XbotInterface API overview

Arturo Laurenzi, Luca Muratore

27th June 2017

XBotDays@IIT

Outline

- Introduction
- Class hierarchy
- Controlling the robot
- Kinematics and dynamics
- RT plugins examples

Introduction

- ***What's* XbotInterface?**

  An easily **configurable** API for robot control…

  … with integrated **kinematics/dynamics** modelling funcionalities!

- Configurable.. in terms of what?
  - **Robot structure**: no assumption is made, API is generated dynamically from standard configuration files
  - **Framework** used to actually communicate with the robot (ROS, YARP, RealTime interface, ...)
  - Specific **kinematics/dynamics library** to be used (RBDL, iDynTree, ...)

Introduction

Robot Control:

- Read joint states (joint position, velocity, torque, impedance, …)
- Read sensors (IMU, FT, …)
- Send commands to actuators (desired joint position, velocity, torque, impedance, …)

Integrated kinematics/dynamics:

- Synchronize a kinematic/dynamic model from the robot state (or just a part of it)
- Set a model state as a command to be sent to actuators

Introduction

Robot description:

- **URDF:** describes a robot as a kinematic tree, specifies links inertial properties and how links are connected with each other
- **SRDF:** semantic view of a robot as a collection of kinematic chains
- **Joint ID Map:** defines numeric IDs for joints. It is needed for the low level (EtherCAT) and also useful in higher-level code

```
# map between the joint robot id and
joint_map:
  1: NeckPitchj
  2: NeckYawj
  11: RShSag
  12: RShLat
  13: RShYaw
  14: RElbj
  15: RForearmPlate
```

Introduction

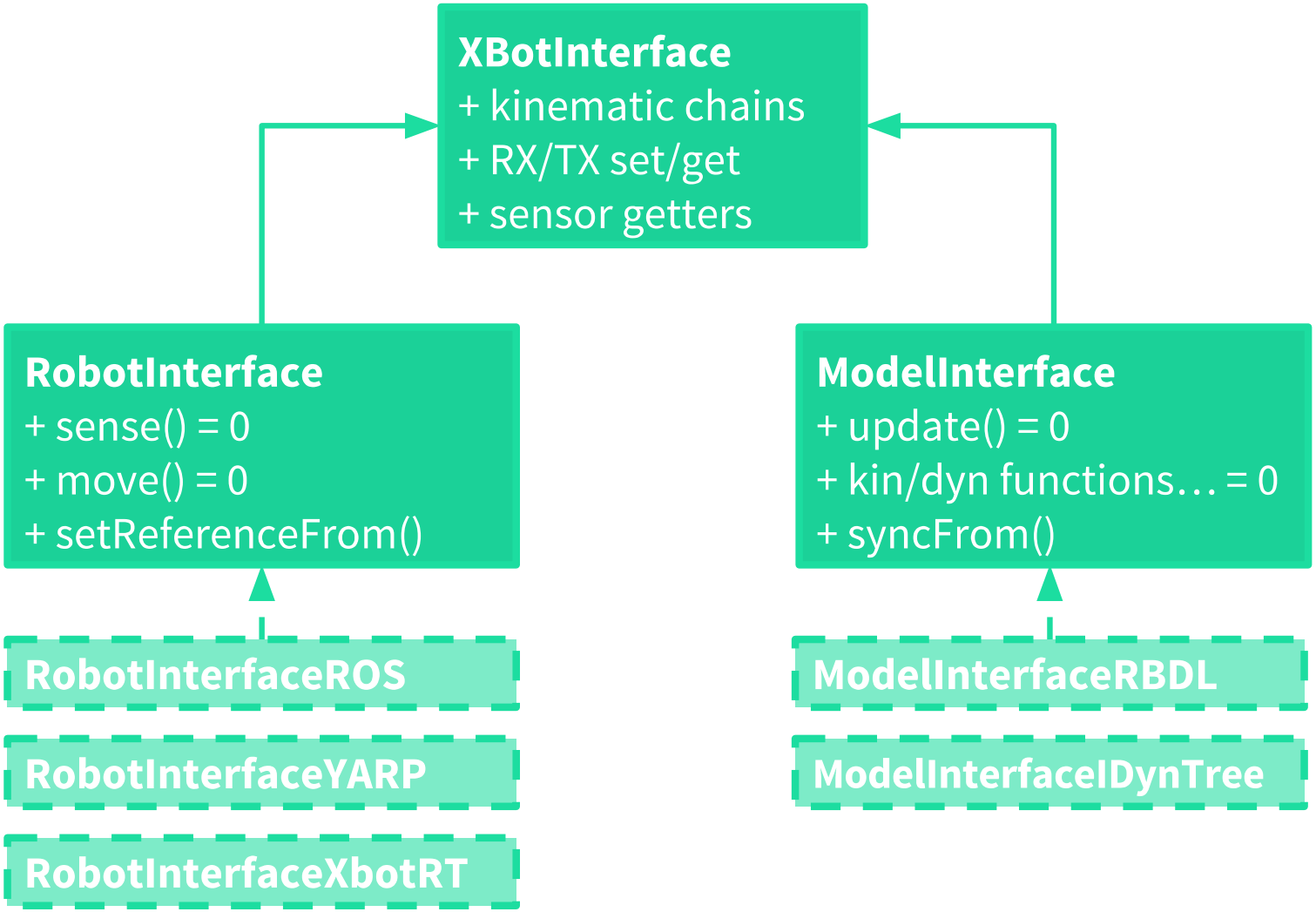
- XBotInterface can use **any framework** to communicate with a robot

