XBot Real-Time Software Framework for Robotics: From the Developer to the User Perspective

Luca Muratore[1,2], Arturo Laurenzi[1], Enrico Mingo Hoffman[1], and Nikos G. Tsagarakis[1]

[1]Humanoids and Human Centered Mechatronics (HHCM), Istituto Italiano di Tecnologia, Genova, Italy
[2]School of Electrical and Electronic Engineering, The University of Manchester, M13 9PL, UK
{luca.muratore, arturo.laurenzi, enrico.mingo, nikolaos.tsagarakis}@iit.it

***Abstract*— The widespread use of robotics in new application domains outside the industrial workplace settings necessitates robotic systems that demonstrate functionalities far beyond those of the classical industrial robotic machines. These emerging applications involve complex tasks that also vary and have to be carried out within a partially unknown environment requiring autonomy and adaptability, which further increase the intricacy of the system software architecture. To cope with the demands and the consequent complexity of the robotic systems and their control, software infrastructures that can be quickly and seemly adapted to these requirements, while providing transparent and standardized interfaces to the robotics developers and users, are needed. In this work we introduce the *XBot* software framework. The development of the *XBot* was driven by the need to provide a software framework that abstracts the diverse variability of the robotic hardware (effectively becoming a cross robot platform framework), provides deterministic hard Real-Time (RT) performance, incorporates interfaces which permit the possibility to integrate state of art robot control frameworks, and delivers enhanced flexibility through a plug-in architecture. The paper presents the insights of the *XBot* framework from the developer to the user perspective, discussing the details of the implementation mechanisms adopted as well as providing tangible examples on the use of the framework.**

I. Introduction and Design Goals

The continuous advancement of robotics with the incorporation of new skills and capabilities required to address effectively applications within unstructured workplaces, has inevitably increased the complexity of robotic hardware, software and control components. As a result the inevitable complexity of today's robots targeting to new domains in partially unstructured environments has reached a noticeable extent, e.g. such robots typically consist of a large number of sensors, actuators, and processors executing numerous control modules that communicate through several and diverse interfaces. To tackle this, several software frameworks have been developed in the past twenty years [1] targeting to provide flexible infrastructures, which not only permit to seemly integrate new functionality and interfaces in the robotic system, but also ensure standardization and easy tracking and maintenance of the software development, despite the increased complexity.

The selection among these available software middlewares is not a trivial task for the research community as well as for companies interested to explore them. Apart from having to deal with the software intricacy, these frameworks have to provide hard RT performance ensuring deterministic response times [2] as required in critical tasks when robots need to perform in autonomous mode, responding to disturbances and interacting with the physical environment. Thus, an essential feature of a software framework for robotics is the RT safeness and the RT scheduling, both needed for effective control, especially when executing high frequency and low jittering (e.g. 1 kHz) control cycles.

Furthermore, a software middleware needs to abstract the complex hardware (e.g. actuators and sensors) of the robot providing a simple, standardized Application Programming Interface (API). In fact a robot can be considered a distributed system composed by a set of hardware devices communicating through a fieldbus. Efficiently developing control and application software that can be shared, ported and reused in different robots with minimum effort, is another fundamental requirement for the software architecture. An important component needed to achieve this goal is the Hardware Abstraction Layer (HAL), which can be incorporated to mask the physical hardware details (e.g. kinematics model, sensor, update frequency, etc) varying from one robot to another. The HAL can provide a relatively uniform abstraction layer that assures portability and code reuse: it permits to develop control modules and easily port them from one robot to another.

The existing robotics software frameworks address different needs and requirements [1], therefore one of the key aspects for a brand-new middleware is the interoperability with well-known and established robotic software platforms. Interoperability should ideally allow users to execute existing software without the necessity of (i) changing the current code and (ii) writing hand-coded "bridges" for each use case [3].

Among state-of-the-art robotics middlewares, we recall *OROCOS* [4] (Open Robot Control Software), an RT framework, which permits to develop robotics control applications consisting of multiple interacting *components*. For strict RT applications, *OROCOS* allows to schedule *components* in a single process while it relies in the background (hidden for normal users) on the Common Object Request Broker [1] (CORBA) architecture for Inter Process Communication (IPC), implemented in C++ using ACE/TAO[2]. *Components*

[1]http://www.omg.org/spec/CORBA/
[2]http://www.theaceorb.com/

IEEE Robotics and Automation Magazine (RAM) paper, presented at IROS 2020. It should be cited as a RAM paper.

may run in a single or separated threads depending on their activities. Despite *OROCOS* is used in a fair number of robotics projects, the framework maintenance as well as the community looks not being very active anymore[3].

A very similar framework to *OROCOS* is *OpenRTM-aist* [5], developed in Japan from 2002 under NEDO's (New Energy and Industrial Technology Development Organization) "Robot challenge program". It is based on CORBA, which increase the software complexity for the end-users/developers. Moreover, a part of *OpenRTM-aist* documentation is available only in Japanese that further impedes its utilization.

PODO [6] is the framework used by KAIST (Korea Advanced Institute of Science and Technology) in the DRC-HUBO robot during the Darpa Robotics Challenge Finals [7]. Its control system has RT control capabilities and its inter-process communication facilities are based on POSIX IPC; moreover it uses a shared memory system called MPC to exchange data between processes on the same machine. This heterogeneous system has the potential to cause confusion, as clearly stated in [8] inside Section 2.1, as it is unclear which architectural style must be used to communicate with a specific component.

YARP (Yet Another Robot Platform) [9] and *ROS* (Robot Operating System) [10] are popular component-based frameworks for IPC that do not guarantee RT execution among modules/nodes. However it is essential to have a component responsible for the RT control of the robot, making these frameworks only suitable as external (high-level) software frameworks. We should mention here that *ROS 2* is moving towards the RT support[4] using the DDS (Data Distribution Service) middleware: anyhow it is still in an early stage development phase, so it can't be used in real-world scenario[5].

In [11] an RT architecture based on OpenJDK is introduced (used by IHMC during the DRC Finals). Nevertheless, to their own admission [12], none of the commercially available implementations of the Java Real Time Specification had the performance required to run their controller. In other words, the existing Real-time Java Support is insufficient.

