SLX FOR ROS

MASTERING SOFTWARE/HARDWARE COMPLEXITY AT THE INTELLIGENT EDGE

Maximilian Odendahl, Benjamin Goldschmidt
odendahl@silexica.com
Autoware Workshop, Paris, 9.6.2019
Silexica

Est. 2014 after a decade of research

Team of world leading software and hardware experts

60 people worldwide, engineering HQ in Germany

3 offices and worldwide local support engineers

© SILEXICA | All rights reserved | Confidential | 6/12/2019 | 2
Understanding both worlds
Unique combination of Source Code & Execution Analysis

- Static Source Code Analysis
  - Detect simple bugs, e.g. division by zero
  - Detect undefined behavior
  - Check for coding standards, e.g. MISRA
  - Target independent analysis

- Dynamic Run-Time Behavior Analysis
  - Profiling based on specific input stimuli
  - Target environment dependent analysis
  - No traceability back to original source code

Silexica uniquely combines both using semantic analysis to derive a complete understanding encompassing both the logical software architecture and its dynamic execution behavior.
SLX Overview

Understanding of Concurrent Software

Sequential / Parallel Code (C/C++)
Linux / Adaptive AUTOSAR / ROS2

SLX for C/C++
SLX for FPGA
High Performance Cluster
FPGA

RT Safety Code

SLX for FPGA
SLX for Scheduling

Safety MPSoC

Optimized Execution on FPGAs

Scheduling Optimization and Memory Mapping
Understanding Concurrent Software

SLX to give deep insights of multi-threaded and multi-application scenarios with understanding of concurrency, communication and synchronization:

- Thread generation and synchronization order
- Communication: global variables, shared memory, sockets, file I/O, etc
- Protection Analysis across applications
- Sequence diagrams for ordering of events and lifetime of objects

SLX provides a continuous up-to-date sw architecture and execution overview from the source code to eliminate the gap from design to implementation
Select function highlighting according to different metrics
Example: Sub-object Insights

Sub-Object Analysis: Arrays and Structs

Accesses to an array variables from the same source line are shown with accessed ranges.

Example: Protection Analysis

Identify missing inter-thread and inter-process protection with full traceability to the problematic source lines

**Industry-First:** Detection of potential data races and dead-locks across separate processes

<table>
<thead>
<tr>
<th>Protection</th>
<th>Column Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected</td>
<td>Unprotected</td>
<td>All accesses happen without a mutex being locked.</td>
</tr>
<tr>
<td>Sometimes</td>
<td>Sometimes</td>
<td>Some accesses happen with a mutex being locked, others without.</td>
</tr>
<tr>
<td>Protected</td>
<td>Mutex Source Location</td>
<td>All accesses happen with a mutex being locked, the declaration location of the mutex is shown.</td>
</tr>
</tbody>
</table>

---

© SILEXICA | All rights reserved | Confidential | 6/12/2019 | 8
Understanding of Concurrent Software

Optimized Execution on FPGAs

Sequential / Parallel Code (C/C++)
Linux / Adaptive AUTOSAR / ROS2

SLX for C/C++
SLX for FPGA
SLX for Scheduling

High Performance Cluster
FPGA
Safety MPSoC

Scheduling Optimization and Memory Mapping

© SILEXICA | All rights reserved | Confidential | 6/12/2019 | 9
Analysis of ROS Applications with SLX
Motivation
Why specific tooling for ROS?

- Lack of systems engineering tooling for ROS
- ROS has well-defined semantics (communication, etc.)
  - Allows better analysis and visualization
- Well-defined set of target platforms (and operating systems)
  - Allows to integrate additional platform-specific tracing sources (e.g. kernel tracing)
- Known set of typical dependencies
  - Integration hurdles can be reduced by providing out of the box support
Motivation
How to harden/optimize ROS systems for real-time usage?