  the framework to be used is defined ***at runtime*** (the same piece of code can run over ROS or YARP without even the need to recompile!)

- XBotInterface can use **any library** to perform kinematics/dynamics computations

  the library to be used is defined ***at runtime*** (the same piece of code can use RBDL or iDynTree without even the need to recompile!)

Class hierarchy



Class hierarchy

`class XbotInterface  {};`

is an organized container for the robot state

- The most atomic component is **Xbot::*Joint***, which holds the robot state variables (both RX and TX) for a single joint
- A **Xbot::*KinematicChain*** is a collection of Xbot::*Joints.* It provides the API to get/set RX/TX fields in a chain-wise way
- ***XbotInterface*** provides the API to get/set RX and TX fields both chain-wise and robot-wise

Class hierarchy

```
class RobotInterface : public XbotInterface {};
```

- Inherits from XbotInterface the organization in kinematic chains, getters and setters, …
- Adds a **sense()** method which collects data from all sensors and fills RX fields inside Xbot::*Joint*s
- Adds a **move()** method which collects TX data from all Xbot::*Joint*s and sends them to the robot
- Adds a **setReferenceFrom**() method to set commands according to a ModelInterface object

Class hierarchy

```
class ModelInterface : public XbotInterface {};
```

- Inherits from XbotInterface the organization in kinematic chains, getters and setters, …
- Adds an **update()** method which collects joint states from Xbot::*Joint*s and updates the kinematic/dynamic model accordingly
- Adds many **common kinematics/dynamics functions** (forward kinematics, jacobians, inverse dynamics, ...)
- Adds a **syncFrom**() method to set the model state from either another model or a robot

Controlling the robot

- **Getting a RobotInterface** is done by calling the static method `getRobot(path_to_config_file)` which returns a shared pointer to a RobotInterface

  You need to provide the path to a **master config file** which contains the following data:

  - Path to URDF, SRDF, Joint ID Map (relative to $ROBOTOLOGY_ROOT)
  - Framework to be used to communicate with the robot
  - Framework-specific parameters (e.g. ROS sensors topic names)
  - Library to be used for kinematics/dynamics
  - ...

