Simplifying Operations in Locomotion - Framework for Efficient Research

Małgorzata Kamedula

A functional robotic platform is a complex synthesis of different cooperating components. Besides the robotic hardware, a one of the key elements of the successful control application is a reliable, flexible software layer. In this work, Simplifying Operations in Locomotion - Framework for Efficient Research (SOL-FER), a software developed to facilitate research in locomotion is described.

SOL-FER provides a way to quickly try out ideas by maximizing the flexibility and reconfigurability of the developed modules. It is achieved through the reconfigurable YAML files and a simple end-user API. An abstraction layers for the robot kinematics and dynamics are provided, and a communication layer that consists of the reconfigurable feedbacks/commands has been developed. This communication layer is automatically loaded on the plugin initialization. The configurable feedbacks/commands allow, for example, to run multiple cooperating plugins simultaneously or to switch between the online/offline implementation of the module without any source code modification, and thus without a need to recompile the code. Furthermore, the software provides a unified interface for the non real-time ROS and the real-time XBotCore middlewares. Thanks to the automatically generated plugins, the software can be moved from the simulation to the hardware with one parameter change in the configuration file.

A Robot Points interface has been developed to simplify a definition of the locomotion problems by providing a common interface for different robot characteristics and corresponding Jacobians. With these elements defined through the common interface, a template class has been developed to

```yaml
mapping: # load mappings
    RBDL_LOWER: # mapping name
        loading: model # use the RBDL model
        urdf: /robot_lower_body # from ROS parameter server
    RBDL_UPPER: # mapping name
        loading: model # use the RBDL model
        urdf: # from file
            path: *config
            file: mwoibn/urdf/centauro_upper_body.urdf
```

Listing 1: Example of the predefined **Bidirectional Map** from external URDF file.

menage the elements together while ensuring access to individual points if required.

This document is organized as follows: First, core components are presented in Section 1, the middleware support and algorithms implementation through the SOL-FER are discussed in Section 2, and a short summary is given in Section 3.

1 Core components

In this section, essential elements that create the SOL-FER are described. It inculdes the hardware and kineamatic abstraction layer and the mapping system described in Section 1.1 and the abstraction layer for robot dynamics presented in Section 1.2. Then, the concept of the **Robot Points** is proposed and its implementation is laid out in Section 1.3 and **Handler** is introduced in Section 1.4 that also includes practical examples of the usage of the **Robot Points**/**Handler** structure.

1.1 Robot Structure

A key component of the developed system is a **Robot** structure that contains description of the robot and its state. Furthermore, it coordinates the model and state updates. The **Robot** kinematics is read from the Unified

Robot Description Format (URDF) file, and the SOL-FER configuration file describes the robot communication layer. Additionaly, the Semantic Robot Description Format (SRDF) file can be provided to specify a robot subgroups and predefined joint-space states.

1.1.1 Robot Description

The robot is described with the rigid-body kinematic model provided by the RBDL library [1], actuation type and state, gravity vector and state size. Furthermore, mappings between the robot links, joints, state and actuators are provided.

1.1.2 Robot State

Elements of the robot state are split into two categories, the joint-space data and the **Robot Points** that describe the higher-level characteristics of the robot, see Section 1.3. The former is implemented through the **State** class that provides the position, velocity, acceleration and torques interfaces by default; more interfaces can be added to each **State** online. The **Robot** structure provides following joint-space states: the current link-side and motor-side states, the desired state and the joint-space lower/upper limits. The **Robot Points** in the **Robot** consist of the Centre of Mass (CoM), Centre of Pressure (CoP), contact points positions and reaction forces.

1.1.3 Mapping system

Integration of various components is necessary in a functional robotic framework. While the Degrees of Freedom (DoFs) order in the SOL-FER corresponds to the DoFs order in the RBDL model, the external components often adopt different conventions. To simplify integration of different elements with the SOL-FER, **Bidirectional Map** class that allows to select and sort group of DoFs between two conventions has been implemented. To improve flexibility of the developed components, a few commonly used bidirectional maps are generated automatically at the **Robot** initialization. It includes mappings between the full robot state and

- the RBDL model[1];
- the actuated DoFs;
- the default state feedback[2];
- the lower-level controller[3].

Furthermore, since the order of DoFs in the RBDL model is used as a defualt for the **SOL-FER plugins**, mappings between the robot state and different RBDL models can be initialized from the external URDF files specified in the configuration file. Code 1 shows part of the example SOL-FER configuration file that defines the external RBDL maps. This mechanism is useful to integrate **SOL-FER plugins** that work with only part of the robot (e.g. arms for manipulation plugins or lower body for the locomotion plugins). Finally, different quaternion conventions have been predefined in the SOL-FER to map the **Robot Points**. Finally, an API is provided to add the **Bidirectional Maps** online.

Besides the **Bidirectional Map**, **Unidirectional Map** is implemented to allow user to select a given subgroup of the robot DoFs (e.g. wheels, left arm). **Unidirectional Map** can be predefined in the SRDF file through the group tag or added online from within the plugin.

1.2 Dynamic Model

Dynamic Model provides an abstraction layer for the robot dynamics. It computes three components of the robot dynamics, the robot bias force, generalized inertia matrix and the inverted inertia matrix. Two dynamic models have been implemented, a full constrained robot dynamics and unconstrained dynamics computed with the QR decomposition of the constraints Jacobian.

Since computation of the robot dynamics is numerically expensive, to decrease the numerical cost of the integrated control scheme, a subscription-

[1]It is an identity map, it has been implemented to maintain the consistency in the interfaces.

[2]specific implementation depends on the middleware interface

[3]See footnote [2]

based update system has been implemented. It allows to share the robot dynamic model between the different parts of the program without unnecessary updates on the robot dynamics.

