 board, runs embedded Linux.
- On-board WAN setup for Ethernet communication.



# IEC61499 based Solution

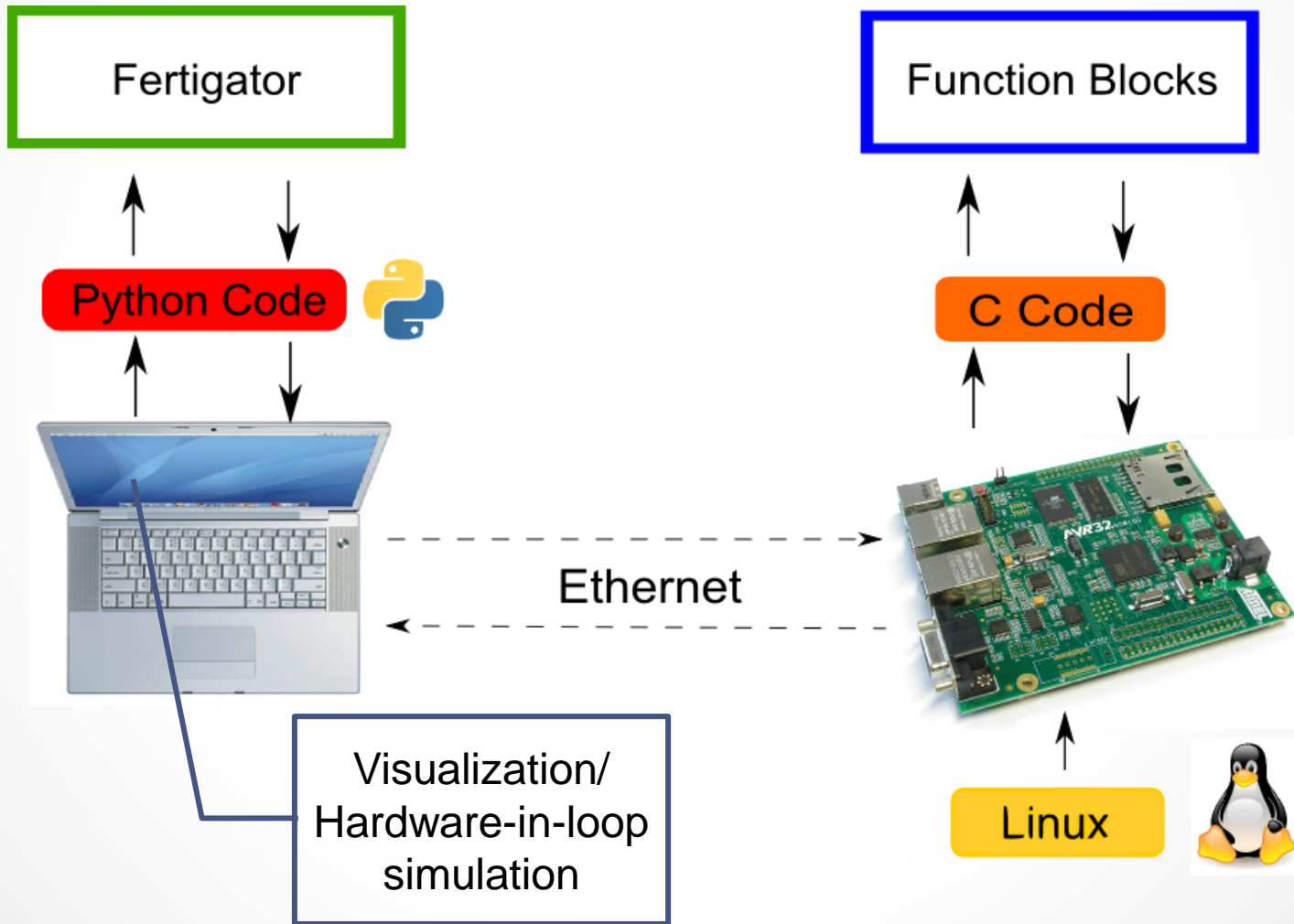
Fertigator

Function Blocks





# IEC61499 based Solution



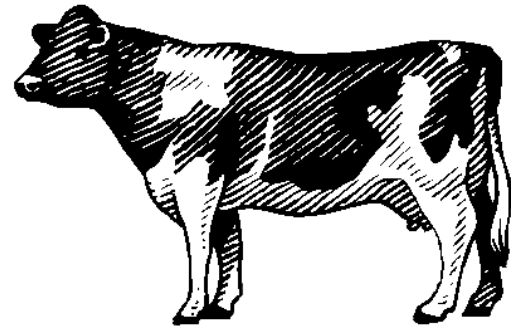
# Agri-tech example

- A large agri-tech products company such as livestock weighing systems and milk meters.
- Client interested in the application of MDE methodology.
- We are jointly exploring the feasibility of using IEC61499 in the embedded system design life-cycle.
- Embedded Systems
  - ARM processor
  - UML and C Code



# Weighing Scale

- Secure connection to Bluetooth EID readers.
- Animal weight history and gain predictions.
- Record keeping for weight , treatment and custom fields of more than 50,000 animals.
