Ctrl-MORE: A Framework to Integrate Controllers of Multi-DoF Robot for Developers and Users

Juan A. Castano, Przemyslaw Kryczka, Brian Delhaisse, Chengxu Zhou and Nikos G. Tsagarakis

Abstract— **In recent years, many different feedback controllers for robotic applications have been proposed and implemented. However, the high coupling between the different software modules made their integration into one common architecture difficult. Consequently, this has hindered the ability of a user to employ the different controllers into a single, general and modular framework.**

To address this problem, we present Ctrl-MORE, a software architecture developed to fill the gap between control developers and other users in robotic applications. On one hand, Ctrl-MORE aims to provide developers with an opportunity to integrate easily and share their controllers with other roboticists working in different areas. For example, manipulation, locomotion, vision and so on. On the other hand, it provides to end-users a tool to apply the additional control strategies that guarantee the execution of desired behaviors in a transparent, yet efficient way. The proposed control architecture allows an easier integration of general purpose feedback controllers, such as stabilizers, with higher control layers such as trajectory planners, increasing the robustness of the overall system.

I. Introduction

Nowadays, many cross-platform software solutions have arisen to provide general purpose tools and accelerate the research in different robotic applications. The purpose of those programs is to provide a structure that can be used by a wide range of roboticist, to integrate their works on the considered hardware. To reach this aim, software developers have worked in different layers of abstraction to facilitate the technological development in robotics.

One of the first abstraction levels is the Hardware Layer. The goal of this abstraction is to hide the hardware complexity from task designers and high level users of the robot [1]. This layer permits to write software solutions that are not robot dependent; since the hardware is transparent to other software layers, such as perception and locomotion [2].

At a different abstraction level we find YARP [3] and ROS [4] which are tools that facilitate the communication between different modules and data sources. These frameworks act as a bridge between different data sources and merge them into a single decentralized application. As result, it is possible to handle information from different sensors, control platforms, and different modules and combine them into single platforms in a transparent and general way.

These software abstractions hide the hardware layer from software developers on other layers. With such abstractions, end users do not need to consider hardware particularities and signal sources. They can use the robots' signals directly, facilitating code development and integration in a centralized application. However, many developments are made as decentralized applications due to the expertise of researchers and the integration of different programs and libraries. This causes difficulty when integrating all of them in a single centralized application.

To integrate specific modules such as manipulation, locomotion, planning, etc. and switch between them, there is a cross-robot software called *XBotCore software platform* [5] which has been implemented on the humanoid robots developed in the authors lab such as COMAN [6] and Walk-MAN [7]. This work is a standard control system that is modular, flexible, and robot independent. It has been designed to centralize the robot's information, and reuse the code between modules and platforms. This work gives the user and interface to develop, integrate, and switch modules in a single application which is transparent and user friendly.

Another architecture to handle the robot's software modules is given in [8]. In this work, the authors present an architecture that provides to the operator control of the robot in different situations. It allows to switch between existing modules, and have explicit control of the robot when necessary. When using this architecture, the operator can control the robot through teleoperation, where the human may command each robot's motions, or use highly autonomous behaviors such as *goto* commands, providing a flexible level of interaction from the robot to the operator.

Given the capability to change between modules and handle them in a secure way, it is possible to develop decentralized software solutions for particular applications, which will then be used for a central module according to the final goal. Therefore, researchers do not need to worry about communication layers, and prioritization w.r.t other modules but may focus on their own research problem.

A lower level of abstraction has been presented in [9]. In that work, the authors introduce *"Free Gait"*. This architecture provides an easy way to integrate locomotion behaviors, through a common interface. The interface gives the user the capability to execute and integrate locomotion behaviors at different levels of abstraction. For example: task space, joint space, full trajectory description etc. Additionally, the architecture includes tracking controllers to increase the robustness during execution. *"Free Gait"* allows the user to integrate in a single module, locomotion gaits, control strategies and additional locomotion capabilities which centralize the locomotion behaviour.