At the plugin initialization, each object that relies on the robot dynamics declares how many times per loop each component of the robot dynamics is going to be called; these declarations increase the **call counter**. The **Update Manager** counts the calls from the last model update and upon the update request, it checks if the call counter has been exhausted. As a result, it updates only elements that have been cleared since the last update; it also prevents computations of the robot dynamics components that are not used by the plugin. Thanks to this system, each component can be programmed as an independent module that updates the dynamic model without consideration given to the order of the objects in the final plugin. As a result, the components can be easily rearranged in/added to/removed from the plugin.

1.3 Robot Points

Robot Point is an interface to simplify a definition of the locomotion problems by providing a common interface for frequently used variables. It relies on the assumption that most of the control problems can be described through a vector of values and a mapping matrix between two spaces. E.g., rigid-body point can be described by the current position or tracking error and the Jacobian mapping the element from the cartesian-space to the joint-space. The basic **Robot Points** interface is given at Code 5. It provides the basic algebraic operations between the **Robot Points** and update method. Through the **Robot Points** following elements are implemented

- Rigid Point Position,
- Rigid Frame Orientation,
- Rigid Body Spatial State,
- Linear/Angular/Spatial Velocity,

- CoM,
- CoP,
- Support Polygon Vertex (SPV),
- Constant Point,
- Point and Rolling Contacts,
- Point Difference,
- Point Norm.

1.4 Handler

With all the elements defined through the **Robot Point** interface, a template class **Handler** has been implemented. It provides methods to handle **Robot Points** updates as a group; it also generates a concatenated vector of **Robot Points** values and a concatenated matrix of **Robot Points** mappings. It also ensures access to all individual **Robot Points** through the custom iterators.

Together, **Robot Points** and **Handler**, create simple, yet powerful tandem that can, in an easy, intuitive way, describe variety of robotics motion control problems. Thanks to an extensive use of templates and simple interfaces, elements implemented through **Robot Points** are reusable, and integrate naturally with the software previously developed with the **Robot Points/Handler** pattern. For example, a simple one task inverse kinematics problem comes to Code 2[4], a more complex example is given at Code 3 where a pseudo-code for implementation of an inverse kinematics problem for multiple aggravated tasks is shown. Specifically, implementation of the IK for the robot CoM tracking task with tracking of the position of the robot hands relative to the torso is shown in Code 3. Note that the update methods are the same in the two examples, and the reference task in Code 3 only adds the task reference number to support multiple tasks. The declaration

[4]The C++ pseudo-code in the examples accurately represents SOL-FER API.

in Code 3 replaces the **Robot Points** with the **Handlers**, and the **Linear Point** in the declaration method in Code 2 is replaced by the **Robot Point** in Code 3 to support variety of tasks besides position tracking of a point on a rigid-body. The basic code of the task initialization is the same; however, in Code 3 dynamic number of linear point tracking tasks is generated. Finally, note that in Code 3 the size of the '_tasks' **Handler** is bigger than the number of tasks as it also contains the support **Robot Points** for the relative tracking task; the order of the points in the '_tasks' **Handler** ensures that all lower-level **Robot Points** are updated before the higher-level, dependent ones. This chaining capability of the **Handler** update method allows to further expand the reusability of the implemented **Robot Points**. Code 4 provides a pseudo-code for the program that computes the norm between the robot hands and the robot reference point. The initialization of this computation takes 7 lines of code: first 3 lines declare the hand and base points in the world frame, the 2 middle lines create the relative hand position and the final 2 lines compute the norm for each relative vector. The last two lines in the init method move the reference point 15 cm away from the origin of the left/right hand frame, respectively. Computation of the norms in the main program loop takes one line – the **Handler** update. Code 4 highlights reusablity and efficacy of the **Robot Points**/**Handler** pattern in a one more manner. The position of the robot reference point is computed only once per loop, even though both distances are computed with respect to this reference point.

1.4.1 Dynamic Points

The **Robot Point** interface well fits the robot first-order kinematics that is described by the linear equations in the form $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$. However, it is not sufficient to describe the second-order kinematic and dynamic problems that read $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$. To support the higher-order robot description, the child **Dynamic Point** interface to the **Robot Points** interface has been introduced. **Dynamic Point** has an additional offset member; and it provides an interface for the following elements

```
protected: // variables declaration
       // Robot Point that computes the position and Jacobian
       ↪  of the point on the rigid-body
       LinearPoint _task;
       // Constant Robot Point that implements the 'set'
       ↪  method to set the state by hand
       Constant _reference;
       // Minus Robot Point that subtracts two Robot Points
       Minus _ik;
public:
// memory allocation, initialize inverse kinematics for the
 ↪  end-effector tracking
function init(Robot& robot, string pointName =
 ↪  "EndEffectorName"){
       // create the end-effector position point
       _task = LinearPoint(pointName, robot);

       // create a reference point of a size to the task
       ↪  '_task'
       RobotPoint _reference = Constant(_task.size());
       // set the reference jacobian to zero for the Minus
       ↪  robot point update method
       _reference.set(Matrix::Zero(_reference.rows(),
       ↪  _reference.cols()));

       // initialize the tracking error computation
       RobotPoint _ik = Minus(_task, _reference);
}

function update(){ // control loop
      // update current end-effector state
      _task.update(true); // true means update jacobian

      // update an inverse kinematics
      _ik.update(); // jacobian_update=true
      _q = inverse(_ik.getJacobian())*_ik.getState();
}

      // update the reference
function reference(const Vector& x_des){
 ↪  _reference.set(x_des);}
```

