Unity: A Unified Software/Hardware Framework for Rapid Prototyping of Experimental Robot Controllers using FPGAs

Anders Blaabjerg Lange, Ulrik Pagh Schultz and Anders Stengaard Soerensen

MMMI, SDU, Denmark
OR why everyone in experimental robotics who cares about the real-world should use FPGA’s

Anders Blaabjerg Lange, Ulrik Pagh Schultz and Anders Stengaard Soerensen

MMMII, SDU, Denmark
Context: Experimental robotics @ SDU
Context: SDIR Q & A

- **Solved**: overall high-level robotic frameworks
- **Best practice**: components, middleware, MDSD
- **Robot-specific**: low-level requirements that dominate high-level design
- **Remains to be solved**: reusable and general low-level software
- **Inadequacy of state-of-the-art**: high-level languages are not low-level, low-level languages lack reusability and abstraction
- **Promising research directions**: MDSD and flexible electronics [modular, FPGA, ...]
Context: Why FPGAs?

Pros
- Portable and reusable components for hardware interface and real-time control
- Systems that are flexible and scalable
  - single-node: true parallelism
  - multi-node: real-time network (TosNet, EPL)

Cons
- Difficult to program
  - different mindset
  - numerous low-level concerns
- History
- (Cost, power)

Idea: generate low-level control of system from high-level specifications
... and automatically integrate into high-level robot frameworks
Traditional MCU based solution

Properties
- Reusable ROS PC with generic software interface based on ROS-serial
- Serial connections between PC and each board
- Custom board with software adapted to protocol and hardware

UNITY FPGA solution

Properties
- Reusable ROS PC with generated software interface based on Unity-Link
- Single serial connection to real-time network shared memory model
- Generic board with reusable interface and real-time control components

Legend:
- Completely reusable (application specific) SW, GW or HW
- Partially reusable SW, GW or HW Component
- Fully reusable, generated or generic SW, GW or HW component.
UL-spec: SDIR-VIII demo

link sdir8demo.sdir8;

public demo(ctrl, uart) {
    TOSNET(CTRL=ctrl, COMM=uart, BASE=0);
    node2acc: acc("Node 2 acc") @x=0x0081, @y=0x0082, @z=0x0083;
    node3acc: acc("Node 3 acc") @x=0x00C9, @y=0x00CA, @z=0x00CB;
    @led=0x0040: WRITE(ID="LED", CRC);
}

acc(name): @x, @y, @z {
    @x: signed READ(ID=name+: "x", PUBLISH(1,1000), CRC);
    @y: signed READ(ID=name+: "y", PUBLISH(1,1000), CRC);
    @z: signed READ(ID=name+: "z", PUBLISH(1,1000), CRC);
}
Unity Link gateware architecture
Experiment: Latency

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Read SI</th>
<th>Write</th>
<th>Write SI</th>
</tr>
</thead>
<tbody>
<tr>
<td>GW</td>
<td>1.14E-06</td>
<td>1.14E-06</td>
<td>1.26E-06</td>
<td>1.26E-06</td>
</tr>
<tr>
<td>Ideal 64Mb/s link</td>
<td>3.52E-06</td>
<td>3.52E-06</td>
<td>3.52E-06</td>
<td>3.52E-06</td>
</tr>
<tr>
<td>Python (low-level)</td>
<td>9.74E-04</td>
<td>2.24E-04</td>
<td>9.71E-04</td>
<td>2.22E-04</td>
</tr>
<tr>
<td>Unity-SW</td>
<td>1.07E-03</td>
<td>6.52E-04</td>
<td>1.06E-03</td>
<td>5.53E-04</td>
</tr>
</tbody>
</table>
Experiment: Throughput
class led_blinker(Node):
    board = XC6LX45(ft232h=ftdi_uart_if)
    led = public(outputport(7))

    def configure(self):
        self.board.ft232h = unity_serial(serial=self.board.ft232h,rate=mbaud(6))
        self.board.leds = self.led

1. Architectural level: system generation from “template”
2. Component level: configuration, composition, generation
Technology comparison

- **Application layers**
  - Single-node: Arduino
  - Distributed: Orocoss, 4DIAC

- **Execution layer**
  - Single-node: Arduino
  - Distributed: LabView, PLC

- **Control layer**
  - Single-node: Arduino
  - Distributed: LabView, PLC

- **Electrical interface**
  - Single-node: LabView, PLC
  - Distributed: EPL, CAN

- **Transducer**

- **Physical/mechanical**

- **Unity (TosNet)**
Unity Link software stack [1/2]

**Interface/Binding layer description**
Application- and language-specific bindings

**Data Proxy layer description**
Proxies for each value in the HW-proxies, and logical groupings of proxies into entities.
- Data proxy: a specific data element.
- Data proxy group: any group of singular data proxies.
- Automated setup of periodic data publishing (GW as well as SW controlled).
- Execution of user-code call-back functions, triggered by publish events.

**Hardware Proxy layer description**
SW abstractions, and mapping of data proxies into the underlying GW components.
- Provides simple component relative addressing.
- Hides the underlying protocol as well as communication interfaces.
- Automatically handles any complex component specific behavior.
- Automatically forwards any published data to the correct Data Proxies.

**Message Link layer description**
Unity Link software stack [2/2]

Message Link layer description
The Link Controller class has threads for handling response- and publish-messages from each stream associated through the Message Handler class.

The Message handler class provides a high-level, message centered and FIFO based interface to the FPGA address space, it handles transmission, re-transmission and reception of both responses and publish messages.

Interface/Binding layer description
Hardware layer implementing transmission and reception of data.

<<note>>
Replace the Message-Link and Stream-Interface layers by a platform specific interface if direct memory-mapped IO e.g. PCIExpress is desired.
UL-spec: SCARA case study

```java
public scara(ctrl, comm) {
    TOSNET(CTRL=ctrl, COMM=comm, BASE=0);
    j1: joint("joint 1") @ (0x80, 0x88, 0x8C, 0x94);
    j2: joint("joint 2") @ (0x81, 0x89, 0x8D, 0x95);
    j3: joint("joint 3") @ (0x82, 0x8A, 0x8E, 0x96);
    j4: joint("joint 4") @ (0x83, 0x8B, 0x8F, 0x97);
}
joint(name): @ (spd, cpos, pos_sp, spd_sp) {
    @ cpos: signed READ(ID=name+".cpos", PUBLISH(1,10));
    @ spd: READ(ID=name+".spd", PUBLISH(1,10));
    @ pos_sp: signed WRITE(ID=name+".pos_sp");
    @ spd_sp: WRITE(ID=name+".spd_SP");
}```