This work is supported by the FP7 European project WALK-MAN (ICT-2013-10).

All authors are with the Department of Advanced Robotics, Istituto Italiano di Tecnologia, via Morego 30, 16163 Genova, Italy.
Email: `name.surname@iit.itt`

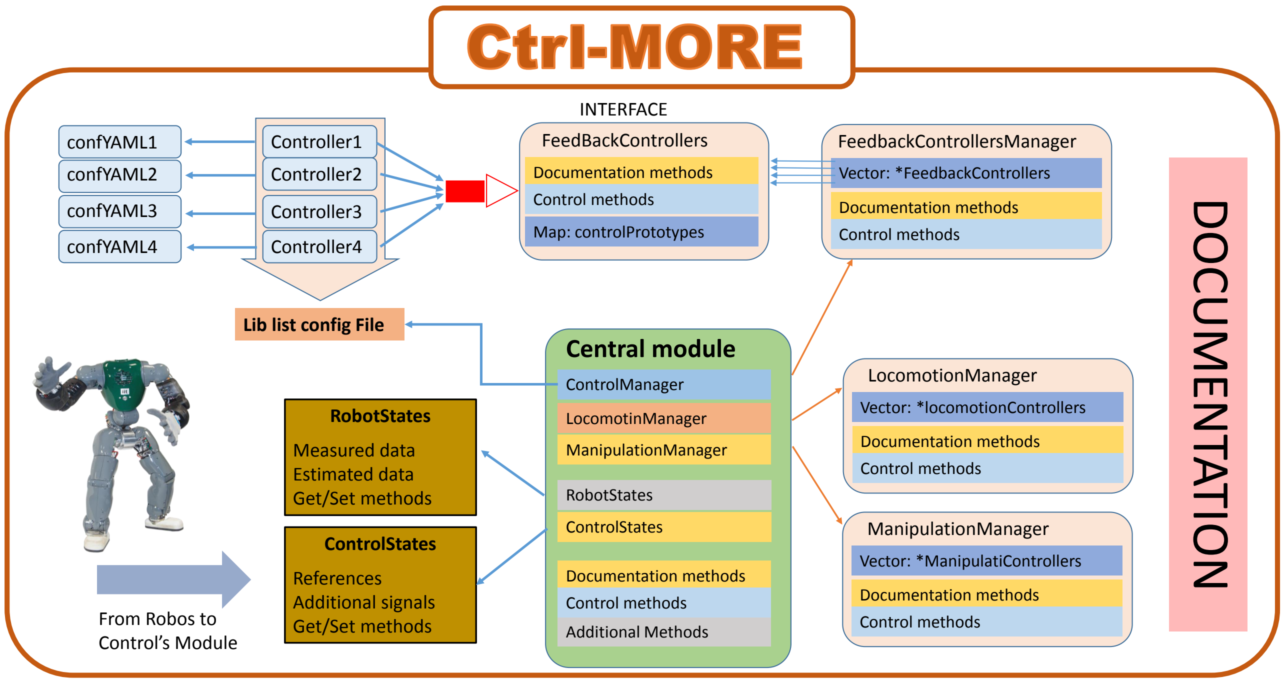

Fig. 1. Brief representation of the overall architecture scheme.

Following the presented line, the purpose of this work is to integrate control developments over different modular software solutions. Ctrl-MORE gives low-level control designers and high-level task designers a way to integrate their works within a single scope, as it is desired in the robotics field. We are willing to have a natural, secure, fast, and reflexive robot behavior. Therefore, task execution must consider external interactions and on-line modification of the original task trajectories. Some of such modifications are given by the task oriented module itself, such as gait pattern generator [10], adaptive bipedal locomotion [11], compliant stabilization [12], or the constrained solutions while using optimized inverse kinematic algorithms [13], [14]. Even though these solutions provide certain feasibility, it is necessary to incorporate additional closed loop capabilities in other modules as shown in [15]. In this sense, control developers are continuously making strategies that produce on-line task modifications, but most of the time these solutions are for specific use even though they can be of general purpose. Some examples are: balancing controllers in cooperation with bipedal walking [15], [16], Stabilization strategies when detecting external interaction [17], CoM and momentum control to provide stable gait execution.