Listing 2: Pseudo-code for the **Robot Points**, **Handler** implementation of the IK scheme for the point position tracking, with example initialization for and user specified end-effector.

```
protected: // variables declaration
    // Handler of Robot Points to support any IK task
    Handler<RobotPoint> _tasks;
    // Handler of references, it is specialized for Constant
     ↪  Robot Point to have an access to the set method
    Handler<Constant> _reference;
    // Handler of the Minus points to compute the tracking
     ↪  errors
    Handler<Minus> _ik;
public:
// memory allocation, initialize inverse kinematics for a
 ↪  given Robot, and requested end-effectors
function init(Robot& robot, str points="hands", str
 ↪  root="torso"){
// initialize the COM tracking task with the COM embedded in
 ↪  the Robot
    _tasks.add(robot.COM());
    // initialize reference and task
    _reference.add(Robot.COM.size());
    _ik.add(Minus(_tasks[-1],_reference[-1]);
    // add the Linear Point of torso
    _tasks.add(LinearPoint(root));
    // functionined the task for each point in the
     ↪  Unidirectional Map requested specified the the user
    for (auto& point: Robot.getLinks(points)){
        // add Linear Points
        _tasks.add(LinearPoint(point));
        // compute a state of the relative point
        _tasks.add(Minus(_tasks[-1], _tasks[0]));
        // initialize reference and task
        _reference.add(Constant(_tasks[-1].size()));
        _ik.add(Minus(_tasks[-1],_reference[-1]));}
    // init the reference Jacobians
    for (auto& point: _reference)
     ↪  point.set(Matrix::Zero(point.rows(), point.cols()));}
function update(){ // control loop
    // update all the support points
    _tasks.update(true); // jacobian_update=true
    // update an inverse kinematics
    _ik.update(true); //  jacobian_update=true
    _q = inverse(_ik.getJacobian())*_ik.getState();}
function reference(Vector& x_des, int i){//reference update in
 ↪  task i
        _reference[i].set(x_des);} // access Constant set
         ↪  method
```


Listing 3: Pseudo-code for the **Robot Points**, **Handler** implementation of the IK scheme for multiple aggregated tasks. Example for the IK: CoM tracking and the position of the 'n' hands with respect to the torso.

```
    protected: // variables declaration
        // Handler of Robot Points to keep all the Robot
        ↪ Points
        Handler<RobotPoint> _points;

    public:
    // memory allocation, create the Norm Points
    function init(){
        // create the basic Linear Points for the rigid body
        ↪ points
        _points.add(LinearPoint("LeftHand"));
        _points.add(LinearPoint("RightHand"));
        _points.add(LinearPoint("Base"));
        // create the point that computes the difference
        ↪ between the "LeftHand" and "Base"
        _points.add(Minus(_points[0],_points[2]);
        // create the point that computes the difference
        ↪ between the "RightHand" and "Base"
        _points.add(Minus(_points[1],_points[2]);
        // create the point that computes norm of the
        ↪ penultimate point in the Handler
        _points.add(Norm(_points.end(1));
        // create the point that computes norm of the
        ↪ penultimate point in the Handler
        _points.add(Norm(_points.end(1));
        // set an offset between the "LeftHand" frame origin
        ↪ and the tracked point given in the "LeftHand"
        ↪ frame
        _points[0].point.position.setFixed(Vector(0,0,-0.15));
        // set an offset between the "RightHand" frame origin
        ↪ and the tracked point given in the "RightHand"
        ↪ frame
        _points[1].point.position.setFixed(Vector(0,0,-0.15));
    }

    function update(){ // main loop
        // compute the norms
        _points.update(false); // do not update the jacobians
        // print the results
        std::cout << "Right Hand distance from the Base is " <<
        ↪ _points.end(1).getState().transpose() << "." <<
        ↪ std::endl;
        std::cout << "Left Hand distance from the Base is " <<
        ↪ _points.end(0).getState().transpose() << "." <<
        ↪ std::endl;
        }
```

Listing 4: Pseudo-code to compute the distances between the end-effectors and the reference point when the tracked points do not coincide with the

```
    virtual void compute() = 0;
    virtual void computeJacobian() = 0;

    virtual void update(bool jacobian = true);

    const mwoibn::Matrix& jacobian() const;
    const mwoibn::VectorN& point() const;

    int size();
    int rows();
    int cols();

    Point& operator=(const Point& other);
    Point& operator+(const Point& other);
    Point& operator-(const Point& other);
    Point& operator+=(const Point& other);
    Point& operator-=(const Point& other);
```

Listing 5: The Robot Points Interface.

- linear, angular and spatial acceleration for the rigid-point/body,
- point/body force, torque and wrench,
- SPV acceleration.

2 Plugins

The SOL-FER adopts a plugin system that exploits two software layers to provide the support for different middlewares. The first layer consists of the **components** that are middleware independent implementations of specific algorithms/programs. Each component has to inherit from the **Base** class and adhere to the provided interface as shown at Code 15. The second layer consists of **SOL-FER plugins** that provide the communication layer, other middleware dependent components of the program and coordinate the kinematic updates of the **Robot** instance. Dynamically loaded plugins for all supported middlewares are auto-generated from the plugin template spe-

cialization that has to be created for each **Component**. An example of the template specialization for the joint-space controller module is given at Code 16.

Remainder of this section is organized as follows: First, the SOL-FER configuarition files are described in Section 2.1, then a few exmples that highlight felxibility of the developed **Plugin/Component** structure are shown in Section 2.2, and the **shared plugin** is introduced in Section 2.3.

2.1 Configuration File

Plugin loads the **Robot** instance based on the URDF/SRDF files defined in the configuration file and the communication layer. The communication layer depends on the middleware, and, after ROS, adopts a strict publisher-subscribes pattern. Each connection between two plugins is refereed to as a **pipe**.

2.1.1 Library of pipes

The communication layer for the robot current/desired **States** is specified in the configuration file. To that end, a configuration for each **pipe** has to be given in the configuration file creating a library of predefined **pipes** the plugins can load from. A **pipe** configuration consists of: a type of information a **pipe** transfers, affected DoFs, **State** interfaces the **pipe** publishes for/subscribes to, robot **State** the **pipe** should be attached to and the **Bidirectional Map** that should be used for mapping.

One of two information types can be passed through the auto-generated **pipe**: joint-space values (see Code 6 for an example configuration) or the state of the robot rigid-body (see Code 7 for an example configuration). Other types of **pipes** are not supported for the auto-generation. If plugin specific higher-level communication not supported through the configuration file is required, it can be defined in the plugin template specialization **_initCallbacks** method (see Code 16).

