An Integrated Modeling and Testing Architecture for ROS Nodes

Jacob Hartzer

ROSCon 2023
Unmanned Systems Lab

- On/Off Road Autonomy
- LIDAR and RADAR Odometry and Perception
- Multi-Sensor Fusion and Calibration
Our Work - Calibration

- Utilize a wide variety of sensors and environments
- RTK GPS in highway applications
- Ultra-Wideband for cooperative localization and detection
- Multi-IMU, Multi-Camera fusion
The Challenge of Calibration

- Sensors do not work well when uncalibrated
- Ideally would like to calibrate sensors online
- Re-calibrating on the fly is more robust
- Proving stability and consistency is hard
- Sensor flexibility makes it even harder

Must test many, many datasets and environments
EKF-CAL Package

- EKF-CAL is flexible MSCKF-based sensor calibration package
- Inputs are YAML based and compatible with ROS 2 parameter declarations
- Inherently multi-sensor (IMU, Camera, and GPS soon)
- Developed with integrated testing and Monte Carlo simulation in mind
- Open Source!
Typical Development

Develop Algorithm in scripted language (MatLab, Julia, etc.)

Find a bug / invalid assumptions

Need performance / Real-Time

Deploy code in compiled language (C++, Rust, etc.)
Typical Development

Simulated Algorithm

Low-Fidelity Measurements

Low-Fidelity Simulation

Main Algorithm

Preprocessors

Sensor Interfaces

True Measurements

True Deployment
Integrated Modeling and Testing of ROS Nodes

Typical Development

Simulated Algorithm

Low-Fidelity Measurements

Low-Fidelity Simulation

Main Algorithm

Preprocessors

Sensor Interfaces

True Measurements

True Deployment

Need More Fidelity!
Typical Development

Integrated Modeling and Testing of ROS Nodes
Integrated Modeling and Testing of ROS Nodes

Typical Development

- Simulated Algorithm
  - Low-Fidelity Measurements
    - Low-Fidelity Simulation
  - Preprocessors
    - High-Fidelity Measurements
    - High-Fidelity Simulation
- Main Algorithm
  - Preprocessors
    - Sensor Interfaces
    - True Measurements
    - True Deployment

Obsolete!
Typical Development

- High-Fidelity Simulation
  - Simulated Algorithm
  - Preprocessors
  - High-Fidelity Measurements

- True Deployment
  - Main Algorithm
  - Preprocessors
  - Sensor Interfaces
  - True Measurements
  - True Deployment
Typical Development

- Simulated Algorithm
  - Preprocessors
    - High-Fidelity Measurements

- High-Fidelity Simulation

- Main Algorithm
  - Preprocessors
    - Sensor Interfaces
      - True Measurements
        - True Deployment
Typical Development

- Simulated Algorithm
- Preprocessors
- High-Fidelity Measurements

Functionally Identical

- Main Algorithm
- Preprocessors
- Sensor Interfaces
- True Measurements
- True Deployment

High-Fidelity Simulation
Typical Development

- Issues:
  - Very iterative development
  - Time-intensive and tedious work
  - Consistent work to maintain simulation

Functionally Identical
Typical Development

<table>
<thead>
<tr>
<th>Simulated Algorithm</th>
<th>Preprocessors</th>
<th>High-Fidelity Measurements</th>
</tr>
</thead>
</table>

High-Fidelity Simulation

<table>
<thead>
<tr>
<th>Main Algorithm</th>
<th>Preprocessors</th>
<th>Sensor Interfaces</th>
<th>True Measurements</th>
<th>True Deployment</th>
</tr>
</thead>
</table>

Too Slow!
Typical Development

- **Low-Fidelity Simulation**
  - Simulated Algorithm
  - Low-Fidelity Measurements

- **High-Fidelity Simulation**
  - Simulated Algorithm
  - Preprocessors
  - High-Fidelity Measurements

- **True Deployment**
  - Main Algorithm
  - Preprocessors
  - Sensor Interfaces
  - True Measurements
  - True Deployment
Result:
- Complex simulations lead to fragmented code
- Fragmented code is expensive to maintain
- Multiple simulations leads to:
  - Uncaught bugs
  - Untested deployment code
Typical Development

- Low-Fidelity Simulation
  - Simulated Algorithm
    - Low-Fidelity Measurements

- High-Fidelity Simulation
  - Simulated Algorithm
    - Preprocessors
      - High-Fidelity Measurements

- True Deployment
  - Main Algorithm
    - Preprocessors
      - Sensor Interfaces
      - True Measurements
      - True Deployment
Proposed Solution

Low-Fidelity Simulation

High-Fidelity Simulation

Algorithm

Preprocessors

High-Fidelity Measurements

Low-Fidelity Measurements

Sensor Interfaces

True Measurements

True Deployment
Proposed Solution

Abstraction Layers!