The present work aims to reduce the gap between control developers and control users. On one hand, the given architecture permit the developers to create their controllers in a common interface to be easily shared with other users reducing the integration time and increasing the cooperation between colleagues. Therefore, the proposed framework, Ctrl-MORE, allows to develop controller that are independent from the robotic platforms and communication middlewares such as ROS [4]. On the other hand, Ctrl-MORE permits final users, such as task planners, to use existing control tools in their own modules in a simpler and clearer way i.e. Combining in the same module feedback stabilizers with manipulation strategies.

By using Ctrl-MORE the effort during the integration of the desired controllers is reduced, and allows the user to perform actions such as: use cascade control strategies, use individual controllers or modify various controller simultaneously. This way, the performance of the user's module increase once additional control strategies are used and active cooperation simplify. The open source files are available at [1]

II. General Description

The proposed architecture in Ctrl-MORE gives an easy interaction between control developers and additional users. From the control developers point of view, the architecture allows to share the developments with other users. Since the interface will provide a general abstraction of the control, it becomes easier to use and the integration time reduced.

In addition, we consider an abstraction of the robot's state using a common structure to encapsulate the robot's signals. This way the architecture is also easily ported to different robots allowing a better collaboration between colleagues.

However, to work in this fashion a responsibility of the developers is required, since each controller must have its corresponding documentation. The documentation lets the user know how to use a specific controller.

A general description of the overall architecture is shown in Fig. 1 and more specific details will be given in the following sections. As it is shown the `Documentation` is exter-

[1]https://github.com/ADVRHumanoids/Ctrl-MORE

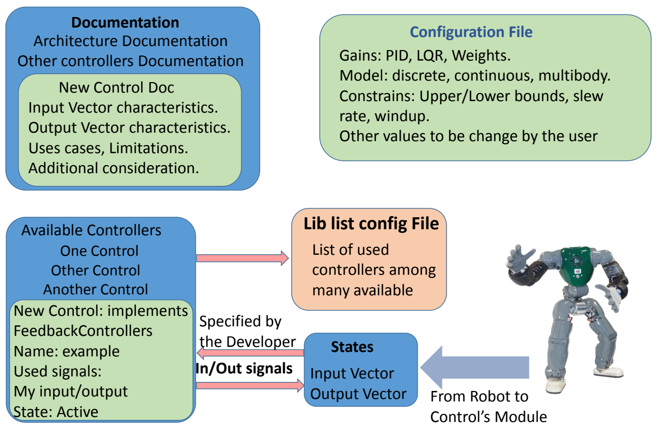
Documentation
Architecture Documentation
Other controllers Documentation
New Control Doc
Input Vector characteristics.
Output Vector characteristics.
Uses cases, Limitations.
Additional consideration.
Configuration File
Gains: PID, LQR, Weights.
Model: discrete, continuous, multibody.
Constrains: Upper/Lower bounds, slew rate, windup.
Other values to be change by the user
Available Controllers
One Control
Other Control
Another Control
New Control: implements FeedbackControllers
Name: example
Used signals:
My input/output
State: Active
Lib list config File
List of used controllers among many available
Specified by the Developer
In/Out signals
States
Input Vector
Output Vector
From Robot to Control's Module