The affected DoFs are described by providing the information which robot chain should be considered (all, SRDF group tags) and what type of DoFs is

```yaml
link_side_online: # name of the communicating pipe
    space: JOINT # pipe space
    dofs: # which DoFs are considered
        chain: all # which part of the robot should be used
        type: actuated # which type of DoFs is considered
        mapping: PYTHON # which mapping should be used
    interface: # which interfaces are active
        position: true
        velocity: true
        effort: true
    function: state # which State it is attached to
```

Listing 6: Example of the **pipe** configuration in the joint-space.

affected (e.g. all, actuated, body, **Bidirectional Map** name). If the body type is used, a link name has to be provided. Finally, the 'function' tag describes **State** a **pipe** is attached to where options are 'state' for the robot current state and 'reference' for the robot desired state. When the **pipe** provides a state of the rigid-body, a convention used to describe it has to be specified (Code 7).

Finally, a middleware specific configuration is required (see Code 8). For example, ROS requires type and topic of a message, and – due to the callback nature of the ROS communication layer – an information if given pipe has to be initialised before the component initial conditions can be set. On the other hand, XBotCore needs the software layer the pipe communicates with and the **pipe** name. Some more specific tags may be required depending on the **pipe** loaded; Table 1 and Table 2 provide overview over implemented **pipes** that can be auto-generated for ROS and XBotCore, respectively.

2.1.2 Plugin Configuration

Plugins are loaded based on the configuration specified in the SOL-FER configuration file; a minimal example is given at Code 9. Each plugin configuration is identified by a unique name ('joint_online' in the example). For XBotCore plugins this name has to correspond with identifier hardcoded in a template specialization required for each component, as the middleware does

```yaml
floating_base:
        space: OPERATIONAL
        convention:
                orientation:
                        type: QUATERNION
                        convention: HAMILTONIAN
                position:
                        type: FULL
        dofs:
                chain: all
                type: body
                name: pelvis
                mapping: RBDL
        interface:
                position: true
                velocity: false
                effort: false
```

Listing 7: Example of the **pipe** configuration for the robot rigid-body.

```yaml
ros:
    feedback:
        whole_body:
            source: desired_state
            message: custom_messages::CustomCmnd
            initialize: false
    controller:
        whole_body:
            sink: desired_state
xbot:
    feedback:
        whole_body:
            layer: NRT
            source: desired_state
    controller:
        whole_body:
            layer: NRT
            sink: desired_state
```

Listing 8: Example of the middleware specific **pipe** configuration.

ROS

type	input	output	comments
join-space	NRT	lower-level	ros_control, impedance controller
join-space	NRT	lower-level	ros_control, velocity controller
join-space	lower-level	NRT	ros_control, state feedback
rigid-body	lower-level	NRT	simulated floating base state
join-space	lower-level	NRT	simulated contact forces
rigid-body	NRT	NRT	pass **Robot Point** state
joint-space	NRT	NRT	pass joint-space state in ROS

Table 1: Overview over ROS pipes implemented for the auto-generation in the SOL-FER plugin.

XBotCore

type	input	output	comments
join-space	RT	lower-level	desired state
join-space	lower-level	RT	state feedback
rigid-body	lower-level	RT	imu readings
joint-space	RT	RT	shared memory
rigid-body	RT	RT	**Robot Point** state, shared memory
rigid-body	RT	NRT	rigid-body state
joint-space	NRT	RT	from ROS topic
joint-space	RT	NRT	publish to ROS topic

Table 2: Overview over XBotCore pipes implemented for the auto-generation in the SOL-FER plugin.

not provide a solution to pass additional arguments on the plugin start-up. In ROS, by default the same configuration corresponding to the hardcoded plugin identifier is loaded. However, the user can load a plugin with any configuration by passing an optional command line argument 'n' when launching the plugin. The command to run the ROS plugins read

```
rosrun plugins plugin -p plugin_identifier -n config_name
```

A plugin configuration has to specify a layer the plugin operates in what determines types of **pipes** loaded and a secondary configuration file[5] that describes parameters dependent on the software layer. These consist of control loop frequency and filters tuning. A plugin configuration also specifies a plugin mode where, the options are 'full' for a plugin working as designed and 'idle' for a plugin working without sending desired commands to the lower-level. In the 'idle' mode plugin still receives the system state, but it does not control the robot. The 'idle' mode is useful to collect the data, debug the code with the real robot state or to check a plugin for the memory allocations. A plugin configuration also contains declarations if plugin relies on the robot kinematics/dynamics, and if it modifies the robot state internally. These flags are used to coordinate updates on **Robot** instances. Finally, the 'controller' and 'robot' parameters define the plugin communication layer. The 'controller' tag consists of list of **pipes** the plugin publishes to, or a name of a predefined set of pipes that can be declared in the 'controllers' tag, see Code 11. The 'robot' tag specifies one of predefined configurations to be loaded. These 'robot' configurations describe the list of subscribers the **Robot** receives the feedbacks from, and if the robot actuation models and contacts are required by the plugin. See Code 10 for example of the online and offline 'robot' configurations. Similarly to the publishers list in the 'controller' tag, the lists of subscribers can also be predefined in the 'feedbacks' tag, see Code 11. In the end, the configuration for each plugin is passed to the **Component** constructor, and so any custom tags can be given to tune/set-up the **Component**.