- **Software architecture:**
  - Redundancy
  - Separate validity checking modules (e.g. based on the physical properties of the underlying system)
  - Add fallback/safe states for modules
  - Define timeouts for all function calls with non-deterministic runtime (e.g. I/O (HDD read/write, network traffic, sensors, etc.), blocking synchronization, etc.)
  - Allow dynamic memory allocations only in the startup phase (e.g. [https://github.com/osrf/osrf_testing_tools_cpp](https://github.com/osrf/osrf_testing_tools_cpp))
  - ...
  - ...
Motivation
How to harden/optimize ROS systems for real-time usage?

- **OS:**
  - Install Linux RT patches (or use a RT OS like QNX)
  - Isolate processors from the OS (e.g. Linux `isolcpus` boot parameter, Hypervisors, etc.)
  - Pin processes to processors
  - ...

- **Processes & standards:**
  - Follow coding guide lines (e.g. MISRA C/C++)
  - Follow development processes for safety-critical systems (e.g. IEC 61508)
  - ...
System analysis, testing, integration:

- Systematically evaluate components in the presence of system stress
- Determine component variability (e.g. through systematic, iterative measurements)
- Chaos engineering (e.g. randomly stop processes, closes sockets, etc.)
- Static and dynamic analysis to derive guarantees (where possible) and enhance analysis

SLX for ROS
SLX FOR ROS
Concept
Multi-level tracing/profiling

- **System-level:**
  - Acquire kernel events (context switches, process starts, I/O, etc.) to establish a global timeline of the system

- **Framework-level:**
  - Acquire framework-specific data (e.g. from the ROS communication stack) to provide semantic context → fuse with system timeline

- **Module-level:**
  - Acquire module-specific data (dynamic information (call stacks, node names, etc.) + static information) to allow an in-depth module analysis → correlate data with global system timeline

- **Build data:**
  - Extract relevant binaries from the build system to filter/enhance acquired tracing data
Concept
Multi-level tracing/profiling

RAW data (tracing context switches): Process names
Filter and highlight relevant processes (based on execution/build analysis).

**Concept**

Multi-level tracing/profiling

![Diagram](image)

- **CPU0**
  - LIDAR
  - OS
  - OS
  - SensorFusion
  - Misc.
  - Publish (topic: pclData)
  - Callback (topic: pclData)
  - Publish (topic: pclData)

- **CPU1**
  - IMU
  - Kernel
  - LIDAR
  - Misc.
  - LIDAR
  - IMU
  - Publish (topic: pclData)
  - Publish (topic: pclData)
  - Callback (topic: pclData)
  - Callback (topic: pcmData)

- **CPU2**
  - Kernel
  - SensorFusion
  - IMU
  - Kernel
  - SensorFusion
  - IMU
  - Publish (topic: pcmData)
  - Publish (topic: pcmData)
  - Callback (topic: pcmData)
  - Callback (topic: pcmData)

- **CPU3**
  - Kernel
  - SensorFusion
  - OS
  - SensorFusion
  - OS
  - Publish (topic: pcmData)
  - Callback (topic: pcmData)
  - Callback (topic: pcmData)

Add ROS events to timeline (with direct backlink to source code → e.g. right-click)
Concept
Multi-level tracing/profiling

Example: Execution of Autoware.auto unit tests (colcon test) (unfiltered Tracecompass output)
Concept
Multi-level tracing/profiling

Example: Execution of Autoware.auto unit tests (colcon test) (filtered (preliminary) SLX output)
**Concept**

System analysis and testing

- Static analysis combined with dynamic data-binary analysis:
  - Detect system calls (read/write, send/recv, etc.) with difficult to predict runtime behavior
  - Detect memory allocations after startup phase
  - Detect non-ROS communication (e.g. shared memory, sockets, etc.)
  - Instrument selectively (e.g. data transfers, execution path)
Concept

System analysis and testing

- Multi-run analysis:
  - Automatically compare different runs and highlight runtime differences (on system-, framework-, and module-level):
    - Compare runs with same/different inputs
    - Compare different code revisions
  - Configurable max. variance limits (e.g. for automated integration tests)
Example: Autoware NDT matching module (v1.8; used for positioning)

Problem: Positioning of car is unstable (in ~60% of tests a positioning lock could not be acquired within 60 s of the start)

Inputs and input timing should always be the same (given by ROSbag)
Position NOT “locked” in:

- Positioning module (marked red) does not run in regular time intervals and has a highly variable runtime

Position “locked” in:

- Positioning module (marked red) runs in regular intervals with acceptable runtime variation
Dynamically inject artificial CPU, memory, etc. intensive loads during test scenarios:
  - Highly configurable (number of threads, number of reads/writes, allocation size, etc.)
  - Loadable/storable load injection profiles (load over time)
  - Record load profiles on existing systems
  - Automated load analysis:
    - Generation of Pareto curves for different load parameters
Prototype: Autoware Test Scenario

- **Test scenario:**
  - Autoware v1.8 (patched to increase stability)
  - Based on “Moriyama” dataset
  - Isolated GPS scenario (runtime: ~10 seconds)
  - Rate of GPS sensor (input data): 25 Hz

- The test scenario runs in a custom Nvidia Docker container (which includes the full Autoware stack and SLX tools)

- **Test system:**
  - Intel i7-6700HQ (4(8)x2.6 GHz)
  - Nvidia Quadro M1000M
  - 40 GB DDR-RAM

Preliminary results!
Prototype: Autoware Test Scenario

Integration

- Run test scenario directly from SLX
  - Creates system overview analysis
Prototype: Autoware Test Scenario

System-level analysis
Prototype: Autoware Test Scenario
System-level analysis

Active modules (incl. topic dependencies) annotated with runtime statistics, context switches, etc.
Prototype: Autoware Test Scenario

System-level analysis

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Runtime</th>
<th>... (mean)</th>
<th>... (median)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector_map_loader</td>
<td>:0</td>
<td>146.6 ms (0.18%)</td>
<td>20.33 us (0.0%)</td>
<td>15.23 us (0.0%)</td>
</tr>
<tr>
<td>nmea2tfpose</td>
<td>:0</td>
<td>69.68 ms (0.09%)</td>
<td>43.88 us (0.0%)</td>
<td>36.54 us (0.0%)</td>
</tr>
<tr>
<td>robot_state_publisher</td>
<td>:0</td>
<td>184.44 ms (0.23%)</td>
<td>17.69 us (0.0%)</td>
<td>14.62 us (0.0%)</td>
</tr>
<tr>
<td>world_to_map</td>
<td>:0</td>
<td>176.59 ms (0.22%)</td>
<td>16.32 us (0.0%)</td>
<td>12.39 us (0.0%)</td>
</tr>
<tr>
<td>play_155111620298243</td>
<td>:0</td>
<td>210.44 ms (0.26%)</td>
<td>103.87 us (0.0%)</td>
<td>36.25 us (0.0%)</td>
</tr>
<tr>
<td>idle</td>
<td>:0</td>
<td><strong>64.63 s (80.12%)</strong></td>
<td><strong>801.23 us (0.0%)</strong></td>
<td><strong>328.65 us (0.0%)</strong></td>
</tr>
<tr>
<td>kernel</td>
<td>:0</td>
<td>177.07 ms (0.22%)</td>
<td>45.78 us (0.0%)</td>
<td>19.73 us (0.0%)</td>
</tr>
<tr>
<td>os</td>
<td>:0</td>
<td>423.04 ms (5.24%)</td>
<td>453.23 us (0.0%)</td>
<td>37.66 us (0.0%)</td>
</tr>
</tbody>
</table>

Filterable, aggregated module statistics, including kernel, OS, and idle statistics
Prototype: Autoware Test Scenario
Module-level analysis

Complete overview over all active functions of a module (call graph)
Summary

- Multi-level (from system- down to function-level) code insights to unveil and monitor the inner mechanics of complex ROS systems
- SW stability analysis to evaluate robustness
- SW variance analysis to detect unexpected behavior
- Full tracability of results back to the source code
- Easy CI integration for systematic SW integration testing
Outlook

- Use current prototype for Autoware.auto 3D object detection stack

- Community Input:
  - Feature priority?
  - Community vs. commercial version (e.g. feature-based, OS-based)?
  - User-cases, e.g. cloud-based CI automation vs daily dev flow?