- Statistics and on-screen display.



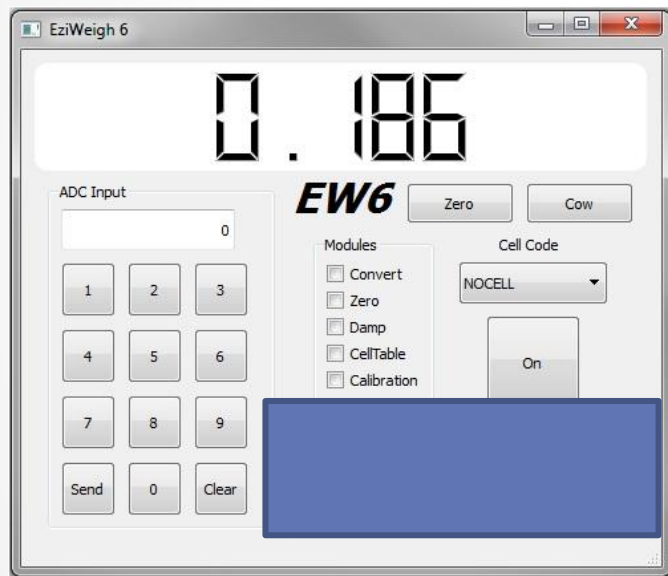
# Legacy System Issues

- Hard to debug and maintain code (200k lines of C code developer in over 20 years). With time, speed of design drops off and existing code base is hard to alter.
- Non modular design makes it hard to reuse code.
- Lack of visualisation of data flow between modules introduces high cost of code maintenance.
- Process of documentation is quite labour intensive and prone to quickly becoming out of date.
- While there is an abundance of validation/simulation tools available there is a noticeable lack of high-level validation/verification techniques such as observer-based verification and model simulation.