The above considerations and limitations of the existing frameworks motivated us to develop the *XBot* framework having in mind that the design of a software platform, which lies at the foundations of such complex and diverse robotic systems, is the most crucial phase in the software development process. *XBot* was designed to be a user friendly, flexible and reusable middleware for both RT and non-RT robotics control. *XBot* was developed starting from the following design goals and features:

- **Hard RT control performance**: it must perform computation inside specific timing constraints with minimum timing jitter. There are several operating systems or platforms which support RT operation, like Windows CE, INtime, RTLinux, RTAI, Xenomai, QNX, VX-Works. We selected a Linux based Real-Time Operating System (RTOS) to avoid a licensed product that does not give us the possibility to modify and adapt the source code to fit it to the specifications of our system. In particular, Xenomai satisfies the requirements for extensibility, portability and maintainability as well as ensures low latency as stated in [13].
- **High control frequency**: robotics applications may often require high frequency control loops, e.g. RT Pattern Generator for Biped Walking, impedance regulation controllers or force feedback modules
- **Cross-Robot compatibility**: it should be possible to use it with any robot, without code modification. It is crucial to be able to reuse the software platform with different robots, or subsystems of the same robotic platform
- **External Framework integration**: it should be possible to use *XBot* as a middleware for any kind of external software framework (RT or non RT) without tailored software or specific bridge for every different case.
- **Plug-in Architecture**: users and third parties should be able to develop and integrate their own modules. In a robotic system platform we need a highly expandable software structure
- **Light-weight**: small number of dependencies on other libraries, it should be easy to install and set up. It is expected to run *XBot* on embedded PCs which can pose low performance requirements in terms of memory and CPU for the control framework in use.
- **Simplicity**: it must be simple. Complex systems may have unneeded and over-engineered features. For robotic applications we need the full control over the software platform. *KISS* ("Keep It Simple, Stupid") principle is essential and unnecessary complexity should be avoided
- **Flexibility**: *XBot* has to be easy to modify or extended to be used in systems and applications other than those for which it was specifically designed

Finally, the *XBot* software framework was not developed to address the requirements of a specific robotic platform, instead its implementation is flexible, generic and cross-robot. Furthermore it does not depend on any existing software or control platform, but it provides to the user the functionality to easily integrate any RT or non-RT frameworks.

II. Framework

As presented in Figure 3, the *XBot* software architecture is composed of different components, described in details within the following sub-sections. Each of them has a dedicated role and functionality and contributes to the realization of one or more of the design goals described in the previous section. Figure 1 presents a detailed view of the threads spawn in the main components of the framework. To avoid scheduling issues and keep the complexity of the software infrastructure as low as possible we currently employ only two RT threads and one non-RT thread in the framework.

[3]In particular we refer to the discontinuity in maintaining the framework under last versions ($\geq$ 3.x) of Ubuntu Xenomai, where the OROCOS porting is still experimental

[4]`http://design.ros2.org/`

[5]`https://index.ros.org/doc/ros2/Features/`

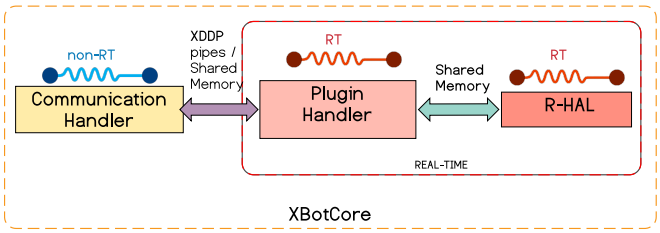

Figure 1. *XBot* threads structure and communication mechanisms.

A. R-HAL

The Cross-Robot compatibility feature is achieved through the development of a suitable hardware abstraction layer [14], which enables the user to efficiently port and run the same control software modules on different robots, both in simulation and on the real hardware platforms. The main idea is to provide an independent layer with respect to the robot hardware and high-level software enabling the integration of new actuators, sensors or other hardware components.

Concerning the threads configuration, *XBot* employs a separate thread to execute the low-level robot control loop and permits to realize controllers with different frequencies. Synchronization between the *Plugin Handler* thread and the *R-HAL* thread is implemented using condition variables and it is needed to safely access the shared data structures.

XBot currently supports EtherCAT (for robots like WALK-MAN, CENTAURO and COMAN+), Ethernet (for COMAN), and KUKA LWR 4 / KUKA LBR arm based robots [15], [16], [17]. The possibility to simulate the robot and its controllers behaviours prior to testing on the real hardware is essential, especially when dealing with complex robotic systems. To achieve this we provide an *R-HAL* implementation for the well known *Gazebo*[6] simulator environment Figure 2. In particular we rely on the *Gazebo* `ModelPlugin` class to be part of the Gazebo internal loop.

B. Plugin Handler

The main component of the *XBot* architecture is called *Plugin Handler* and it is represented in Figure 3 with dark pink colour. The software design of this component relies on two core requirements of a robotic system (described in section II): the RT control and the highly expandable software structure. To achieve this the *Plugin Handler* is implemented using a single RT thread running at high frequency (e.g. 1 kHz) and is responsible for the following actions with the order they appear below.

1) load the set of plugins as shared objects in the filesystem requested by the user from a configuration file,
2) initialize all the loaded plugins, and start them upon user request,
3) execute the started plugins sequentially,
4) reload and reinitialize a plugin upon user request,
5) close and unload all the loaded plugins.

[6]http://gazebosim.org/

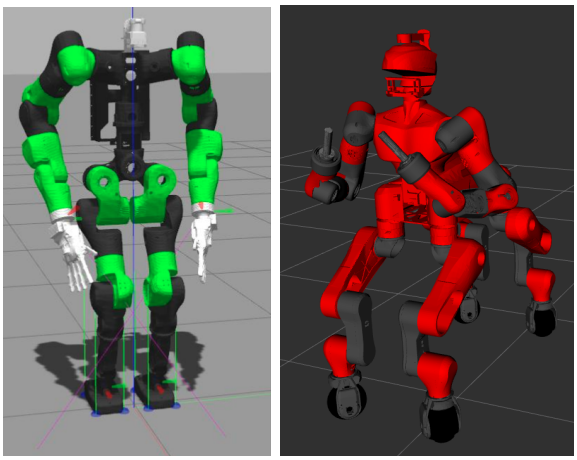

Figure 2. COMAN+ robot controlled inside the Gazebo simulator (left) and CENTAURO robot in RViZ (right): both using two different implementations of the *R-HAL* provided in the *XBot* software architecture.

In Figure 4, the UML state diagram representing the life-cycle of a plugin is presented.