- Measurement Processor Abstraction
  - Low-Fidelity Measurements
  - Preprocessors

- Interface Abstraction
  - High-Fidelity Measurements
  - Sensor Interfaces

- Hardware Abstraction
  - True Measurements
  - True Deployment

Integrated Modeling and Testing of ROS Nodes
Cost of Integrated Coding

- Higher initial cost
- Pays off in long-run
Abstraction - Example

- Sensors call updates to filter / algorithm
  - Utilize real or simulated sensor messages
- Feature tracker utilizes camera measurements
  - True deployment utilizes true camera measurements
  - High-Fidelity simulation provides ray-traced images
  - Low-Fidelity simulation provides “pre-tracked features”
Abstraction - Example

TruthEngine.cpp
- GetState(double time);

SimCamera.cpp
- GenerateImages()

Camera.cpp
- ReadImages()

SimFeatureTracker.cpp
- GenerateTracks()

FeatureTracker.cpp
- TrackFeatures()

ekf.cpp
- PredictFilter()
- UpdateMSCKF()
Abstraction - Example

TruthEngine.cpp
GetState(double time);

SimCamera.cpp
GenerateImages();

SimFeatureTracker.cpp
GenerateTracks();

FeatureTracker.cpp
TrackFeatures();

ekf.cpp
PredictFilter();
UpdateMSCKF();

Camera.cpp
ReadImages();

True Hardware
Abstraction - Example

**TruthEngine.cpp**

GetState(double time);

**SimCamera.cpp**

GenerateImages();

*Hardware Abstraction*

**SimFeatureTracker.cpp**

GenerateTracks();

**FeatureTracker.cpp**

TrackFeatures();

**ekf.cpp**

PredictFilter();

UpdateMSCKF();

**Camera.cpp**

ReadImages();

Integrated Modeling and Testing of ROS Nodes
Abstraction - Example

TruthEngine.cpp
GetState(double time);

SimCamera.cpp
GenerateImages();

SimFeatureTracker.cpp
GenerateTracks();
Processor Abstraction
TrackFeatures();

Camera.cpp
ReadImages();

FeatureTracker.cpp

ekf.cpp
PredictFilter();
UpdateMSCKF();
\[
\begin{bmatrix}
\tilde{a}_x \\
\tilde{a}_y \\
\tilde{a}_z
\end{bmatrix} =
\begin{bmatrix}
S_{ax} & 0 & 0 \\
0 & S_{ay} & 0 \\
0 & 0 & S_{az}
\end{bmatrix}
\begin{bmatrix}
1 & \alpha_{a1} & \alpha_{a2} \\
\alpha_{a3} & 1 & \alpha_{a4} \\
\alpha_{a5} & \alpha_{a6} & 1
\end{bmatrix}
\begin{bmatrix}
a_x \\
a_y \\
a_z
\end{bmatrix}
+ \begin{bmatrix}
b_{ax} \\
b_{ay} \\
b_{az}
\end{bmatrix}
+ \begin{bmatrix}
n_{ax} \\
n_{ay} \\
n_{az}
\end{bmatrix}
\]
Abstraction Models: Camera

\[
x_n = \frac{C \mathbf{p}_x f_x + c_x}{w C \mathbf{p}_z} - 1
\]

\[
y_n = \frac{C \mathbf{p}_y f_y + c_y}{h C \mathbf{p}_z} - 1
\]

\[
r^2 = x_n^2 + y_n^2
\]

\[
\begin{bmatrix}
x \\ y \\ y_n
\end{bmatrix} = \begin{bmatrix}
x_n \\ y_n
\end{bmatrix} \cdot \begin{bmatrix}
1 + k_1 r^2 + k_2 r^4 + k_3 r^6 \\
1 + k_1 r^2 + k_2 r^4 + k_3 r^6
\end{bmatrix} + \begin{bmatrix}
2p_1 x_n y_n + p_2 \cdot (r^2 + 2x_n^2) \\
p_1 \cdot (r^2 + 2y_n^2) + 2p_2 x_n y_n
\end{bmatrix}
\]

Radial

Tangential

Camera 3 Triangulation Errors

Time [s]
Abstraction - Benefits

- Improves accuracy of simulation
- Catches more bugs earlier
- Reduces rework (no code divergence)
- More beneficial unit testing
- Robust Monte Carlo testing
Unit Testing

- This architecture allows test-driven development
- Any tweaks or examples can become tests
- Tests ensure code accuracy and functionality
- Can be automated per commit / merge request

It's Easy!