Controlling the robot

```
# CENTAURO config
XBotInterface:
  urdf_path: "configs/ADVR_shared/centauro/urdf/centauro_upper_body_updated_inertia.urdf"
  srdf_path: "configs/ADVR_shared/centauro/srdf/centauro_upperbody.srdf"
  joint_map_path: "configs/ADVR_shared/centauro/joint_map/centauro_bonn_review.yaml"

RobotInterface:
  framework_name: "XBotRT"

ModelInterface:
  model_type: "RBDL"
  is_model_floating_base: "false"

RobotInterfaceROS:
  control_message_type: "AdvrCommandMessage"
  jointstate_message_type: "AdvrJointStateMessage"
  ft_topic_names: ["/ft_foot_1", "/ft_foot_2", "/ft_foot_3", "/ft_foot_4"]
  imu_topic_names: ["/imu_1_data", "/imu_2_data"]
  subclass_name: "RobotInterfaceROS"
  path_to_shared_lib: "libRobotInterfaceROS.so"
  subclass_factory_name: "robot_interface_ros"

XBotRTPlugins:
  plugins: ["HomingExample", "IkRosSharedMemoryPublisher", "OpenSotIk"]
  io_plugins: ["IkRosIo"]

RobotInterfaceXBotRT:
  subclass_name: "RobotInterfaceXBotRT"
  path_to_shared_lib: "libRobotInterfaceXBotRT.so"
  subclass_factory_name: "robot_interface_xbot_rt"

GazeboXBotPlugin:
  gains:
    j_arm1_1: {p: 5000, d: 30}
    j_arm1_2: {p: 8000, d: 50}
    j_arm1_3: {p: 5000, d: 30}
    j_arm1_4: {p: 5000, d: 30}
    j_arm1_6: {p: 2000, d: 5}
    j_arm2_1: {p: 5000, d: 30}
    j_arm2_2: {p: 8000, d: 50}
    j_arm2_3: {p: 5000, d: 30}
    j_arm2_4: {p: 5000, d: 30}
    j_arm2_6: {p: 2000, d: 5}
    torso_yaw: {p: 5000, d: 30}
  control_rate: 0.001
```

You can find some example XBotCore config files inside the **ADVR_shared** git repository!

Controlling the robot

What getRobot() does under the hood is to

- parse the URDF to understand what are joint names
- parse the SRDF to understand
  - what are the kinematic chains
  - which joints are disables
  - which links are associated with IMUs/Fts
- start the communication with the robot (e.g. ROS topic subscription/advertisement, YARP ports initialization, ...)

Controlling the robot

- **Updating the robot with latest sensor readings**

  is done by calling the ***sense()*** method. Under the hood, it

  - collects data from the framework

    ROS: looks for new messages on joint_state/imu/ft topics

    XbotRT: reads newest joint states from PDO
  - updates the Xbot::*Joint* internal variables
  - updates Xbot::*ImuSensor*s and *ForceTorqueSensor*s
  - updates an internal *ModelInterface* object with the robot state

Controlling the robot

- **Extracting joint states from a RobotInterface**

  XbotInterface provides getters returning

  - std::map <joint ID, joint value>
  - std::map <joint name, joint value>
  - Eigen::VectorXd

  The same getters can also be called on *KinematicChains*

```
JointIdMap right_arm_velocity;
robot.chain("right_arm").getJointVelocity(right_arm_velocity);

Eigen::VectorXd torso_torque;
robot("torso").getJointEffort(torso_torque);

JointNameMap robot_torque;
robot.getJointPosition(robot_torque);

Eigen::VectorXd robot_position;
robot.getJointPosition(robot_position);
```

Controlling the robot

- **Imu & Force-torque sensors**

  A RobotInterface provides two ways of reading them

  - getFt() → returns a std::map <sensor name, shared pointer to *Xbot::ForceTorqueSensor*>
  - getFt(parent_link_name) → returns a shared pointer to *Xbot::ForceTorqueSensor*

  Then, you ask the Xbot::*ForceTorqueSensor* for the measured wrench

```
ForceTorqueSensor::ConstPtr l_arm_ft = robot.getForceTorque(robot("left_arm").getTipLinkName()
Eigen::Vector6d l_arm_wrench;
l_arm_ft->getWrench(l_arm_wrench);
```

  Note! Sensor names are specified inside the SRDF, look for `imu_sensors` and `force_torque_sensors` tags

Controlling the robot

- **Sending commands to the robot**

  First, you need to update the TX fields inside the Xbot::*Joint*s. This is done using the provided setters, either chain-wise or robot-wise.

  Then, you call the ***move()*** function, which

  - Collects commands from Xbot::*Joint*s
  - Sends them over the framework

    ROS: publish a message on the command topic

    XbotRT: fill the EtherCAT PDO

Controlling the robot

Miscellaneous features

- `getJointLimits(), getVelocityLimits(), getEffortLimits()`
- `checkJointLimits(), …`
- `operator<< to print the robot/chain/joint state`
- `getRobotState() gets e.g. an homing configuration from the SRDF`
- `log the whole robot state to a MAT file`
- `...`

Kinematics and dynamics

Why to introduce a model abstraction layer?