[5]See Section 2.1.3 for details on the secondary configuration files.

```yaml
modules:
        joint_online: # plugin name/configuration id
            layer: NRT # layer the plugins runs at
            robot: joint_space # which robot configuration
             ↪ should be loaded
            controller: direct # which controllers should be
             ↪ loaded
            mode: full # if 'full' all controllers are loaded,
             ↪ if 'idle' the desired states are not send to
             ↪ the lower-level controller
            kinematics: false # does it require kinematic
             ↪ update
            dynamics: false # does it require dynamic update
            model_change: false # does it modify the robot
             ↪ state internally
```

Listing 9: Example of the plugin configuration for the joint-space controller.

```yaml
robot: # robot configurations
        online: # configuration id
                feedback: # which feedbacks should be loaded
                        layer: online # read a predefined
                         ↪ set-up
                actuators: # if the actuation model should be
                 ↪ loaded
                        read: true
                contacts: # does it need contacts
                        read: true
        offline:
                feedback:
                        list: [whole_body, reference] # read
                         ↪ given list
                actuators:
                        read: false
                contacts:
                        read: true
```

Listing 10: Example of the predefined **Robot** configurations for the online and offline plugins.

```yaml
feedbacks: # predefined feedback set-ups
        online: [link_side_online, odometry, reference]
        offline: [whole_body]
controllers: # predefined controller set-ups
        direct:
                list: [position_controller,
                ↪  velocity_controller]
```

Listing 11: Example of the predefined pipeline set-ups.

```yaml
config_name:
        secondary_file: # if present, use secondary file
            path: *support # specify path to the file
            file: upper_body.yaml # file name with file
             ↪  extension
```

Listing 12: Example of the predefined pipeline set-ups.

2.1.3 Secondary Configuration Files

With an extensive use of the libraries of the predefined elements in the configuration files – the library of pipes, publishers/subscribers layers and the **Robot** configurations – range of the different plugins arrangements can be achieved with a minimal change in the configuration file. However, often only a small change in a given predefined configuration is required for a desired configuration. Rather than maintaining multiple full configuration files with a one/two lines difference, a secondary configuration files have been introduced. The secondary files are merged with the full configuration file with the priority given to the secondary file tags when the two overlap. Thus, the secondary configuration files allow to overwrite any element of the original configuration file and add tags not present in the original file. Secondary files are specified for each plugin independently by an optional 'secondary_file' tag in the plugin configuration, see Code 12.

Two sets of secondary files are used by default in the SOL-FER. These files load the configuration specific for the software layer the plugin operates in (control loop frequency) and the configuration specific for the hard-

```yaml
mwoibn:
        ros: # cheange the urdf to the upper-body
                source:
                        urdf: "/robot_upper_body"
        xbot:
                source:
                        urdf:
                                file:
                                 ↪  mwoibn/urdf/centauro_upper_body.
        feedbacks:
                online: [link_side_online] # remove the state
                 ↪  estimation from the deafult feedbacks,
                 ↪  upper-body is a static model
```

Listing 13: An example secondary configuration file for the manipulation plugin.

ware/simulation (sensors calibration). These files can be expanded to add plugin configuration specific for the software layer or the hardware/simulation (e.g. filter tunning).

Furthermore, the secondary files simplify integration of the SOL-FER with the XBotCore. This mechanism allows to add only one parameter to the XBotCore configuration file – a path to the SOL-FER configuration file – to run multiple XBotCore plugins. Otherwise, starting multiple XBotCore plugins with contradicting parameters would require specifying and maintaining a SOL-FER configuration file for each XBotCore plugin independently.

Code 13 gives a full secondary configuration file for the manipulation plugin. This file changes the robot URDF to the fixed base upper-body model, and it removes the state estimation from the default feedback. Another example is given at Code 14 where the secondary file for the odometry pulgin is given. In this file, the default whole-body robot URDF is changed to the lower-body model, and the default position controller configuration is modified to control only the robot floating base, i.e. only unactuated DoFs in the 'base' chain.

```yaml
mwoibn:
        controller:
                position_controller: # control only the robot lower-body
                        dofs:
                                chain: base
                                type: unactuated
                                mapping: PYTHON
        ros: # change the robot urdf source to the lower-body urdf
                source:
                       urdf: "/robot_lower_body"
        xbot: # change the robot urdf source to the lower-body urdf
                source:
                        urdf:
                                file: "mwoibn/urdf/centauro_lower_body.urdf"
```

Listing 14: An example secondary configuration file for the odometry plugin.

2.2 Examples

Fig. 1 visualises the two layers **Plugin**–**Component** structure, and Fig. 2 presents an example of the manipulation plugin controlling the robot upper-body where the mapping between the upper-body and the lower-level controller is generated automatically in the plugin layer. The automatic map generation based on the robot kinematics specified in the configuration file means the same plugin can be used to control different parts of the robot without any modifications in the **Component**/**Plugin** source code (e.g. gravity compensation, joint-space control, IK, Inverse Dynamics (ID)). Furthermore, definition of the communication layer through the configuration files allows to switch between the online and offline plugin implementation without any modification of the source code, see Fig. 3 for a concept scheme, and Code 10 for the corresponding robot/feedbacks configurations. The 'reference' and 'whole-body' feedbacks in Code 10 are both position/velocity feedbacks of the full robot state, where the former subscribes to the robot current state

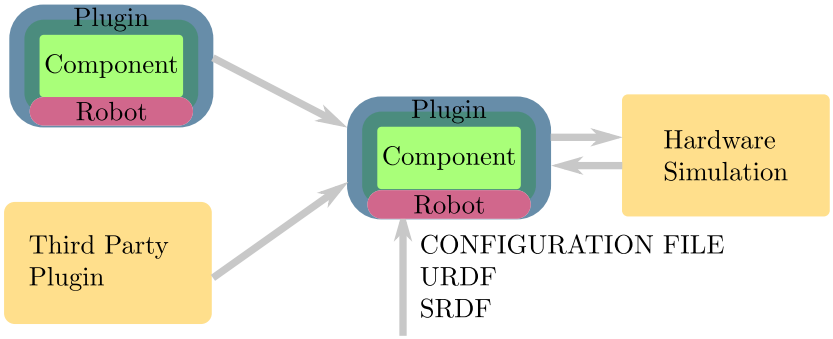

Figure 1: Scheme that visualises the **Plugin**–**Component** structure and the **Plugin** interaction with the middleware. Pink areas show loaded **Robot** configuration, blue areas indicate the **Plugin** space which green interiors represent the **Component** space. Yellow areas symbolise the third party elements.