Fig. 2. Architecture use from the developers point of view

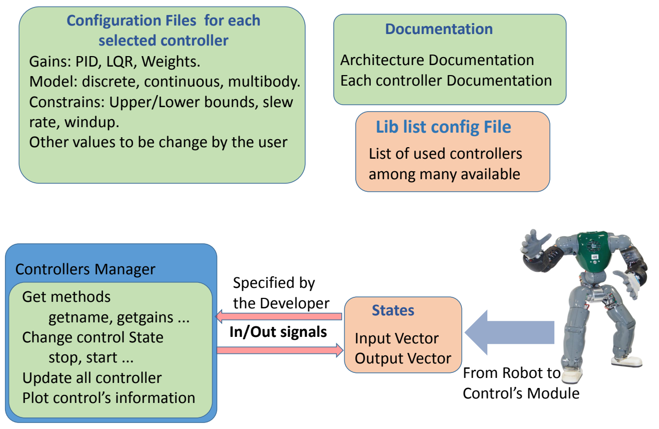
Configuration Files for each selected controller
Gains: PID, LQR, Weights.
Model: discrete, continuous, multibody.
Constrains: Upper/Lower bounds, slew rate, windup.
Other values to be change by the user
Documentation
Architecture Documentation
Each controller Documentation
Lib list config File
List of used controllers among many available
Controllers Manager
Get methods
getname, getgains ...
Change control State
stop, start ...
Update all controller
Plot control's information
Specified by the Developer
In/Out signals
States
Input Vector
Output Vector
From Robot to Control's Module

Fig. 3. Architecture use from the users point of view

nal to all modules, but fundamental for the proper use Section III-C. The modules `RobotStates` and `ControlStates` Section IV, encapsulates the robot's and controls' data in a common interface, facilitating the integration of the control strategies with different robots.

in Fig. 1 the `FeedbackControllersManager` is found, it is described in Section V-B. The manager provides an interface to handle and execute each of the controllers listed in the `Lib list config File`. This module provides an interface that gives the user the ability to apply different controllers and receive different information from the controllers such as states, names, and used signals.

To have a common interface for developers and users, each of the feedback controllers needs to implement the `feedbackController` class, which is an abstract class that defines the controller's interface and gives the standard methods to each controller such as *execute, paused, active, document*, and *read*. As can be seen in the diagram, each controller is linked to its particular config file which provides the final user with some degrees of freedom to tune the gains of the particular controller. Additionally, `Lib list config File` will contain the list of all the available controllers and will provide the final user a text interface to define which controllers should be loaded.

The `central module` of Ctrl-MORE implements the different managers for different purposes such as the `FeedbackControllersManager`. The list containing the managers and the corresponding control members i.e. (`ControlManager->FeedbackCtrls List`) are listed in a config file (`Lib list config File` in the figure). Additionally, the central module implements the robot and control states which are shared by all controllers.

A representation of the modules that are used by the controls' developers is given in Fig. 2, and we highlight the parts that are modified by the controller's developers in green. More details of each part of the architecture will be provided in the following sections.

First, the robot signal, including sensors readings, signal estimation, and control references are translated from the robot into the corresponding state container mentioned in Section IV. This step is guaranteed by the controllers' and robot's user. Each signal is stored in a map as a vector with a corresponding label. The vector characteristics, length, and order of data are defined by the architecture and controller's developer and should be properly documented.

Having the required signals in the state's module, the developer implements the specific controller and adds the controller's descriptive information such as controller's name, used signals, and gain characteristics. The handler will permit the final user to use but not to modify the controller shown in Section V-A.

During the implementation procedure, the developers have to add the descriptive characteristics needed by the user, and define the interface to load the gains using the parser Section III-A. With this in mind, the developer needs to provide the configuration file which gives the user capability to tune the controller variables like gains, limits, models etc.

Finally, the developer has to implement the documentation including the controller's characteristics and its uses and limitations, in a clear way for the user. Additionally, it is necessary to include the the paths to the new control library and the path to the configuration file in the list of existing controllers `Lib list config File`.

From the final user perspective, a brief description of the architecture is presented in Fig. 3, where we highlight the parts that are modified by the end user in red.

The final user has access to the architecture documentation including the documentation of each controller, written by the different developers. In addition, the user also has the configuration files to tune the controller's performance. The control architecture has the controller's manager interface, which provides a set of methods to get information from the controllers, methods to modify each control state, and a method to update all the active controllers.