The Plugin implementation is compiled as a shared object library (.so). In details a Plugin is a simple class inheriting from the abstract class `XBotControlPlugin`; this means that writing a Plugin is straightforward for the user, as he/she just needs to implement three basic functions:

- an `init_control_plugin()` function, which is called by the *Plugin Handler* after the plugin is loaded/reloaded and is useful to initialize the variables of the Plugin
- a `control_loop()` function, which is called in the run loop of the *Plugin Handler* after the plugin is started
- a `close()` function, which is called in the *Plugin Handler* closing phase

A ready to use code generator script is provided inside the *XBot* framework, to enable the user to create a skeleton of a new plugin in no time, as described in section IV.

The user might have the need to run a set of Plugins in the non-RT layer without modifying their implementation: the *NRTDeployer* component, represented in Figure 3 with green colour, is provided to emulate the *Plugin Handler* behaviour in the non-RT layer. This can be useful for not-expert users, since they can use the same RT *XBot* plugin structure, but they do not have to deal with the constraints that the RT kernel is posing, i.e. mainly to not have any context switches inside the control loop by avoiding non-RT system calls.

C. Communication Handler

The above mentioned software components do not give the possibility to communicate with external modules/hosts outside the robot: for this purpose the software framework of a robotic system should incorporate a set of non-RT threads that permit the communication of the system with remote pilot stations or cloud services. To this aim, *XBot* provides the *Communication Handler* component as

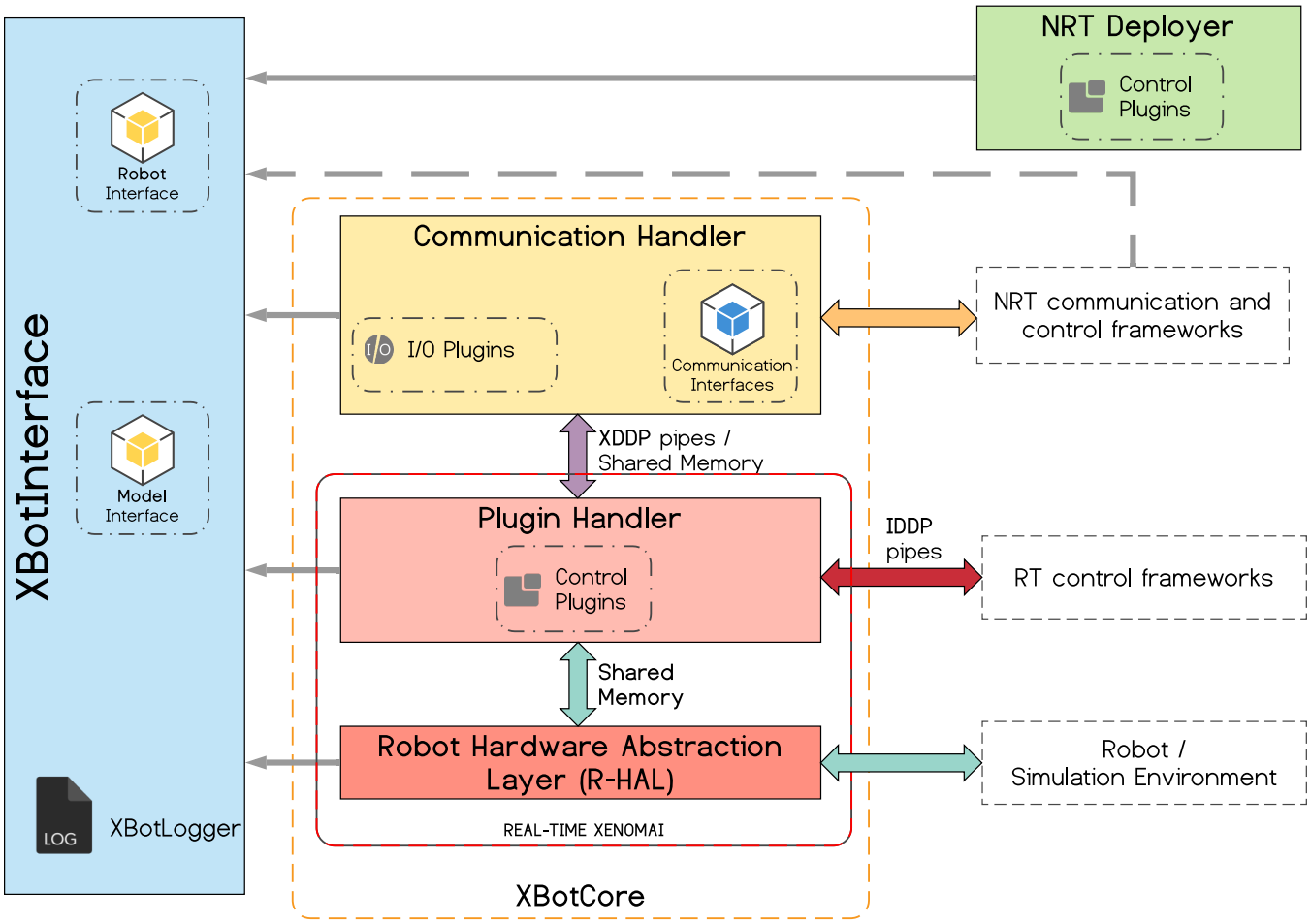

Figure 3. *XBot* software architecture: components overview and interaction.

a non-RT thread with the primary goal of servicing all operations which would break determinism. Data exchange between such a thread and the *Plugin Handler* is done either via shared memory (relying on lock-free synchronization patterns on the RT side), or by exploiting a Xenomai-specific datagram protocol named XDDP (Cross Domain Datagram Protocol) which achieve asynchronous communication between RT and non-RT threads, without any *mode switches*[7]. Similar to the *Plugin Handler*, a dynamic loading mechanism is employed in order to achieve easy expandability of the systems in terms of the non-RT components that can be loaded. Such components belong to two categories: (i) `CommunicationInterfaces`, which implement the framework-specific robot API (e.g. broadcasting robot TFs and joint states to ROS topics), and (ii) `IOPlugins`, which provide access to the shared memory and XDDP pipes. As for the standard Plugin, *XBot* provides a ready-to-use skeleton (simple script to run) for the user who wants to implement a new `IOPlugin`. The execution loop of the `Communication Handler` thread is responsible for updating the internal robot state using the XDDP pipe with the non-RT robot API, sending the robot state to all the communication frameworks implemented as `CommunicationInterfaces`, receiving the new reference from the "master" `CommunicationInterface` (to avoid having multiple external frameworks commanding the robot) and finally for sending the received reference to the robot using the XDDP non-RT robot API.

