# Software libraries for increasing the reliability of C++ software

Geoffrey Biggs
Open Source Robotics Corporation

# Outline

- What is ROS?
- The ROS 2 Safety Working Group
- Contract enforcement in C++
- Software watchdogs

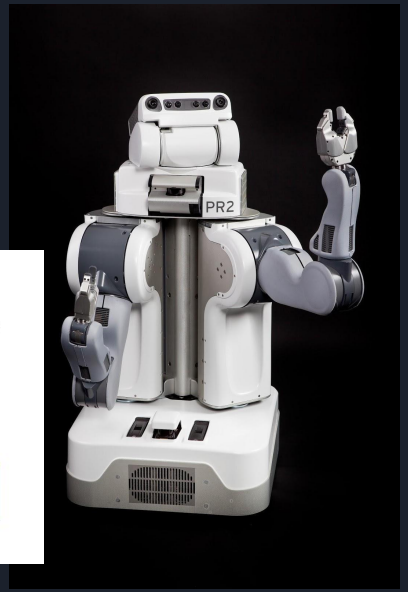
# What is ROS?

- A software platform for distributed, embedded software systems
  - Most notably, robots
- Massively popular in the robotics field
  - Used on a huge variety of robots
  - Popular in research and industry
  - Used on products you can purchase today

ROS

# Where did ROS come from?

- Originally developed by Willow Garage, a robotics venture
- Designed to make software development for robotics easier
  - Particular focus on Willow Garage's robot, the PR2: a single robot with massive (for the day) computing power designed for research
- Open source from the start
- Now maintained by the non-profit Open Source Robotics Foundation



# ROS 2

- Second version of ROS
- Designed and implemented from scratch to fix the shortcomings of ROS 1
- ROS 1 is a massively-successful set of software, but...
  - Not great for small-scale embedded systems
  - Too much home-grown software when good solutions exist already
  - Organically-grown code base
  - Inflexible architecture
- ROS 2 leverages existing software (e.g. DDS for communications) and adds what is unique about ROS on top

# ROS 2 Safety Working Group

- ROS 2 community development is organised by
  - A Technical Steering Committee with restricted membership
  - Open-to-all working groups for specific topics
- The Safety WG focuses on anything that may be useful when building (with ROS) robots that may have safety concerns
- Twice-monthly meetings and ongoing software development activities

# ROS 2 Safety Working Group

- Current projects include:
  - An open source tool chain for specifying and managing requirements in open source software, based on Doorstop and Git
  - A library for enforcing contracts in C++
  - A set of ROS nodes (distributed entities) for software watchdogs in ROS-based software systems
  - Best practice documentation for using ROS safely

# Software contracts

- Contracts are a mechanism for defining what a software function wants and what it provides/does not provide
- Originally developed for the Eiffel programming language in 1986
- Useful for:
  - Specifying software more exactly
  - Assisting with semi-formal and formal verification of software
  - Catching errors due to incorrect use of a function
- Use of contracts is one way to achieve range/plausibility checks as recommended by most safety-critical software standards, such as IEC 61508 and ISO 26262

# Software contracts

- A contract consists of:
  - The pre-conditions for a function to execute correctly
  - The post-conditions after the function has executed correctly
  - The invariants that will hold before, during and after function execution
- Contracts typically map well to requirements
- Contracts must be checked before and after function execution
  - Some may be checkable at compile-time
  - Others need to be verified at run-time
- The use of contracts replaces many exceptions, making control flow visible

# Enforcing contracts in C++ software

- Created a library that provides contract enforcement in C++
- Explicitly define the pre-conditions, post-conditions and invariants in the software code
- The library provides the runtime checking necessary to enforce them
- Based on the contract enforcement/handling mechanisms proposed for C++20
- Contract types are defined enabling reusable checks for greater efficiency

# Function definition without contracts

```
/// @brief Toy example of a function without contracts.
float foo(float height, float deg, float scalar, size_t count)
{
   if (!std::isfinite(height) || (height < 0.0f)) { throw ... }
   if (!((deg >= 0.0f) && (deg < 90.0f))) { throw ... }
   if (!std::isfinite(scalar)) { throw ... }
   if (count > SOME_BOUND) { throw ... }

   // Convert degrees to radians
   auto rad = some_conversion_function(deg);

   // do some work, compute 'bar' of type float

   if (!std::isfinite(bar) || (bar <= 0.0f)) { throw ... }
   return bar;
}
```

With thanks to Mapless AI, Inc.

# Function definition with contracts

```
///
/// @pre  0 <= height < inf
/// @pre  0 <= deg < 90
/// @pre  -inf < scalar < inf
/// @pre  0 <= count <= SOME_BOUND
/// @post 0 < ret < inf
///
/// @implements{REQ001}
StrictlyPositiveRealf foo(NonnegativeRealf height, AcuteDegreef deg,
                          Realf scalar, SizeBound<SOME_BOUND> count)
{
   // Convert degrees to radians
   AcuteRadianf rad = deg;

   // do some work, compute 'bar' of type float

   return bar;
```

With thanks to Mapless AI, Inc.