- future-proof your code
- get a cleaner API

If someone comes up with a more-efficient library, it can be used effortlessly in all existing code

Efficient libraries sometimes have a clumsy API...

```
RigidBodyDynamics::Utils::CalcCenterOfMass(_rbdl_model, _q, _qdot, mass, _tmp_vector3d, nullptr, nullptr, false);
```

```
Eigen::Vector3d com_pos;
model.getCOM(com_pos);
```

Kinematics and dynamics

- **Getting a ModelInterface** is done by calling the static method `getModel(path_to_config_file)`, which returns a shared pointer to a ModelInterface.
- **Getting/setting the model state** is done via the same interface of RobotInterface (they both inherit from XbotInterface!)
- **Remember to call update()!** Recall that setting the model state only updates some variables inside the Xbot::*Joint*s.

```
Eigen::VectorXd q = ..., qdot = ..., qddot = ..., tau;
model.setJointPosition(q);
model.setJointVelocity(qdot);
model.setJointAcceleration(qddot);
model.update();
model.computeInverseDynamics(tau);
```

Kinematics and dynamics

- **Exchange information between Robot and Model**

  Inside a control loop, there can be the need to

  - set the model state according to the robot state; this is done with `model.syncFrom(robot)`
  - send the state of a model as a command to the robot; this is done with `robot.setReferenceFrom(model)`

  In either cases, you can select which fields to synchronize by using the appropriate flags

```
model.syncFrom(robot, Sync::Position, Sync::Effort);
```

Take care! Exchanging data using vectors is unsafe!

Kinematics and dynamics

Some supported functionalities…

- `getPointPosition(), getOrientation(), getPose(), getCOM(), ...`
- `getVelocityTwist(), getAccelerationTwist(), computeJdotQdot(), getComVelocity(), ...`
- `getJacobian()`
- `computeGravityTerm(), computeNonlinearTerm(), computeInverseDynamics(), computeInertialTerm()`
- `getInertiaMatrix()`

Note! A RobotInterface has an internal model which is automatically updated with the robot state whenever you call **sense()**! You can get it with a `robot.model()` !

Writing RT plugins

1) **generate a skeleton**
2) **implement five functions:**
   1) `init_control_plugin(...)`
   2) `on_start(double time)`
   3) `on_stop(double time)`
   4) `control_loop(double time, double period)`
   5) `close()`

Example 1 – Computed torque

Goal: $\boldsymbol{\tau} = B(\boldsymbol{q})\ddot{\boldsymbol{q}}_d + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + B(\boldsymbol{q})(K_D\dot{\tilde{\boldsymbol{q}}} + K_P\tilde{\boldsymbol{q}})$

```
35  void XBot::IdExample::on_start(double time)
36  {
37      /* Acquire joint position on plugin start */
38      _robot->model().getJointPosition(_q0);
39
40      /* Save plugin start time */
41      _first_loop_time = time;
42
43      /* Set impedance to zero in order to perform pure torque control */
44      _robot->setStiffness(Eigen::VectorXd::Zero(_robot->getJointNum()));
45      _robot->setDamping(Eigen::VectorXd::Zero(_robot->getJointNum()));
46
47      /* Update the model with robot state */
48      _model->syncFrom(*_robot);
49  }
```