There are two modification the user needs to make. The first one is the control selection file `Lib list config File`, which contains the list of desired controllers. The second modification is to write the robot's data into the given data container, following the signal's requirements of each control and the containers characteristics. This means that, if the IMU data are organized as three vectors namely $angPos, angVel, LinAcc$, the data are encapsulated in one of the maps in RobotState as a single vector as the pair [key,vector] $('imu', [angPos, angVel, LinAcc])$. This need to be done for each of the signals used by the controllers in agreement with the provided documentation.

III. Communication Interfaces

We have created a structure that is well documented and includes examples. It is ready to be used and both users, developers and control users, can start their integration using the proposed architecture. However, it is the developer's responsibility to document their own controllers for the final user. The documentation should clarify, how to use the controller, when to apply it, and the controller's gain characteristics, etc.

In order to make use of the different controllers and facilitate the integration and debugging, we use parser tools to load control libraries and individual controller's gains. Additionally, debug tools for the developing, integration and used phases are also required.

Even though these modules are standard, it is important to remark them since they are a crucial part of the structure and make possible a clear cooperation between colleagues.

A. Parser

The first communication tool is the parser, it is used to load the control gains, initialization parameters and control characteristics. In this architecture we used YAML files [18]. Using YAML files it is possible to load the controller's parameters using a text file. From the configuration point of view, the variables included in the parser are the degrees of freedom provided that the end-user has.

In this architecture, the parser has two applications: the first one is to store the gains, limits, models and other variables of each controller and that the final user might modify. The second is to load specific controllers of a list through a text interface. Furthermore, there is a YAML file that contains a list of paths to the control libraries and configurations files. In this file, the variable `loadControllers` lists the controllers that the end-user wants to load. A simple example of this file is reproduced below.

```
loadControllers: [Attitude, ZmpIP]
Compliant:
  libPath: /LIBPATH/locomotion/libfbkCtrlCompliant.so
   configFile: /PROJECTPATH/configs/compliant.yaml
ZmpIP:
  libPath: /LIBPATH/locomotion/libfbkCtrlzmpip.so
  configFile: /PROJECTPATH/configs/zmpip.yaml
AttitudeMPCController:
  libPath: /LIBPATH/locomotion/libfbkCtrlAttitude.so
  configFile: /PROJECTPATH/configs/Attitude.yaml
```

B. Logger

The logger we include in the interface is the one given in [5], the logger permit the display of warning and errors in the code. This way the control developers and final users have the information about possible code problems and where to locate them. The logger is used as a debugging tool and on each controller warnings and errors should be present. It is the developers responsibility to include the warnings and errors on the controller. This way, the final user is able to detect and correct possible errors during the control implementation and use. To keep the work flow independent from external modules, the logger is embedded in the structure and the XbotCore software is not required.

C. Documentation

To document the presented control structure and the developed controllers we are using Doxygen [19]. It provides a clear and well documented tool that final-users and controller's developers require to use the system and document the specific controllers. Since this framework aims to enrich cooperation between control developers and final users, proper documentation is imperative. The documentation of each controller clarifies it's use conditions and working scenarios. This way the final user can decide which control to use and know which are the implementation requirements.

IV. System States

As introduced in Section II, the framework encapsulates the robot data in a predefined interface. This is done to hide the controllers' developments from the hardware and software final user applications.

This generates a proper user-developer interaction since both users know beforehand the way they should handle input/output data. To encapsulate the data we implement two modules, the robot, and the control states. The first one centralizes the robot's information including measured data from encoders, IMU data, etc, as well as estimated data from external modules. The second module centralizes the controller's references and efforts. The two modules are decoupled such that the final user can consider the controller's information according to the particular needs.

A. Robot States

The robot state is a class that contains a set of get/set methods. The data are stored in maps of vectors with the corresponding string key identifier. Four different maps of data are used through the code:

- `measuredData`: map of measured data as vectors.
- `measurementTimeStamp`: time stamp for measured data, the key name should agree with the keys in `measuredData`.
- `estimatedData`: map of estimated data as vectors.
- `estimateTimeStamp`: data time stamp corresponding to the estimated data, the key name should agree with those in `estimatedData`.