[7]In a Xenomai system, threads can operate either in "primary" mode (managed by the real-time scheduler), on in "secondary" mode (managed by the standard Linux kernel). The term *mode switch* then refers to a RT thread involuntarily switching to secondary mode due to a non-RT system call.

D. XBotInterface

After the design and the implementation of the latency-free, hard real-time layer the next significant feature is accompanied by the implementation of flexible interfaces, which permit our framework to integrate with state-of-art, widely spread robot control frameworks like ROS, YARP, and OROCOS. During the initial phase of our design, we immediately recognized the importance of providing the user with a standard way of communicating with the robot, regardless of its specific structure (humanoid, quadruped, manipulator, etc), and also independently of the particular software layer that the user wanted to operate within. To satisfy this we aimed at developing an API that could be used to send commands to a robot, and receive its current

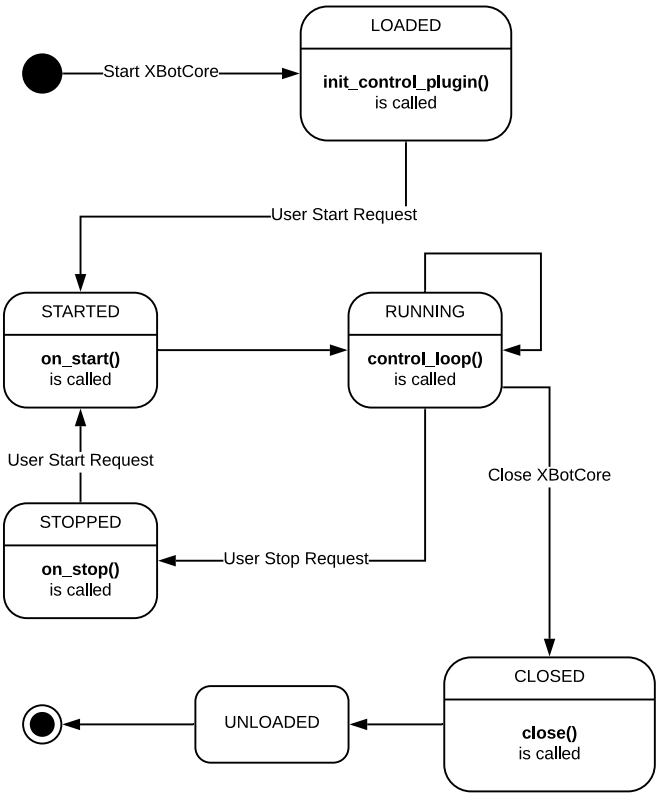

Figure 4. UML state diagram showing a *XBot* plugin lifecycle.

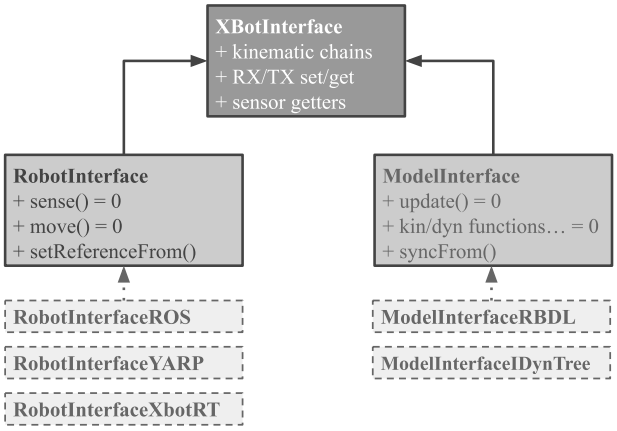

Figure 5. UML class hierarchy diagram for the the XBotInterface library.

state, from a ROS node, an OROCOS component or a *XBot* RT plugin, in a uniform way. With this in mind, we started the design of the *XBotInterface* library.

An object of the XBotInterface class is essentially a big, organized, container for the robot state, including its onboard controllers. As such, it includes quantities that describe the measurements coming from the robot sensors (e.g. joint position and torque, motor current, IMU states, force/torque sensing etc), and control references (joint position, torque and impedance, etc) as well. As a complex robot, e.g. a humanoid can have as many as forty joints, it is important to organize such a state rationally. To this aim, we define the `XBot::Joint` to be the most atomic component of the library. We then define the `XBot::KinematicChain` as a collection of joints belonging to the same chain. This enables the user to specify a joint by the name of the chain it belongs to, and its position inside the chain itself, rather than remembering its literal name or its position inside a possibly huge vector of joints. However, this semantic approach to the robot description comes with disadvantages as well, mainly because of the fact that it is inconvenient to use it for the development of control algorithms, that often rely on the manipulation of the joint states according to the rules of linear algebra. Hence, we also decided to provide a full-robot interface that relies on *Eigen3*, a state-of-art linear algebra library. Furthermore, interfaces to two families of sensors that are crucially important for real-time control, i.e. force-torque (FT) sensors and inertial measurement units (IMU) were implemented and incorporated in the *XBotInterface* library.

While the XBotInterface class organizes the robot state, to enable actual communication with a robot, we defined the `RobotInterface` as a subclass of *XBotInterface* that introduces a `sense()` method for collecting sensory feedback from the robot, and a `move()` method for sending reference commands to the robot. Both functions are defined as pure virtual, since their implementation depends on whether the `RobotInterface` is being used from a ROS node, an OROCOS component, or a *XBot* RT plugin.

Besides communicating with the robot, it is often the case that a piece of control code may involve kinematic and dynamic computations, which are performed by some external library. Such library must take a model state (e.g. joint positions, velocities, and acceleration) as input, and return the joint references (e.g. joint positions or torques) as output. Both states are usually in the form of arrays, which are arranged according to an order that is specific to the library itself. Again, we found this to be an inconvenient and error-prone format for the human user, especially in the case of complex multi-chained robots. In an effort to ease the user's work, we decided to reuse our robot state description, which is the *XBotInterface* class, with the following three main goals in mind:

- provide a uniform interface not only to the state of actual robots, but also of the corresponding model counterparts;
- standardize the API for retrieving the outcome of the most common kinematic and dynamic computations;
- ease the data exchange between a robot and a corresponding model, and vice-versa.