# Enforcing contracts in C++ software

Get the library here:

https://github.com/ros-safety/contracts_lite

# Watchdogs

- Watchdogs are a commonly-used mechanism in safety-critical systems
- The watchdog watches the health/liveness of an entity critical to safe operation
- When a failure is detected, the watchdog takes its designed action
  - e.g. Resetting the system
  - e.g. Applying the emergency brakes

# Software watchdogs for ROS 2

- A set of nodes providing a watchdog mechanism for ROS 2-based systems
  - A node is a ROS 2 entity representing a unit of behaviour that can be executed
  - Nodes may be executed as individual processes, or combined into a single process
- Relies on the capabilities of “lifecycle” nodes and the underlying communications middleware
  - This is usually DDS

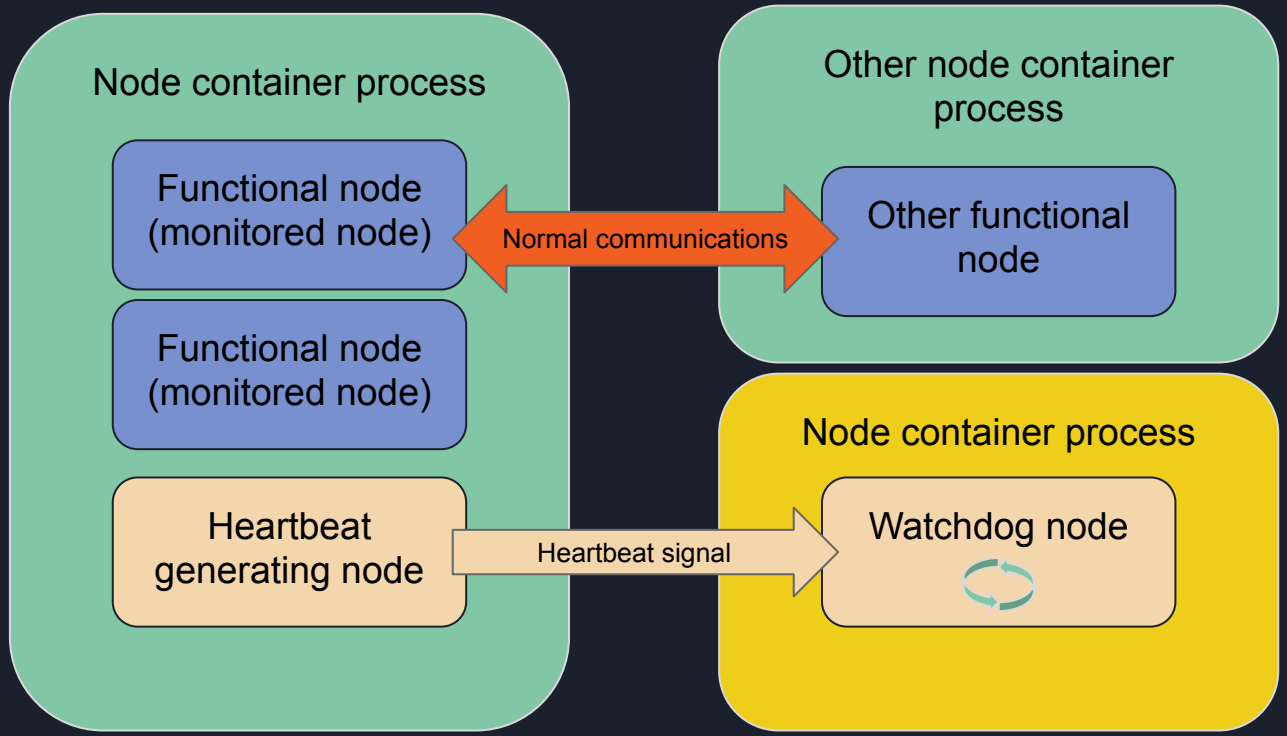
# Sending heartbeats

- Uses the DDS “heartbeat” QoS facility
  - Sends a regular heartbeat signal to anyone interested in knowing if the DDS entity is alive
- Heartbeat is provided by a dedicated node
  - Heartbeat is sent automatically while the node is alive
- Heartbeat node is composed in a single process with nodes that need to be monitored
  - Does not require modifying existing nodes’ source code to add a heartbeat

# Monitoring heartbeats

- Watchdog node whose purpose is to monitor a heartbeat signal
- Declares a failure if the heartbeat does not arrive at the configured period
  - Can be configured to allow a few heartbeats to slip within a specified window period, for robustness in jittery environments
- Failure is declared by the watchdog shifting into its error state,  where failure handling can be done

# Software watchdog structure



# Software watchdogs

Get the library here:

https://github.com/ros-safety/software_watchdogs

# Questions?

https://github.com/ros-safety/contracts_lite

https://github.com/ros-safety/software_watchdogs