By organizing the data in a map, the developer can ask directly for specific measures. To do so, the vector characteristics and the key name should be well documented by the controller's developer. As a result of this, the final user knows the input/output characteristics that they need to consider for each of the loaded controls.

One of the advantages of using maps is that they can be scaled according to the user requirements. When a new vector of data is required, a new pair `key,vector` is added to the particular map.

The robot states module informs users and developers when a new key-vector couple is added and when wrong access was required, i.e. a wrong size vector was added to a particular map or reading a non existing [key, vector] pair. This way the debug process during the control implementations is simplified. Additionally, the class has a print function that shows the information contained in each map.

B. Control States

This module is similar to the robot's state (see Section IV-A). However, this one contains information such as references and control efforts which are the controller's output.

Given that this framework was developed for robotics application we defined a set of containers for specific data. So, both the user and developer know in advance where to locate specific kind of signals with the corresponding set/get methods. The predefined containers have the corresponding errors and warning alerts already implemented to detect wrong access. The containers are:

- `taskSpaceReferencePose`: Task space reference pose.
- `taskSpaceReferenceVel`: Task space reference vel.
- `taskSpaceReferenceAcc`: Task space reference acc.
- `jointSpaceReference`: Joint space reference.
- `otherReference`: Other kind of signal.

As it can be seen, each map corresponds to a common type of reference in robotics which is in general a well defined size vector. The map `otherReference` contains all additional references that the controllers might require with no specific size. As an example, consider a reference of m samples for a predictive controller. Well known variables such as the Centre of mass, linear and angular momentum have their own container and get/set methods. The print function that shows the information contained in each map.

V. Controllers

To develop each of the controllers, and having the data centralized as described above, we provide a common interface to the controller's developers. We defined two classes; the first defines the set of capabilities each controller has and the methods that create a communication bridge between users and developers. This class is only used by developers. The second class is the user's interface that is used to apply one or more of the existing controllers.

A. Feedback Controller

The abstract class `feedbackController` declares a set of methods that has to be implemented by each new feedback controller. The methods define the requirements and general information the controller should have, providing a general interface to implement and share the controllers. In addition, the `feedbackController` class contains a type definition which is a pointer to a `feedbackController` class and an external map `feedbackcontrollerPrototypes`. Using this map, the controller's manager loads only the controllers the user requires. Further details are found at Section V-B.

The methods contained in each controller can be divided into control and documentation methods.

1) Documentation Methods: These are functions within each controller to inform the user about the specific controller characteristics. The given information includes input/output signals names, keys on the robot and control states maps, the controller's name, and the controller's state publisher. Additionally, There are functions to get and set the controller's name and the control/robot states; functions that check if the control is enabled or paused, and which are the states used by the control; and finally there are functions to provide the control characteristics as gains and developer additional information.

2) Control Methods: This set of virtual or pure virtual functions generalize the controller's implementation. These methods should be implemented by the user according to the specific control structure. The control methods permit to change the state to be paused or to resume the controller, enable or disable it; functions that initialize the controller and as principal function, there is `update()` which is called at each sampling to run the controller.

Implemented APIs
Documentation

`setName("name"), getName(), plot()`
`addUsedControlState("stateName")`
`addUsedRobotState("stateName")`
`isUsingRobotState("stateName")`
`isUsingControlState("stateName")`
`getUsedControlState() getGains()`

Control

`enable(), disable(), pause() and resume()`
`initialize(),loadGains("fileName"),update()`

TABLE I
IMPLEMENTES APIS IN THE FEEDBACKCONTROLLER

This module defines each controller interface and is common to all of the available ones, the implemented APIs are listed in Table. I. The variables to which each controller refer, are those related to the `robotStates` in Section IV-A and `controlStates` in Section IV-B. Additional variables should be defined as internal variables or in the configuration file to which the user has access.