We achieve these objectives by defining the `ModelInterface` as a subclass of *XBotInterface*, which mainly adds an *update()* method where the underlying kinematic/dynamic library is updated with the current model state. Similarly, it is a pure virtual function whose

implementation depends on the specific back-end that is being used. In addition, we also introduce a minimal set of pure virtual methods for every fundamental algorithm from the domains of kinematics (e.g. forward kinematics, differential kinematics, ...) and dynamics (e.g. inverse dynamics, bias terms, mass matrix computation, ...) . As a third step, we also provide functions for synchronizing robots and models, i.e. for setting a model state from the sensory feedback from the robot, and for turning a model state into a reference for the robot controllers. In our experience, these two syncing methods have turned out to be useful especially when robot and model do not have the same structure, as it is often the case when dealing with complex robots. As a simple example, a manipulation module may use a model for just the upper-body of a legged robot, while the robot object always takes into account the system as a whole. The resulting class hierarchy for the *XBotInterface* library is summarized in Figure 5.

E. RT software middlewares - The OROCOS integration

The *XBot* framework provides the `RT communication` plugin for the integration of any other RT software framework thanks to the use of the IDDP (Intra Domain Datagram Protocol) pipes, a Xenomai-specific datagram protocol for RT inter-process communication. The difference between the XDDP and the IDDP protocols is that the former is available for mode-switches-free data exchange between Xenomai threads and regular Linux threads (RT $\Longleftrightarrow$ non-RT), while the latter enables real-time threads to asynchronously exchange datagrams within the Xenomai domain, via socket endpoints with a latency-free communication(RT $\Longleftrightarrow$ RT). In the OROCOS use-case, we implemented the `XBotOrocos RTT Task` component to have a transparent communication between the `OROCOS Task`, implemented as another *OROCOS* component, and the target robot. The above integration was used and validated in [18].

F. Non-RT software middlewares - The ROS integration

During an initial phase of the *XBot* development, the target for the RT part of our architecture was to execute mostly low level control algorithms that would not need too much flexibility for their I/O operation from and towards the higher level non-RT domain. The reader could think, for instance, of a closed-loop inverse-dynamics plugin, which provides feed-forward torques to the actuators, or of a center-of-mass stabilizer that is used for balancing and locomotion. Modules of this kind do not need to exchange much information with external pieces of software. On the contrary, it turned out later that our users wanted to take advantage of the low latency guarantees granted by the RT layer to implement complex controllers that *do* need flexible I/O in terms of receiving *reference set-points*, online *parameter tuning*, and sending information on the *controller state*. Even though it was possible to build such an infrastructure leveraging the flexibility of the communication handler loop by means of *IOPlugins* (as discussed in Section II-C) in combination with XDDP communication, we chose to reuse as much as possible standard tools that are well established in the robotics community such as *ROS subscribers*, *publishers*, *service servers*, and *dynamic reconfigure*. While it is not immediately possible to use these tools from the RT domain, we realized that our threading structure allowed us to adapt them with moderate effort, and with no required modification to the ROS source code.

More in detail, subscribers, service servers, and dynamic reconfigure are based on callbacks. Since the data reception part involves system calls to the Linux kernel, such callbacks should be processed by a non-RT thread, which in our case is the *Communication Handler*. Once received, callbacks are packaged as `std::function`-like objects and sent to the RT thread for execution using lock-free queues.

Publishers are slightly more complex to adapt; the standard workflow when using publishers from a normal process would be to produce a message, and then to call a `publish()` function that serializes it and sends the corresponding bytes via TCP. The data transmission part needs to be performed from a non-RT thread, since it involves a system call to the Linux kernel. On the other hand, the serialization part should be done where the message is produced, since it is the only place where its *type* is known. Consequently, it is necessary to "split" the `publish()` function in a way such that the serialization is performed on the RT plugin, while the actual publishing is done on the communication handler. Indeed, ROS allows us to take over control of such details by using its advanced API. Summarizing, we enable the developer of a RT plugin to:

- subscribe to arbitrary ROS topics of any message type (including custom messages) without the need for any adaptation step;
- implement a service server inside a *XBot* RT plugin;
- publish arbitrary messages to a topic;
- tune online the module parameters with the popular ROS dynamic reconfigure tool.

III. User Tutorial

The starting point for the user tutorial is the following GitHub repository: `https://github.com/ADVRHumanoids/XBotControl`.

This contains all the releases of the XBotControl framework which includes not only the *XBot* software platform, but also the *OpenSoT*[19] and the *CartesI/O*[20] robot control libraries.

Once the installation and configuration steps are executed following the online instructions[8], the user will be able to run a basic example usage of the framework, since a set of ready-to-use robot model and plugins will be available on the user's system.

XBot needs just a YAML configuration file as an input; for example we can try to control the CENTAURO robot in simulation doing the following:

- set the CENTAURO YAML configuration file for the basic example we want to run:

[8] `https://github.com/ADVRHumanoids/XBotControl/wiki/Install,-Configure,-Uninstall`

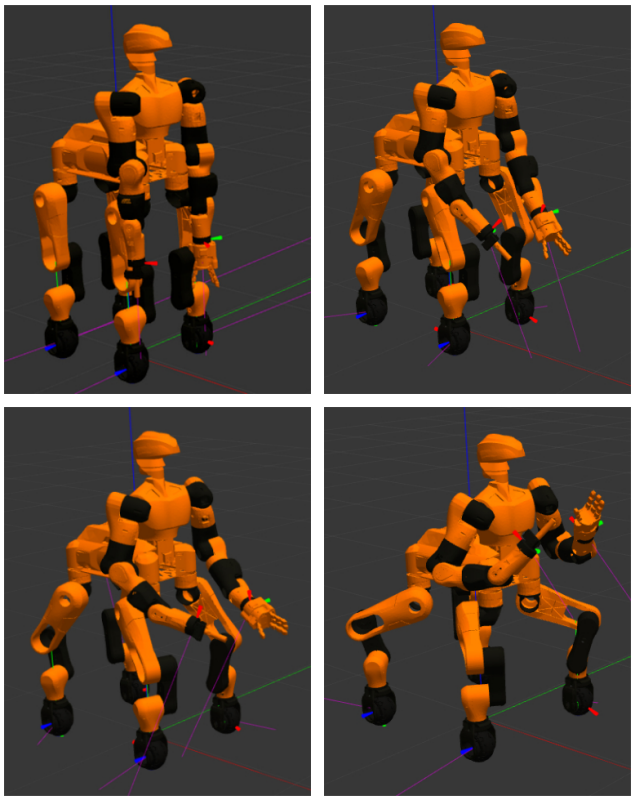

Figure 6. CENTAURO robot executing the Homing plugin inside the Gazebo simulator with the *XBot* built-in support.