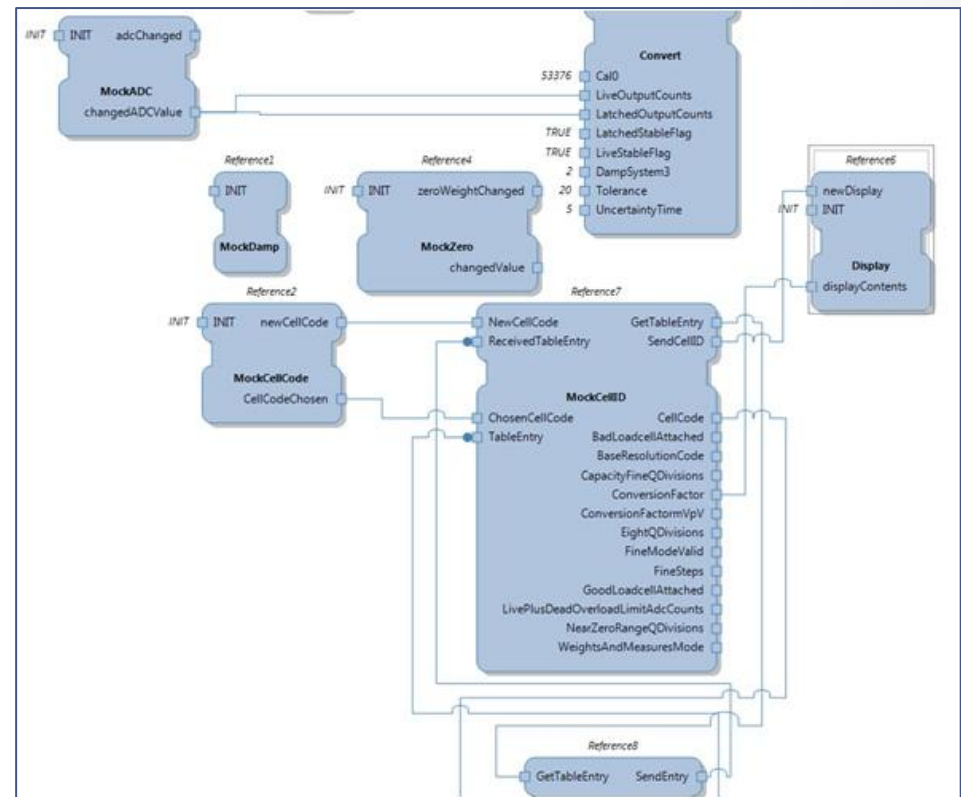


# IEC61499 based Solution

Visualization



Function Blocks



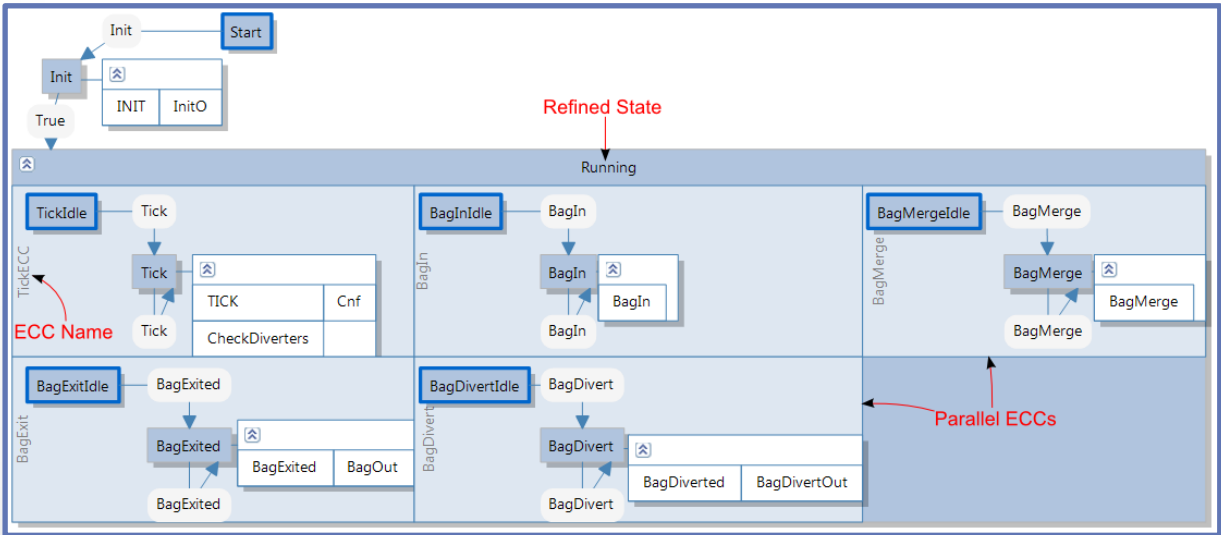
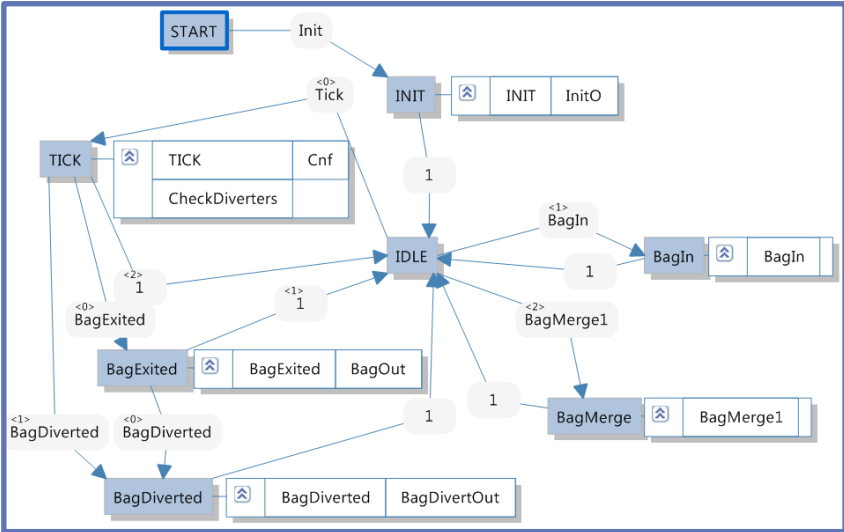
# Proposed Approach