B. Feedback Control Manager

The main functionality of the proposed software structure lies on this module. Here the user from different areas can call and apply different controllers in an easy and clear way. This class is the interface to the controllers and is where the user defines how to handle them. Some capabilities are: decide which of the available controllers should be loaded, activated or paused; change the controller's execution order, which is useful in case of prioritization or cascade control strategies; deactivate all the controllers that use a particular signal, which is useful during failures.

Through this class the final user can decide to use one or more controllers in a few steps as:

1) Define the control to be loaded in a configuration file.
2) write the input/output information using Section IV-A and Section IV-B.
3) execute the update method provided by the class.
4) use the control effort stored in the control states Section IV-B according to the specific application.

To have a wide range functionalities, additional features have been considered. This way we got a flexible use of different controllers while keeping the code of the users clean. These functionalities also minimize the code integration effort. The additional functionalities that were included are:

- `enable(), disable(), pause(), resume()`
  This set of functions will modify the state of all the controls that were loaded. This can be used for example during emergency stopping conditions, initialization and stop phases, etc.
- `isEnable("controlName")` `isPaused("controlName")`
  These functions check the state of the specific controller. These function are not only informative, but might prevent that two incompatible controllers are used at the same time.
- `pauseAllUsingRobotState("Name")` `pauseAllUsingControlState("Name")`
  These functions pause all controllers that are using a specific signal either from the robot state or the control state. This can be helpful in case of a damaged sensor, or when other programs will use a particular end effector.
- `resumeAllUsingRobotState("Name")` `resumeAllUsingControlState("Name"`
  These functions activate all controllers that are using a specific signal either from the robot or control state. This is useful during recovery phases, initializing phases, and modules activation.
- `getControllersNames()`
  Gives the user a vector of data containing the names of all the controllers that were loaded by the user independently of the controller's states
- `setExecutionOrder("stringVector")`
  By default, the execution order is the same as the one specified in the config file that defines which control should be loaded (Fig. 3. However, through this function, it is possible to modify the execution order while the robot is active. This way priorities and cascade behaviors can be adjusted to provide different capabilities and performances.

This way the user can decide within the same module whether to use one or more controllers, pause them according to the module or robot state, and activate them when required. This can be done without the necessity of additional initialization or communication with external modules since the control capabilities will be embedded on the user's loop.

Given that the idea of the present software structure is to centralize the feedback control developments, each user will have access to one or many controllers at the same time. However, it is desired to avoid an extended compilation while loading the controllers. Therefore, a map that links to the stand alone library of each controller is used during the initialization, loading only the required controllers and avoiding having a single library with several not used controllers. To this aim, each controller is compiled as an independent library but they are handled by the provided architecture. To permit this behavior, there is a shared map within the controllers' structure, mapping feedback controllers' prototypes. The developer should enable the prototype and include it in the YAML configuration file referred to in Section III-A. The prototype is created internally by the code of each controller using different naming such as:

```
extern ”C”
    FeedbackController* feedbackExampleMaker()
        return new ControlExample();
  } }
class dummyExample {
 public: dummyExample() {
     feedbackControllerPrototypes.insert({”MyController”
, feedbackExampleMaker});
 } };
 dummyExample localDummyExample;
}
```

Once the user loads the desired controllers which are given in a YAML file, the system will link the pre-compile controllers that are listed. See Section III-A

VI. Central module

Given that the presented structure can be extended for other applications rather than controllers, i.e, walking algorithms, an additional level of centralization was added to Ctrl-MORE. With the new abstraction layer, the central module is able to handle the individual managers such like `locomotionManager`, `feedbackControlManager`, etc.

This module provides unique robot and control states handlers to be used for all the different modules in a defined order by the final user. For example, executing first the locomotion algorithms and then the feedback controllers to obtain modified references. In addition, it provides the access to the individual manager configurations so that the user modifies the different manager's members states. The module gives permission to configure all control loops independently but updating all control and robot states at once.

A. User case example

As an example of the versatility of Ctrl-MORE, we implemented the attitude controller in [16] as one of the feedback controllers. This controller was ported to the presented structure implementing the `feedbackController` class.