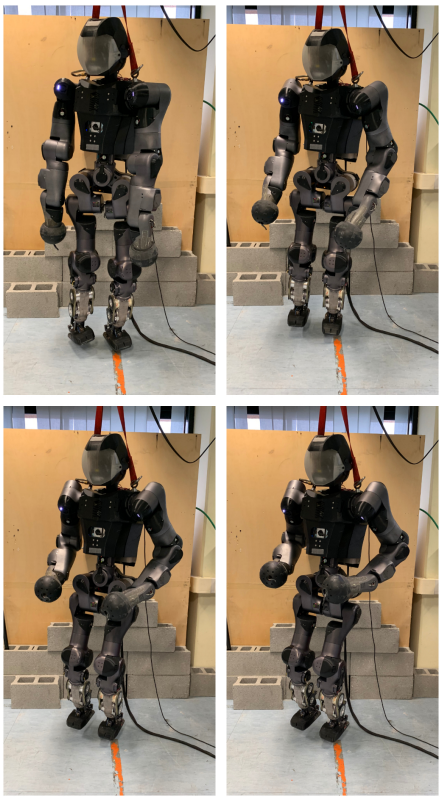

Figure 7. COMAN+ robot executing the Homing plugin.

```
set_xbot_config /opt/xbot/build/
    install/share/xbot/configs/
    CentauroConfig/centauro_basic.
    yaml
```

- start the roscore since we rely on the ROS framework for the non-RT communication in this particular example:

```
roscore
```

- start the *XBotCore* process in dummy mode (-D option) to start a kinematic simulation of the robot and load the set of plugins specified in the `centauro_basic.yaml` configuration file:

```
XBotCore -D
```

- we will now be able to use RViZ[9] to visualize the *RobotModel*[10] showing the links of the CENTAURO robot as definied in the URDF[11] specified in the `centauro_basic.yaml` configuration file, in their correct poses according to the *tf*[12] transform tree. XBotCore runs the *robot_state_publisher*[13] component internally, so that ROS nodes are able to read the robot transforms from the /tf topic. Moreover, the robot URDF is published to the ROS parameter server under the name */xbotcore/robot_description*.
- moreover thanks to the *XBot* we are also able to read the state of the joint of the robot using the custom ROS message available here `https://github.com/ADVRHumanoids/xbot_msgs/blob/master/msg/JointState.msg`, which is published by the *Communication Handler* in the topic `/xbotcore/joint_state`.
- once *XBotCore* is started and the robot is correctly visualized on *RViZ* we are able to start one of the RT plugins listed in the `centauro_basic.yaml` configuration file. For example we can move the robot to an "Homing" configuration, calling the available "switch" ROS service ready-to-use thanks to the framework:

```
rosservice call /xbotcore/
    HomingExample_switch 1
```

- the robot should now be visualized in the "Homing" configuration on RViZ. The user can decide to stop the RT plugins and start the control of the robot using a non-RT framework, by doing the following:

[9] `http://wiki.ros.org/rviz`
[10] `http://wiki.ros.org/rviz/DisplayTypes/RobotModel`
[11] `http://wiki.ros.org/urdf`
[12] `http://wiki.ros.org/tf`
[13] `http://wiki.ros.org/robot_state_publisher`

```
rosservice call /xbotcore/
    HomingExample_switch 0

rosservice call /xbotcore/
    XBotCommunicationPlugin_switch 1
```

- the `XBotCommunicationPlugin` is a special *XBot* plugin which enables the control of the robot through a non-RT master framework specified in the configuration file we set. In this basic example we rely on ROS and we defined the following custom ROS message to send reference commands to the robot, available here: `https://github.com/ADVRHumanoids/xbot_msgs/blob/master/msg/JointCommand.msg`
- An easy to run example for this feature employs either the usage of a simple ROS node (implemented in C++ or Python) to move the CENTAURO robot in the joint space using the custom joint command message described above or even a simpler command-line publish of data to the `/xbotcore/command` topic as showed below:

```
rostopic pub /xbotcore/command
    xbot_msgs/JointCommand "header:
  seq: 0
  stamp: {secs: 0, nsecs: 0}
  frame_id: ''
name: ['torso_yaw']
position: [0.5]
velocity: [0]
effort: [0]
stiffness: [0]
damping: [0]
ctrl_mode: [1]
aux_name: ''
aux: [0]"
```

During the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) 2018, a full day tutorial about the *XBot* framework was held with the title: "A Hands-on Tutorial on XBotCore: A Real-Time Cross-robot and Cross-framework Software Architecture"[14]; the reader can find all the related material online at `https://github.com/ADVRHumanoids/tutorial_iros2018`.

One of the core functionality of the *XBot* framework is the cross-robot compatibility thanks to the *XBotInterface* component which dynamically generates the robot API and the kinematic/dynamic model of it, taking as an input just the URDF, SRDF[15] and joint-ID map specified in the *XBot* configuration file. Moreover thanks to the *R-HAL* component we are able to go from the kinematic simulation mode (dummy mode above described), to the dynamic simulation mode, for example using the *Gazebo* simulation environment just adding the following plugin in the input URDF:

```
<gazebo>
    <plugin name="xbot_plugin_handler"
        filename="libGazeboXBotPlugin.so"
        />
</gazebo>
```

The above described Homing plugin can be easily run not only in dummy mode or on the real robots, but also in the Gazebo simulator, as shown in Figure 6.

At the same time we are able to go from the simulation mode (either kinematic or dynamic) to the real robot, just providing in the configuration file the low level *R-HAL* implementation to load. As an example the COMAN+ humanoid robot is capable to execute the same "Homing" plugin of the simulated "dummy" CENTAURO (Figure 7), just changing the configuration file which will now load the COMAN+ URDF, SRDF, joint-ID map and R-HAL EtherCAT implementation:

```
set_xbot_config /opt/xbot/build/install/
    share/xbot/configs/CogimonConfig/
    cogimon_basic.yaml

rosservice call /xbotcore/
    HomingExample_switch 1
```

More complex cross-robot plugins (RT or IO) are described in depth inside the tutorial repository `https://github.com/ADVRHumanoids/tutorial_iros2018/tree/master/plugins`. The rest of the online tutorial covers the whole-body inverse kinematics and dynamics tools and the related interfaces available inside the *XBotControl* framework thanks to the *OpenSoT* and the *CartesI/O*. A set of examples of the capabilities of the framework are summarized in Figure 8 .