Example 1 – Computed torque

Goal: $\boldsymbol{\tau} = B(\boldsymbol{q})\ddot{\boldsymbol{q}}_d + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + B(\boldsymbol{q})(K_D\dot{\tilde{\boldsymbol{q}}} + K_P\tilde{\boldsymbol{q}})$

```
50
51  void XBot::IdExample::control_loop(double time, double period)
52  {
53      const double PERIOD = 1.0;
54      const double OMEGA = 2.0 * 3.1415 / PERIOD;
55
56      /* Compute reference trajectory up to 2nd order derivatives */
57       qref =  q0 +  dq * std::sin(OMEGA*(time- first loop time));
58       qdotref =  dq * std::cos(OMEGA*(time- first loop time)) * OMEGA;
59       qddotref = -1.0 *  dq * std::sin(OMEGA*(time- first loop time)) * OMEGA * OMEGA;
60
61      /* Set actual robot positions / velocities to the model used for ID */
62       model->syncFrom no update(* robot, Sync::Position, Sync::Velocity);
63
64      /* Set acceleration reference */
65       model->setJointAcceleration( qddotref);
66       model->update();
67
68      /* Set tau = B*qddotref + h(q, qdot) */
69       model->computeInverseDynamics( tau);
70
71      /* Add the PD term */
72       model->getInertiaMatrix( B);
73       robot->model().getJointPosition( q);
74       robot->model().getJointVelocity( qdot);
75
76      const double kp = 50, kd = 15;
77
78       tau +=  B*( kp*( qref- q) + kd*( qdotref- qdot) );
79
80       model->setJointEffort( tau);
81
82      /* Set torque command from the model */
83       robot->setReferenceFrom(* model, Sync::Effort);
84
85      /* Send command to the robot */
86       robot->move();
87
88  }
```

Example 2 – Shared memory

In example 1 we were generating references inside the same plugin which is in charge of executing them

To improve code modularity, we now want to modify the computed torque plugin so that it can read references from other plugins

XBotCore supports fast communication between plugins via a **shared memory** mechanism

1) Declare a `SharedObject<TypeName>`
2) Assign it with the return value of `SharedMemory::get<TypeName>(shared_obj_name)`
3) Use the shared object as if it was a pointer (*, →)

Example 2 – Shared memory

```
private:

    XBot::SharedObject<Eigen::VectorXd> _qref, _qdotref, _qddotref;
```

```
10
11  bool XBot::IdExample::init_control_plugin(std::string path_to_config_file,
12                                            XBot::SharedMemory::Ptr shared_memory,
13                                            XBot::RobotInterface::Ptr robot)
14  {
15      /* Save the robot to a class variable */
16      _robot = robot;
17
18      /* Get a model */
19       _model = ModelInterface::getModel(path_to_config_file);
20
21      /* Initialize shared objects */
22      _qref = shared_memory->get<Eigen::VectorXd>("/joint_position_reference");
23      _qdotref = shared_memory->get<Eigen::VectorXd>("/joint_velocity_reference");
24      _qddotref = shared_memory->get<Eigen::VectorXd>("/joint_acceleration_reference");
25
```

Recall that shared objects behave like pointers, so they must be dereferenced...

```
74      /* Set acceleration reference */
75      _model->setJointAcceleration(*_qddotref);
76      _model->update();
```

Example 3 – Cross-domain communication

What if we want to exchange data between the RT and the NRT domains?

For this purpose, Xenomai provides **XDDP** (Cross-domain datagram protocol) **pipes**. It is quite a low-level tool:

- the sender writes a known number of bytes on the pipe
- the receiver reads the known number of bytes and casts the result to the desired type (class)

It only works for byte-serializable classes, i.e. **classes cannot have dynamic arrays as members!! (as std::vector, Eigen::***Xd, std::string, …)**

Also, Xenomai API is quite cumbersome (it *is* a low level tool!)

Example 3 – Cross-domain communication

To abstract away some of this complexity, XbotCore provides:

- `PublisherRT<TypeName>, SubscriberRT<TypeName>`
- `PublisherNRT<TypeName>, SubscriberNRT<TypeName>`

They all provide a `init(pipe_name)` and a `read()/write()`

```
private:

    Eigen::Matrix<double, 15, 1> _qref, _qdotref, _qddotref; // Fixed-size column vectors
    XBot::SubscriberRT<Eigen::Matrix<double, 15, 1>> _rtsub_q, _rtsub_qdot, _rtsub_qddot;
```

```
    _rtsub_q.init("joint_position_ref");
    _rtsub_qdot.init("joint_velocity_ref");
    _rtsub_qddot.init("joint_acc_ref");
```

init_control_plugin

```
67
68      /* Receive references from NRT */
69      bool qref_received = _rtsub_q.read(_qref);
70      bool qdotref_received = _rtsub_qdot.read(_qdotref);
71      bool qddotref_received = _rtsub_qddot.read(_qddotref);
```

control_loop