GoogleTest + CMake + Colcon + lcov
Unit Testing Nodes

- Ideally, we should test as much as possible
- To unit test ROS nodes, we split the node into an entrypoint and node class

```
main.cpp
int main();
Node::Node();
```

```
entrypoint.cpp
int main();
```

```
node.cpp
Initialize();
DeclareParameters();
LoadParameters();
```
## Unit Testing Nodes

### `node.cpp`
- Initialize()
- DeclareParameters()
- LoadParameters()

### `entrypoint.cpp`
- `int main();`

### `test.cpp`
```cpp
TEST_F(ExampleNode, ExampleNode_test)
{
    ExampleNode node;

    node.Initialize();
    node.DeclareParameters();

    node.set_parameter(rclcpp::Parameter("param1"));
    node.set_parameter(rclcpp::Parameter("param2"));

    node.LoadParameters();
}
```
Unit Testing Nodes

**entrypoint.cpp**

```cpp
int main();
```

**node.cpp**

```cpp
Initialize();
DeclareParameters();
LoadParameters();
```

**test.cpp**

```cpp
TEST_F(ExampleNode, ExampleNode_test)
{
  ExampleNode node;

  node.Initialize();
  node.DeclareParameters();

  node.set_parameter(rclcpp::Parameter("param1"));
  node.set_parameter(rclcpp::Parameter("param2"));

  node.LoadParameters();
}
```
## Unit Testing Nodes

### entrypoint.cpp

```cpp
int main();
```

### node.cpp

```cpp
Initialize();
DeclareParameters();
LoadParameters();
```

### test.cpp

```cpp
TEST_F(ExampleNode, ExampleNode_test)
{
    ExampleNode node;

    node.Initialize();
    node.DeclareParameters();

    node.set_parameter(rclcpp::Parameter("param1");
    node.set_parameter(rclcpp::Parameter("param2");

    node.LoadParameters();
}
```
### Unit Testing Nodes

<table>
<thead>
<tr>
<th>entrypoint.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize();</td>
</tr>
<tr>
<td>DeclareParameters();</td>
</tr>
<tr>
<td>LoadParameters();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_F(ExampleNode, ExampleNode_test)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>ExampleNode node;</td>
</tr>
<tr>
<td>node.Initialize();</td>
</tr>
<tr>
<td>node.DeclareParameters();</td>
</tr>
<tr>
<td>node.set_parameter(rclcpp::Parameter(&quot;param1&quot;));</td>
</tr>
<tr>
<td>node.set_parameter(rclcpp::Parameter(&quot;param2&quot;));</td>
</tr>
<tr>
<td>node.LoadParameters();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
## Unit Testing Nodes

### entrypoint.cpp

```cpp
int main();
```

### node.cpp

```cpp
Initialize();
DeclareParameters();
LoadParameters();
```

### test.cpp

```cpp
TEST_F(ExampleNode, ExampleNode_test)
{
  ExampleNode node;

  node.Initialize();
  node.DeclareParameters();

  node.set_parameter(rclcpp::Parameter("param1"));
  node.set_parameter(rclcpp::Parameter("param2"));

  node.LoadParameters();
}
```
## Unit Testing Nodes

<table>
<thead>
<tr>
<th><strong>entrypoint.cpp</strong></th>
<th><strong>test.cpp</strong></th>
</tr>
</thead>
</table>
| int main();       | TEST_F(ExampleNode, ExampleNode_test) {
|                   |   ExampleNode node;
|                   |   node.Initialize();
|                   |   node.DeclareParameters();
|                   |   node.set_parameter(rclcpp::Parameter("param1"));
|                   |   node.set_parameter(rclcpp::Parameter("param2"));
|                   |   node.LoadParameters();
|                   | }

<table>
<thead>
<tr>
<th><strong>node.cpp</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize();</td>
</tr>
<tr>
<td>DeclareParameters();</td>
</tr>
<tr>
<td>LoadParameters();</td>
</tr>
</tbody>
</table>
Monte Carlo Testing

- With fast enough simulations, we can run thousands of example datasets
- Random initialization and measurement errors are inserted
- Utilizing abstractions increases confidence of filter stability
Key Takeaways

- Stop developing simulations separate from deployments
- Utilize existing models for integrated simulations
- Abstract layers as low as possible
- Utilize multiple layers of abstraction for various fidelity / execution speed
- Try out EKF-CAL! We love feedback and collaboration!
Presentation References