IV. Discussion and Future Work

In this manuscript we presented the *XBot* RT software architecture. It provides to the users a software infrastructure which can be used with any robotic system enabling fast and seamless porting of the code from one robot to the other, requiring no code changes, assuring flexibility and reusability. The implementation of the framework ensures easy interoperability and built-in integration with other existing software tools for robotics, such as ROS, YARP or OROCOS. The component-based development of the *XBot* includes a Robotic Hardware Abstraction Layer (R-HAL) interface and a set of ready-to-use tools to control robots either within a simulation environment or the real robot platforms.

The framework has been successfully used an validated as a main software infrastructure (Figure 8) for humanoid robots such as WALK-MAN (result of WALK-MAN EU FP7 project[16], notably *XBotCore* received the EU innovation

[14] `https://www.iros2018.org/tutorials` `https://xbotcoretutorial.weebly.com/`

[15] `http://wiki.ros.org/srdf`

[16] `https://www.walk-man.eu/`

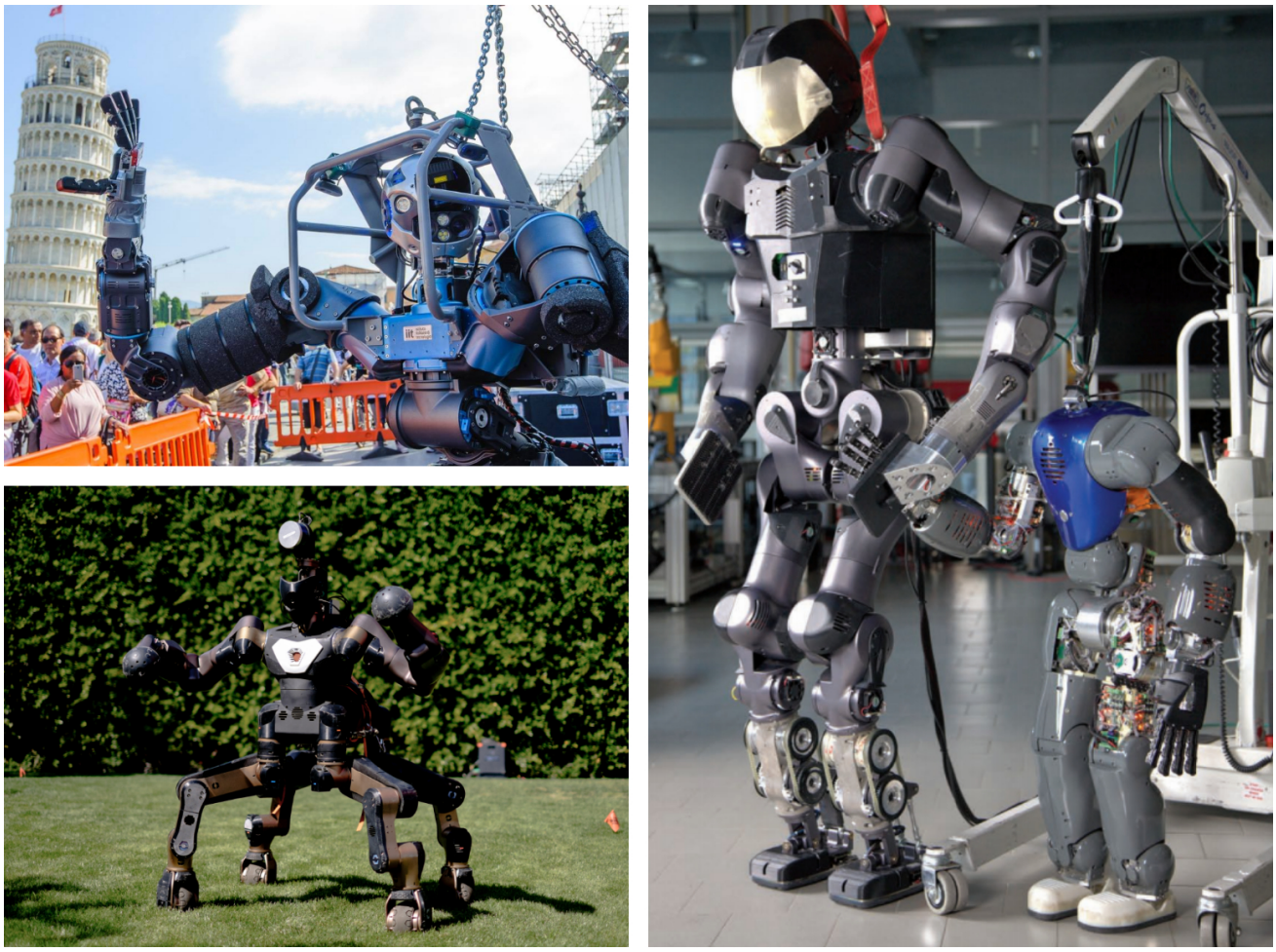

Figure 8. *XBot* framework usage examples: WALK-MAN robot in Pisa (top left), CENTAURO robot untethered and outdoor (bottom left), and the COMAN and its scaled-up version COMAN+ humanoid robots shaking hands (right).

radar award in this context[17].) and COMAN+ (result of COGIMON EU H2020 project[18]) or for quadruped centaur-like robots as CENTAURO (result of the CENTAURO EU H2020 project[19]). Moreover the cross-robot functionality has been exploited to develop both RT and non-RT control modules not only for the above mentioned robots, but also for commercial robotic systems such as KUKA LBR, KUKA 4+ or Franka Emika Panda, or other humanoid robots like COMAN or iCub.

Regarding the simulation part, *XBot* enables the direct porting of the control modules tested in the simulator to the real hardware using the same interfaces and without requiring any code modifications. The built-in simulator supported in the framework is Gazebo, but there is the option to support other simulation environments (as it happened inside the CENTAURO H2020 project with the VEROSIM simulator[20]).

[17]https://www.innoradar.eu/innovation/30632
[18]https://cogimon.eu/
[19]https://www.centauro-project.eu/
[20]https://www.verosim-solutions.com/en/

XBot became a mature and stable software and control middleware for robotics in the past few years and the Section III of the work presented herein attempted to provide both industrial and research community with a tutorial on the basic features of the framework, and how it can be used to effectively develop and integrate software and control components for robotic systems showing the flexibility and capability of the framework to work with diverse range of robot hardware.