- We are trying to reverse engineer a part of this weighing application in IEC61499.
- This subset consist of a series of signal processing modules, which have been now captured using IEC61499.
- We have used TimeMe to generate code and have performed visualization-based testing on a PC and hardware-in-the-loop simulation.
- The generated code is behaviorally equivalent.

# Extensions to IEC 61499

- Need for capturing the Hierarchy and Concurrency at function block level.
  - Hierarchical Concurrent Execution Control Charts (HCECC)
- Need to bridge the gap between industry practices (i.e. UML) and IEC 61499.
  - Mealy/Moore hybrid function blocks
- Need to interop with legacy code and existing system designs.
  - Reusing existing C code files in algorithms using header files.
- Verification and Validation
  - Observer function blocks
  - CTL verification
  - Tick-based simulation

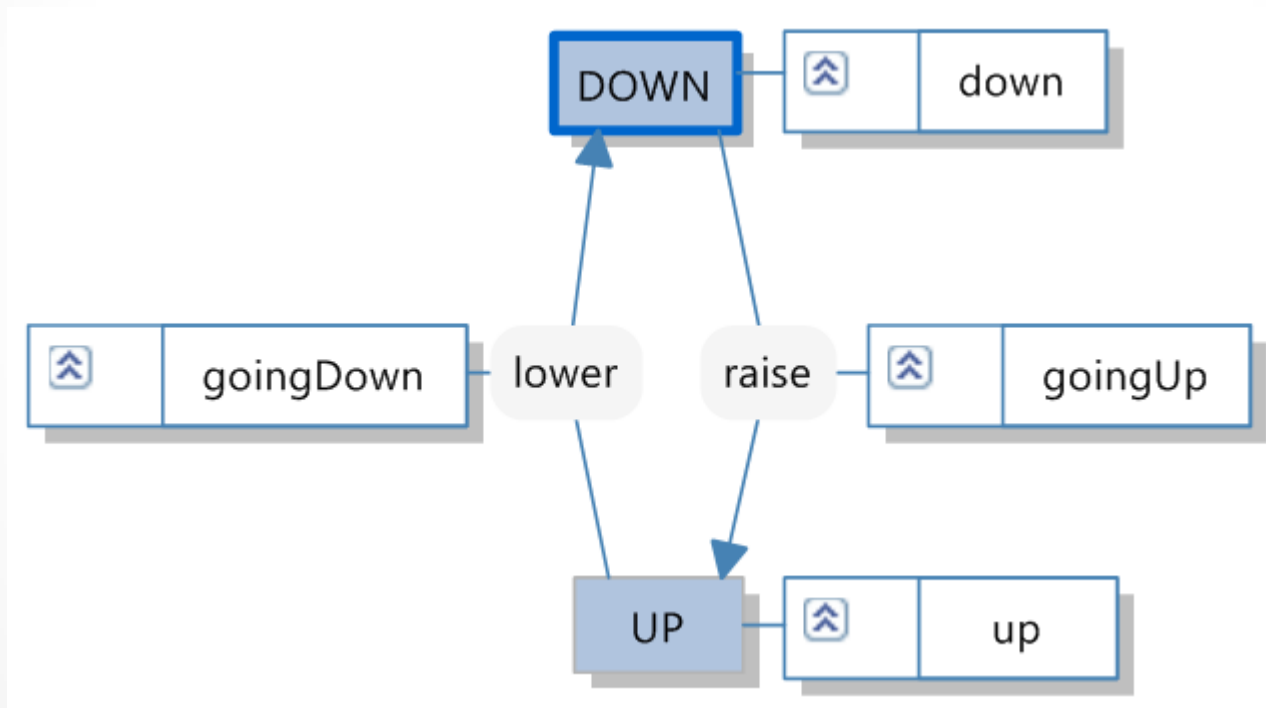
# HCECC





# Mealy/Moore Hybrid

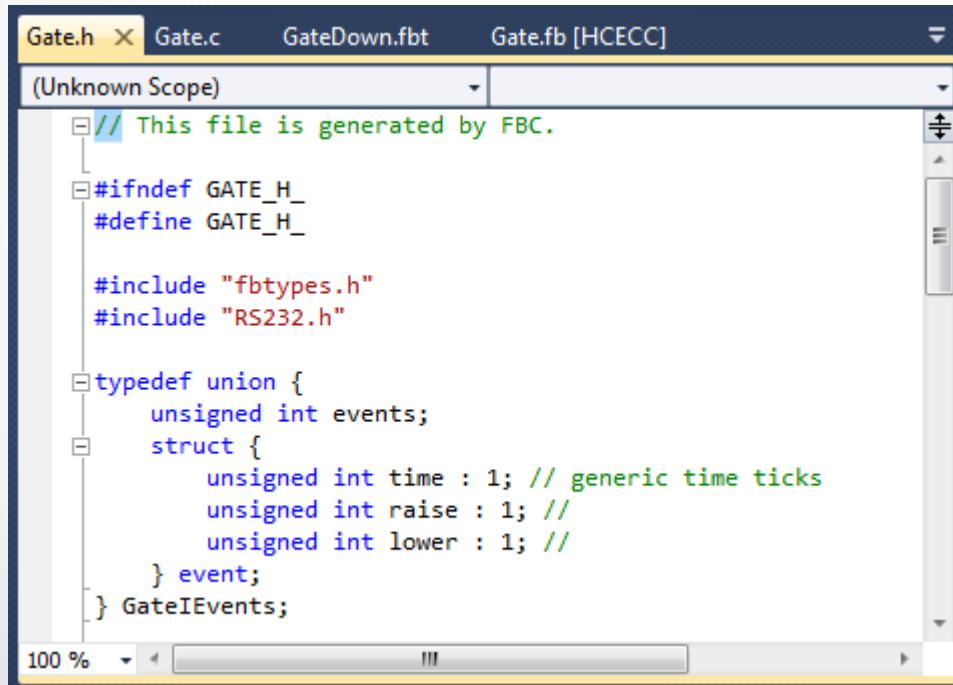
- Results in a smaller state machine.
- Makes IEC 61499 compatible with existing designs.



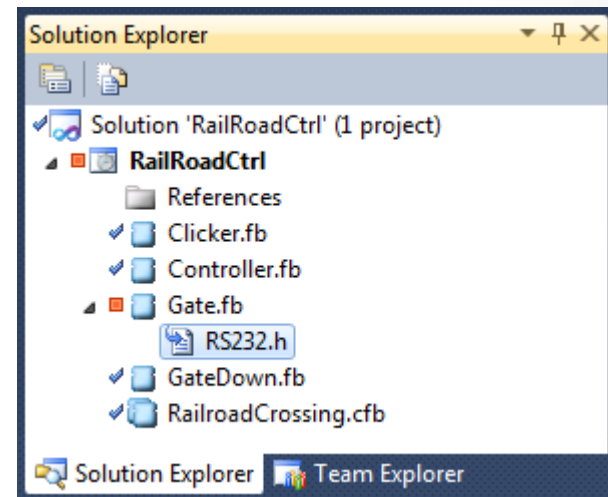
# Re-using Legacy Code

- Adding C header file to IEC 61499 XML.

```
<CompilerInfo header="#include &quot;RS232.h &quot;;"  
             classdef="">
```



```
Gate.h x Gate.c GateDown.fbt Gate.fb [HCECC]  
(Unknown Scope)  
// This file is generated by FBC.  
#ifndef GATE_H_  
#define GATE_H_  
  
#include "fbtypes.h"  
#include "RS232.h"  
  
typedef union {  
    unsigned int events;  
    struct {  
        unsigned int time : 1; // generic time ticks  
        unsigned int raise : 1; //  
        unsigned int lower : 1; //  
    } event;  
} GateIEvents;
```