(function: state in the **pipe** configuration) and the latter subscribes to the desired state (function: reference in the **pipe** configuration).

2.2.1 Shared control

For a complex robots with multiple kinematic chains, only part of the robot structure may be required to solve a given task. In these cases, to speed-up the computations, a simplified, partial robot model is typically used. Thanks to the flexible communication layer and the mapping system (see Section 1.1.3), multiple plugins that control specific parts of the robot structure can cooperate without interference. Fig. 4 shows an example of the locomotion and manipulation plugins working simultaneously controlling the robot upper-body and lower-body structures, respectively. In this example, the gravity compensation plugin that operates on the whole-body model maps partial – lower-body and upper-body – solutions to the full robot state and sends the whole-body command to the firmware. Note, that in this example any component[6] that loads the whole-body model could be used to merge the partial commands.[7] That is because the pipes are combined at the

[6]including an empty component that does not implement any algorithms

[7]To be more specific, a full whole-body model is not a requirement. Any model that includes all DoFs controlled by other plugins is sufficient. Otherwise, the DoFs not represented in the final model would not be parsed to the lower-level.

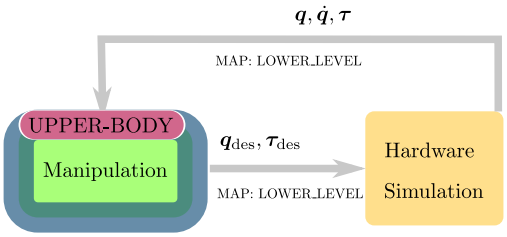

Figure 2: An example of the plugin communicating with the lower-level control. Pink area marks loaded **Robot** configuration, blue area indicates the **Plugin** space which green interior represents **Component** space. Yellow area symbolises the third party element - hardware/simulator.

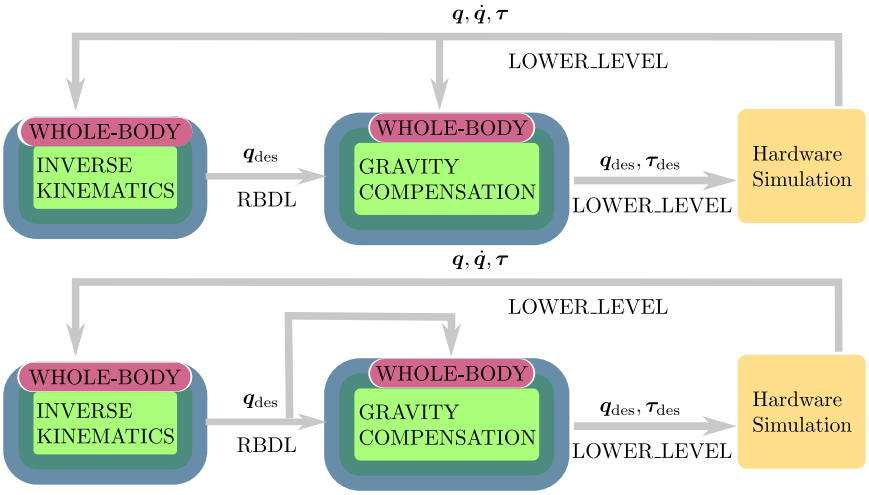

Figure 3: An example of the configuration for the online/offline plugin configuration. Pink areas mark loaded **Robot** configurations, blue areas indicate the **Plugin** space which green interiors represent **Component** space. Yellow areas symbolise the third party elements - hardware/simulator.

```
public:
Base(mwoibn::robot_class::Robot& robot);

virtual ~Base() {}

virtual void init() = 0; // runs at the robot initalization
virtual void update() = 0; // runs at each update loop
virtual void send() = 0; // runs the outputs communication
    ↪ layer
virtual void stop() = 0; // runs when the module is stoped
virtual void close() = 0; // runs on the module shut down
virtual void setRate(double rate); // update the module update
    ↪ frequency

virtual void startLog(mwoibn::common::Logger& logger); //
    ↪ initialize the logger
virtual void log(mwoibn::common::Logger& logger, double time) =
    ↪ 0; // write the log loop

mwoibn::robot_class::Robot& model(){return _robot;}; //
    ↪ provides acces to the Robot object

mwoibn::common::Flag kinematics; // does the kinematic update
    ↪ is required?
mwoibn::common::Flag dynamics; // does the dynamic update is
    ↪ required?
mwoibn::common::Flag modify; // does the module modifes the
    ↪ robot current state?

const std::string& name(){return _name;} // name of the module
```

Listing 15: Public interface of the module interface.

```cpp
#include "mgnss/plugins/generator.h" // the plugin template
#include "mgnss/ros_callbacks/joint_states.h" // the higher-level communication
#include "mgnss/controllers/joint_states.h" // component - joint-space controller
#include <custom_services/jointStateCmnd.h> // ros topic

template<typename Subscriber, typename Service, typename Node, typename Publisher>
class JointStates : public mgnss::plugins::Generator<Subscriber, Service, Node, Publisher>
{
  typedef mgnss::plugins::Generator<Subscriber, Service, Node, Publisher> Generator_;

public:
    JointStates() : Generator_("joint_states"){ }

    virtual ~JointStates(){}

protected:

// specify component to be loaded
virtual void _resetPrt(YAML::Node config){
    Generator_::controller_ptr.reset(new mgnss::controllers::JointStates(
    *Generator_::_robot_ptr.begin()->second, config));
}

// initialize the higher-level communication
virtual void _initCallbacks(YAML::Node config){
    Generator_::_srv.push_back(Generator_::n->template advertiseService<custom_services::jointStateCmnd::Request, custom_services::jointStateCmnd::Response>(
    Generator_::controller_ptr->name() + "/trajectory",
    boost::bind(
    &mgnss::ros_callbacks::joint_states::referenceHandler,_1, _2, static_cast<mgnss::controllers::JointStates*>(
    Generator_::controller_ptr.get()))));

}
};
```


Listing 16: Plugin template specialization for the joint-space controller **Component**.