Future works on the *XBot* will consider the support and the fusion of multiple communication links (inside the *Communication Handler*) that will enable the distribution and transmission of data to remote command and control stations based on priority classes, security and bandwidth stability. An example can be the usage and blending of cellular, WiFi and RF data channels and the distribution of data in these channels based on the quality of service available in terms of communication.

XBot currently relies on a dual-kernel approach using

Xenomai, which performs better than PREEMPT_RT[21], both in terms of system predictability and absolute latencies. Nevertheless Xenomai in the long term can introduce disadvantages by making the software development more complex, which means harder maintainability and lower portability.

Further development of the framework will target to provide synchronized distributed execution of multiple RT threads in multiple computational units. In fact currently the *Plugin Handler* is only able to execute a set of plugins in sequence, without any concurrency. This makes the maintenance of the framework easier, but restricts the performance in terms of computation power. Moreover the current architecture is characterized by a unique point of failure since both the *R-HAL* thread and the *Plugin Handler* (which executes RT plugins) thread run in the same process. In fact, there is the possibility that a misbehaving RT plugin might cause memory corruption, or crash altogether, causing also the *R-HAL* to crash. Currently only experts users are allowed to load their RT plugins in the *Plugin Handler*, but it is desirable to eventually separate the *R-HAL* and *Plugin Handler* either in two different processes or in two different machines to improve isolation.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644839 (CENTAURO), No 644727 (CogIMon) and No 779963 (EUROBENCH).

REFERENCES

[1] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and Attribute-Based bibliography," *Journal of Robotics*, vol. 2012, 7 May 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/959013

[2] G. C. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

[3] M. Aragão, P. Moreno, and A. Bernardino, "Middleware interoperability for robotics: A ros–yarp framework," *Frontiers in Robotics and AI*, vol. 3, p. 64, 2016. [Online]. Available: https://www.frontiersin.org/article/10.3389/frobt.2016.00064

[4] H. Bruyninckx, "OROCOS: design and implementation of a robot control software framework," *Proc. IEEE RAS EMBS Int. Conf. Biomed. Robot. Biomechatron.*, 2002.

[5] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-middleware: distributed component middleware for rt (robot technology)," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2005, pp. 3933–3938.

[6] L. Jeongsoo, L. Jungho, and O. Jun-Ho, "Development of robot software framework podo: Toward multi-processes and multi-users," *Workshop on software architectures and methodologies for developing humanoid robots, IEEE HUMANOIDS 2014*, 2014.

[7] J. Lim, I. Lee, I. Shim, H. Jung, H. M. Joe, H. Bae, O. Sim, J. Oh, T. Jung, S. Shin, K. Joo, M. Kim, K. Lee, Y. Bok, D.-G. Choi, B. Cho, S. Kim, J. Heo, I. Kim, J. Lee, I. S. Kwon, and J.-H. Oh, "Robot System of DRC-HUBO+ and Control Strategy of Team KAIST in DARPA Robotics Challenge Finals," *Journal of Field Robotics*, vol. 34, no. 4, pp. 802–829, Jun. 2017. [Online]. Available: http://doi.wiley.com/10.1002/rob.21673

[8] T. Houliston, J. Fountain, Y. Lin, A. Mendes, and others, "NUClear: A loosely coupled software architecture for humanoid robot systems," *Frontiers in Robotics*, 2016.

[21]PREEMPT_RT was introduced to have RT capabilities in the Linux kernel avoiding the adoption of a dual-kernel.

[9] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal on Advanced Robotics Systems*, 2006.

[10] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[11] J. Smith, D. Stephen, A. Lesman, and J. Pratt, "Real-time control of humanoid robots using openjdk," in *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '14. New York, NY, USA: ACM, 2014, pp. 29:29–29:36. [Online]. Available: http://doi.acm.org/10.1145/2661020.2661027

[12] M. Johnson, B. Shrewsbury, S. Bertrand, T. Wu, D. Duran, M. Floyd, P. Abeles, D. Stephen, N. Mertins, A. Lesman, J. Carff, W. Rifenburgh, P. Kaveti, W. Straatman, J. Smith, M. Griffioen, B. Layton, T. de Boer, T. Koolen, P. Neuhaus, and J. Pratt, "Team IHMC's lessons learned from the DARPA robotics challenge trials," *J. Field Robotics*, vol. 32, no. 2, pp. 192–208, 1 Mar. 2015.

[13] J. H. Brown, "How fast is fast enough? choosing between xenomai and linux for real-time applications," *Twelfth Real-Time Linux Workshop*, 2012.

[14] G. F. Rigano, L. Muratore, A. Laurenzi, E. M. Hoffman, and N. G. Tsagarakis, "A mixed real-time robot hardware abstraction layer (r-hal)," *Encyclopedia with Semantic Computing and Robotic Intelligence*, 2018.

[15] N. G. Tsagarakis, D. G. Caldwell, F. Negrello, W. Choi, L. Baccelliere, V. Loc, J. Noorden, L. Muratore, A. Margan, A. Cardellino *et al.*, "Walk-man: A high-performance humanoid platform for realistic environments," *Journal of Field Robotics*, vol. 34, no. 7, pp. 1225–1259, 2017.

[16] N. Kashiri, L. Baccelliere, L. Muratore, A. Laurenzi, Z. Ren, E. M. Hoffman, M. Kamedula, G. F. Rigano, J. Malzahn, S. Cordasco, P. Guria, A. Margan, and N. G. Tsagarakis, "Centauro: A hybrid locomotion and high power resilient manipulation platform," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1595–1602, 2019.

[17] N. G. Tsagarakis, S. Morfey, G. M. Cerda, L. Zhibin, and D. G. Caldwell, "Compliant humanoid coman: Optimal joint stiffness tuning for modal frequency control," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 673–678.

[18] P. Mohammadi, E. M. Hoffman, L. Muratore, N. G. Tsagarakis, and J. J. Steil, "Reactive walking based on upper-body manipulability: An application to intention detection and reaction," in *2019 International Conference on Robotics and Automation (ICRA)*, 5 2019, pp. 4991–4997.

[19] E. Mingo Hoffman, A. Rocchi, A. Laurenzi, and N. G. Tsagarakis, "Robot control for dummies: Insights and examples using opensot," in *17th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2017, Birmingham, UK, November 15-17, 2017*, 2017.

[20] A. Laurenzi, E. M. Hoffman, L. Muratore, and N. G. Tsagarakis, "Cartesi/o: A ros based real-time capable cartesian control framework," in *2019 International Conference on Robotics and Automation (ICRA)*, 5 2019, pp. 591–596.