# Client feedback

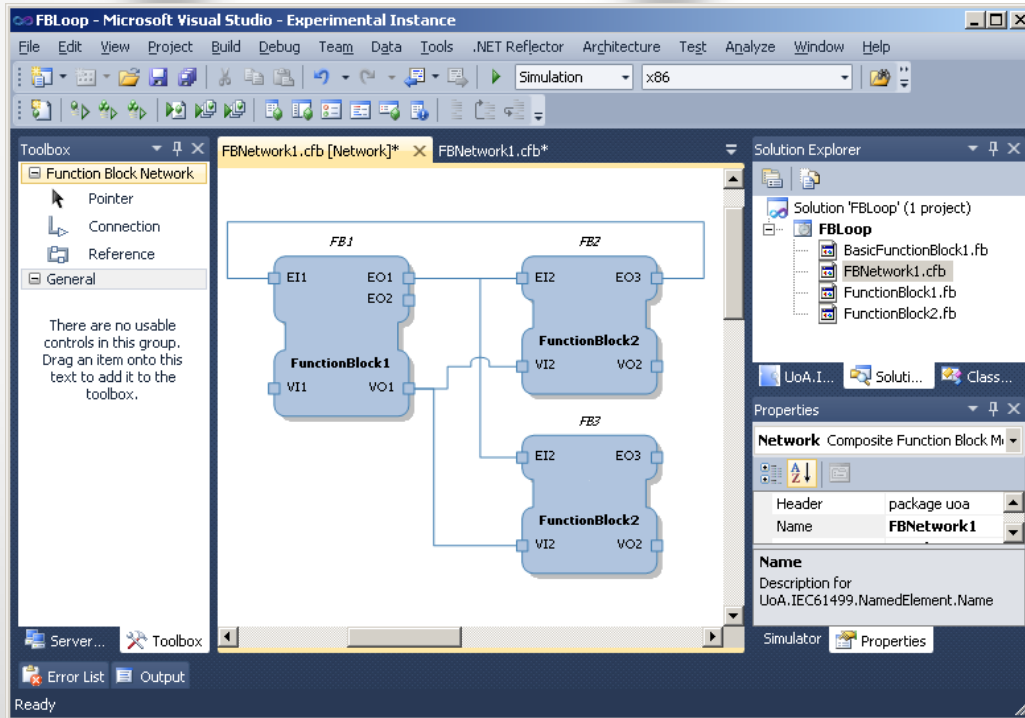
- Main advantages:
  - Explicit connection between modules / functions very useful for code maintenance.
  - HCECCs useful in their design-flow.
  - They envisage that adopting this new paradigm will slow down initial projects.
  - They also envisage that they will gain after a few projects through code reuse.
  - They also envisage reduction in maintenance cost.

# Tool Chain Characteristics

A very efficient IEC 61499 code generator

Generates deterministic and deadlock-free code

Only IEC 61499 implementation that doesn't need a run-time environment



Supports observer-based formal verification

Enables static timing analysis

# C Code Generation

The image displays a screenshot of Microsoft Visual Studio in a debugging environment, showing the generation of C code for a simulation. The interface includes a File Explorer window on the left, the Visual Studio IDE with multiple code files open, and a Properties window on the right.

**File Explorer (Left):** Shows the file structure for the project, including folders like Desktop, Downloads, Dropbox, Recent Places, Libraries, Documents, Music, Pictures, and Videos. The file `DistStnWithNetwork.c` is highlighted, showing its properties: C Source, Date modified: 4/29/2012, Size: 2.05 KB, and Date created: 4/29/2012.