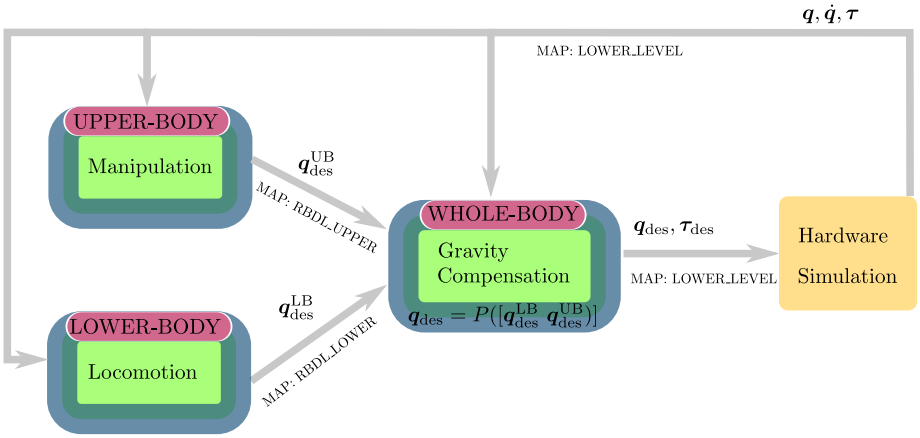

Figure 4: An example of the shared control structure. Pink areas mark loaded **Robot** configurations, blue areas indicate the **Plugin** space which green interiors represent the **Component** space. $P()$ represents the permutation operation, and yellow area symbolises a third party element - hardware/simulator.

auto-generated plugin layer. However, although not necessary, the merging component is highly recommended to – for safety reasons – have only one plugin that sends commands directly to the lower-level. Depending on the communication layer design in the underlying middleware two plugins sending partial state commands to the lower-level may or may not be supported.

2.3 Shared plugin

With the basic ROS/XBotCore plugins, elements share the states through the **pipes**, but each plugin runs as an independent, self-sufficient program. In practice, when running multiple plugins, the same robot model/feedback configuration is often used multiple times. In such cases, number of updates on the robot kinematics/dynamics could be reduced if the plugins would share the same **Robot** instance. To that end, a **shared plugin**, for both ROS and XBotCore middlewares, have been developed to coordinate allocation and updates of the **Robot** instances for the **Components**. It implements a mechanism to automatically detect that multiple plugins configurations require compatible **Robot** instances. It automatically creates correct **Robot**