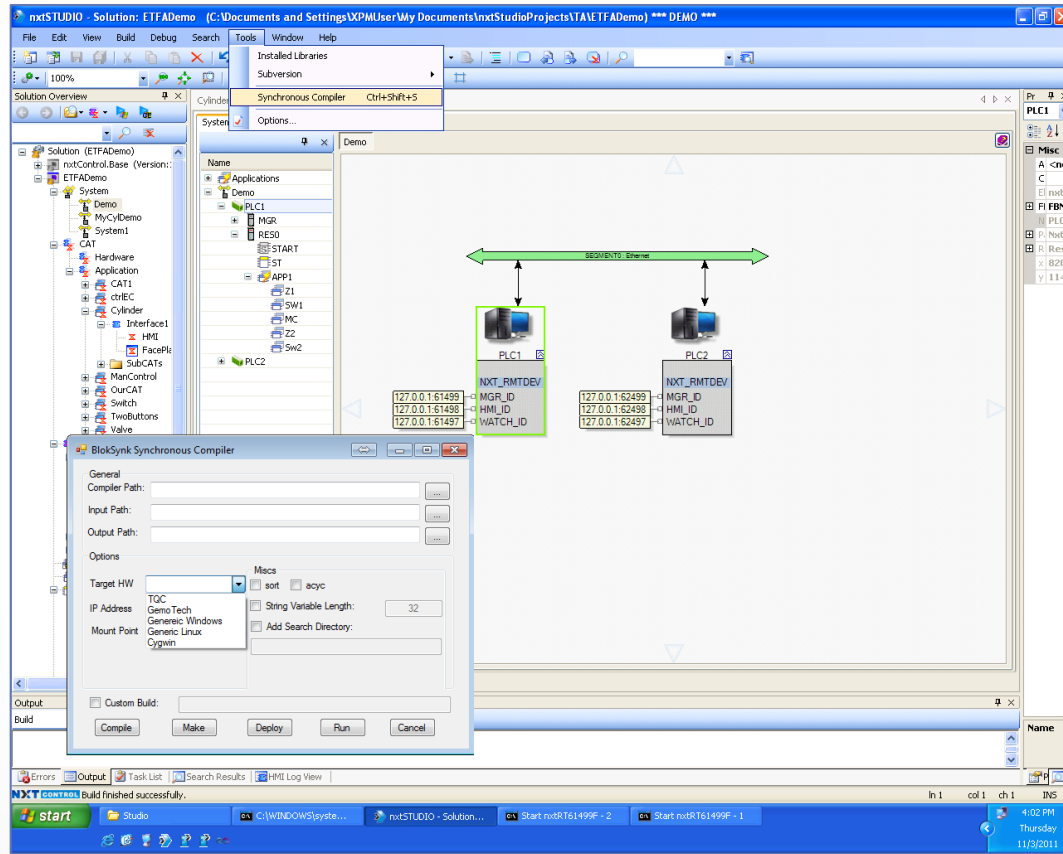
**Visual Studio IDE (Center):** The main workspace shows three code files:

- `DistStnWithNetwork.c`: Contains a function `DistStnWithNetworkinit` for initializing the network station. The code includes memory allocation, subscription initialization, and setting of various IDs and flags.
- `DistStnArm.fb [HCECC]`: A file with a C source icon.
- `DistStnPusher.c`: Contains a function `DistStnPusherGoForward` for pushing data forward, including state management and event handling.

**Properties Window (Right):** Shows the Properties window for the selected file, indicating "No Source Available" and "Solution Explorer".

**Visual Studio Interface:** The top menu bar includes File, Edit, View, Qt, Project, Build, Debug, Team, Data, Tools, Mono, Architecture, Test, Analyze, Window, and Help. The toolbar shows various development tools, and the status bar at the bottom displays "Item(s) Saved", "Ln 37", "Col 28", "Ch 28", and "INS".

# Plugin to NxtStudio



# Simulation

DistributionStation (Debugging) - Microsoft Visual Studio

File Edit View Qt Project Build Debug Team Data Tools Mono Architecture Test Analyze Window Help

Simulation availableInputs

DistributionStation.cfb [Network] DistStnPusher.fb [HCECC] UoA.IEC61499.CompositeFunc...

The diagram shows two components: **Pusher** and **DistStnPusher**. **Pusher** has provided interfaces: `InputsChange`, `PosChange`, `PusherCtrl`, and `ArmStatusChange`. **DistStnPusher** has required interfaces: `PosBack`, `PosFront`, `CylinderEmpty`, `PusherBack`, `PusherFront`, `ItemPresent`, and `ArmClear`. The diagram illustrates how the provided interfaces of **Pusher** match the required interfaces of **DistStnPusher**.

Build succeeded

DistributionStation (Debugging) - Microsoft Visual Studio

File Edit View Qt Project Build Debug Team Data Tools Mono Architecture Test Analyze Window Help

Simulation availableInputs

DistStnArm.fb [HCECC] DistributionStation.cfb [Network] DistStnPusher.fb [HCECC]

The state machine diagram shows three states: **Waiting**, **PickingUp**, and **Suck**. **Waiting** has transitions for `ItemStatus & (ItemNeedingPickup)` to **PickingUp** and `PosChange & (PosReadyToPickup)` to **Suck**. **PickingUp** has a transition for `PosChange & (PosReadyToPickup)` to **Suck**. **Suck** has a transition for `1` back to **Waiting**. Each state has associated actions: **Waiting** (DoNothing, ArmCtrl, ArmStatus), **PickingUp** (GotoPickup, ArmCtrl, ArmStatus), and **Suck** (PickupItem, ArmCtrl).

Name	Language
Tick 6	IEC61499
Tick 5	IEC61499
Tick 4	IEC61499
Tick 3	IEC61499
Tick 2	IEC61499
Tick 1	IEC61499

Name	Value	Type
InputVariables	UoA.IEC61499.Schen	UoA.IE
PusherBack	False	System
PusherFront	False	System
ArmAtPickup	False	System
ArmAtDropoff	False	System
CylinderEmpty	False	System

Build succeeded

# Formal Verification

The image displays two instances of Microsoft Visual Studio, illustrating the formal verification process for a CruiseControl system.

**Left Screenshot:** Shows the UML class diagram for the CruiseControl system. The **CruiseManager** class (under the *controller* package) has methods: `set`, `off`, `resume`, `quickAccel`, `quickDecel`, `brakePressed`, and `cclock`. It has attributes: `regulOff`, `regulSet`, `regulStdby`, `regulResume`, and `speedSet`. The **SpeedGauge** class (under the *speedo* package) has methods: `cclock`, `speed`, `time`, and `rotaryCount`. It has attributes: `speedVal` and `speed`.

**Right Screenshot:** Shows a state transition diagram for the `InStandBy` property. The diagram includes nodes for `regulStdby`, `set`, `breakPressed`, `NotInStandBy`, `InStandBy`, and `Violation`. A `True` node is also present. A dialog box titled "Success" is overlaid on the diagram, containing the text: "The verification of observer 'observer' was successful. No counter examples were found." The dialog box has an "OK" button.



# Static Timing Analysis

The screenshot displays the Microsoft Visual Studio IDE with the CruiseControl project open. The main window shows a state machine diagram with three states: NORMAL, ACCEL, and CRUISE. Transitions are labeled with events and their counts in angle brackets. For example, from NORMAL to ACCEL, the transition is triggered by <1> accelPressed. From ACCEL to CRUISE, it is triggered by <1> accelPressed. From CRUISE to NORMAL, it is triggered by <2> accelReleased. There are also self-loops on each state. The diagram includes associated data tables for each state: NORMAL has 'normal' and 'throttleChg'; ACCEL has 'cclock'; CRUISE has 'cruise' and 'throttle'. A dialog box titled 'Success' is overlaid on the diagram, containing a checkmark icon and the text: 'Timing analysis of function block was successful. The Worst Case Reaction Time value = "192"'. The dialog has an 'OK' button. The IDE interface includes a menu bar (File, Edit, View, Qt, Project, Build, Debug, Team, Data, Tools, Mono, Architecture, Test, Analyze, Window, Help), a toolbar, and a Solution Explorer on the right showing the project structure. The status bar at the bottom indicates 'Build succeeded'.

# Deployment

The screenshot shows the Microsoft Visual Studio IDE with the following elements:

- Title Bar:** WagoDistStation - Microsoft Visual Studio
- Menu Bar:** File, Edit, View, Qt, Project, Build, Debug, Team, Data, Tools, Mono, Architecture, Test, Analyze, Window, Help
- Toolbar:** Includes a 'Deploy' button and a GUID: cc1af4e1-8b54-4f2b-981e-0155
- Project Explorer:** Shows 'WagoSystemWithIO.cfb [Network]' and 'WagoSystemWithIO.cfb'
- Diagram:** A 'WagoInputBlock' component with 'Inputs' (INIT, CNF) and 'Outputs' (DIO-DI7). A 'ByteIndex' is set to 0.
- Deployment Dialog:**
  - Title:** Deployment
  - Icon:** Wago device chip
  - Section:** Wago Device Deployment
  - Text:** Please provide the wage device connection parameters for the deployment process
  - Fields:**
    - IP Address: 192.168.1.190
    - Username: root
    - Password: [masked]
  - Buttons:** Deploy, Cancel
- Status Bar:** Error List, Output, Find Symbol Results, Build succeeded