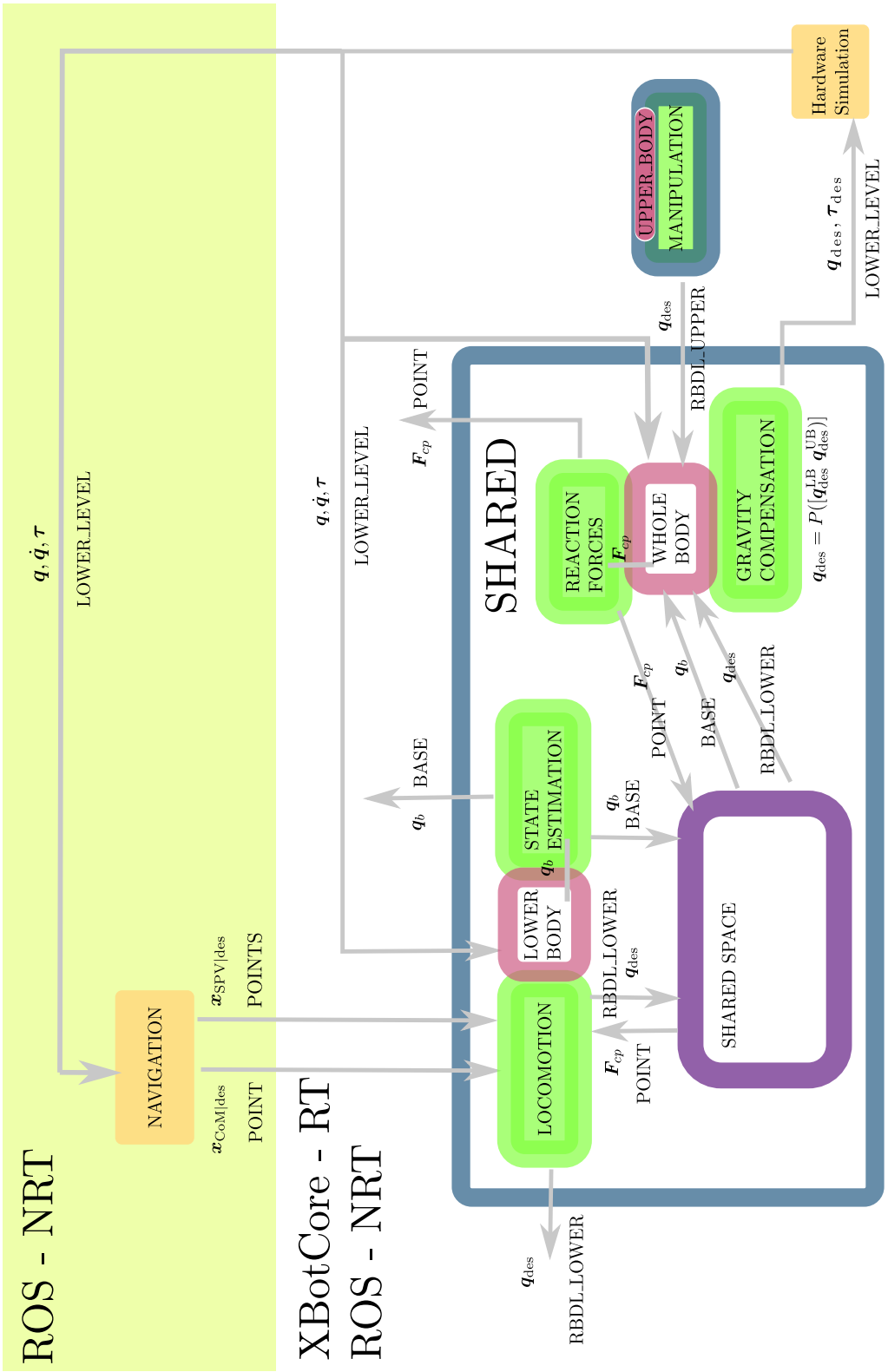

Figure 5: An example of the full control scheme implemented with the SOLFER including cooperation between the RT and NRT layers. Pink areas show loaded **Robot** configuration, blue areas indicate the **Plugin** space, and green areas represent the **Component** space. Yellow areas symbolise the third party elements, and purple area denotes the **shared plugin** shared space.

```
shared:
    layer: NRT
    plugins: [UpperBodyIK::upper_body_ik, odometry3::odometry,
     ↪ ground_forces, gravity_compensation,
     ↪ NwheelsZMPII::wheeled_motion]
```

Listing 17: A configuration of the **shared plugin** to load 5 **Components/Plugins**.

instances and provides the **Components** with references to them.

Shared plugin also provide a shared space – a middleware independent communication layer between the **Components** that allows to share the information immediately without relying on the middleware communication mechanism that may introduce delays. To that end, and new types of **pipes** for the plugins to communicate the joint-space and **Robot Points** states through the shared space have been developed. These shared **pipes** are created automatically when the **pipe** publisher is created to communicate with the middleware, i.e. all information broadcasted outside the **shared plugin** is also broadcasted to the shared space. When the subscriber is created for the **Component** inside the **shared plugin**, priority is given to the shared space **pipes**; the middleware **pipes** are generated if no matching shared publisher exists. Other, custom shared **pipes** can be initialized in the **_initCallbacks** method of the plugin template. Finally, the **shared plugins** provides a mechanism in XBotCore to load the same **Plugin/Component** with different configurations.

The **shared plugin** configuration does not adhere to the minimum plugin configuration described in Section 2.1.2. The only required tags are 'layer' and a list of plugins to be loaded. Custom plugin configuration may be specified using the double colon as a separator; otherwise, a default configuration is used. Code 17 shows a **shared plugin** configuration to load 5 different plugins. Note that the same plugin can be loaded multiple times simultaneously with different, or even the same, configurations.

Fig. 5 shows an example control scheme built from the **shared plugin** cooperating with a SOL-FER **plugin**, a higher-level third party component

and a hardware/simulation. In the **shared plugin**, four basic **plugins** cooperate: the state estimation and the whole-body locomotion control use the same lower-body robot model, and the estimation of the reaction forces and the gravity compensation share the full whole-body model.

On **shared plugin** update, the **Robot** lower-body kinematics is updated, and the state estimation **Component** computes the estimated robot floating base position. The state estimation plugin updates the state of the lower-body **Robot** it is attached to, sends the estimated floating base position to the shared space, and to the middleware. Then, the whole-body **Robot** floating base state is read from the shared space and the kinematic and dynamic models are updated. The next step consists of update of the reaction forces **Component** that estimates and updates the reaction forces in the whole-body **Robot**, sends the estimated reaction forces to the shared space and the middleware space. Next, the lower-body **Robot** kinematics is updated again, since the state estimation **Component** modified its current state internally. Moreover, the estimated reaction forces are read from the shared space and updated in the lower-body **Robot**. Then, the locomotion **Component** runs and computes the lower-body joint-space commands to track the desired CoM and SPV positions received from the third party navigation plugin operating in the non-real-time ROS space. The computed joint-space command is sent to the shared-space and the middleware space. Finally, the gravity compensation **Component** is computed. Since, the whole-body **Robot** state has not been changed since the last update, there is no update of the whole-body **Robot** kinematics. The desired lower-body command is read from the shared space, and the desired upper-body command is read from the middleware. The whole-body gravity compensation is computed based on the contact points positions, and whole-body position, velocity and torque commands are send to the lower-level simulation/hardware. The upper-body reference is defined by the external SOL-FER manipulation plugin that operates on the static upper-body model. In XBotCore, the manipulation plugin is ensured to be updated before the **shared plugin**. Otherwise, since in ROS the order of the plugins updates cannot be secured, the last known desired upper-body joint-space command is used. In the **shared plugin**, order of

the **Components** updates is always ensured.

These control scheme has been tested in non-real-time ROS with CENTAURO robot simulation and in real-time XBotCore with CENTAURO robot hardware. In real-time implementation the higher-level locomotion CoM and SPV commands have been send from the non-real-time ROS layer.

3 Summary

In this document, Simplifying Operations in Locomotion - Framework for Efficient Research (SOL-FER), a software for the locomotion research has been introduced. This framework has been designed to provide a way to quickly try out ideas by maximizing the flexibility and reconfigurability of the developed modules. An abstraction layers for the robot kinematics and dynamics have been introduced, and the communication layer that consists of the reconfigurable feedbacks/commands has been described. These reconfigurable feedbacks/commands allow, for example, to run multiple cooperating plugins simultaneously or to switch between the online and offline implementations of a module without any source code modification, and thus without a need to recompile the code. Furthermore, the **shared plugin** that decreases the computational burdain of the multiple cooperating plugins by sharing the same robot abstraction layers have been proposed. Finally, the **Robot Points/Handler** structure have been introduced to simplify the implementation of the locomotion control schemes.

References

[1] Martin L. Felis. "RBDL: an efficient rigid-body dynamics library using recursive algorithms". In: *Autonomous Robots* (2016), pp. 1–17.