Below we show an example of the final user code when implementing the attitude controller. It shows that the code is robot independent and the migrations from robot to robot is transparent, since it depends only on the data encapsulation, config files names and required control signals. This controller was implemented in three different robots using the same attitude controller with a slight change in the way IMU data and pelvis reference are considered.

```
\%Load controllers
fbkMngr=new Locomotion::Locomotor("fdbkConfFiles");
fbkMngr-> m_feedbackControllers ->("Attitude");
%Encapsulate Data in the robotState}
imu_map["imulink"]->(Rpelvis_abs ,LnAcc ,AngVel);
VectorXd imudata(9);
imudata<<AngPos ,AngVel ,LnAcc;
m_robotState.setMeasuredData("imu",imudata ,0);
\%Apply controller
fbkControllersMngr->update(m_robotState , &m_controlState);
PelvisO= m_controlState.getPelvisOrientation();
```

VII. Conclusions and future work

Ctrl-MORE allows both user and developers to actively cooperate towards more stable and capable robots. Through the use of the present structure, it is possible that a final user can have access to several controllers without developer supervision. As a result of this, the developments can be shared externally to a specific lab or project. From the user point of view, this tool allows easy access and interaction with different control strategies that might cooperate with locomotion or manipulation modules to generate more stable gaits and increase the robot's capabilities.

On the other hand, the use that developers might make of this structure depends on their desire to share and integrate their controllers with other users. This is because it implies a compromise of a well documented and proper implementation of all methods in Section V-A, which are not required for the control itself, but allow users to understand and properly use the controllers. This implies additional effort from the developer's point of view which in many cases is not necessary nor desired.

In Ctrl-MORE the use of paths is spread since it is required to load not only the gains files, which are modified by the user, but it required to locate the lib paths. This can generate errors and we are working towards developing an automatically linking system, reducing the complexity of the integration procedure. To provide a general tool, during the development of Ctrl-MORE's architecture we implemented different controllers in different robots, identifying needs and requirement towards a more general tool. However, possible modifications and additional capabilities are welcome, to create a general purpose tool for developers and user of different areas.

As mentioned in Section VI, Ctrl-MORE can be extended to another gait, such as locomotion. For this aim, it would be necessary to extend the manager class to include additional communication protocols. Recently, we have been working in the development of a locomotion control manager, to provide a tool that fills the requirements according to the lab expertise in bipedal locomotion. For this aim, we are considering the standardization and encapsulation of information like foothold characteristics and transition characteristics. We also consider trajectory handles to cope with CoM, CoP, ZMP trajectories etc. Additionally, we are defining communication protocols to generate a general interface to process different walking algorithms. These protocols should be global for the different controllers, similar to the `update()` in the feedback controllers.

References

[1] S. Jorg, J. Tully, and A. A. Schaffer, "The hardware abstraction layer - supporting control design by tackling the complexity of humanoid robot hardware," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 6427–6433.

[2] T. Murray, B. Pham, and P. Pirjanian, "Hardware abstraction layer (hal) for a robot," Mar. 20 2008, uS Patent App. 11/945,893. [Online]. Available: https://www.google.ch/patents/US20080071423

[3] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics*, vol. 3, no. 1, 2006.

[4] M. Quigley, K. Conley, B. 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*, vol. 3, no. 3.2, 2009, p. 5.

[5] L. Muratore, A. Laurenzi, E. M. Hoffman, A. Rocchi, D. G. Caldwell, and N. G. Tsagarakis, "Xbotcore: A real-time cross-robot software platform," in *2017 First IEEE International Conference on Robotic Computing (IRC)*, April 2017, pp. 77–80.

[6] N. Tsagarakis, S. Morfey, G. Medrano-Cerda, Z. Li, and D. Caldwell, "Compliant humanoid coman: Optimal joint stiffness tuning for modal frequency control," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013, pp. 665–670.

[7] 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*, 2016.

[8] S. Kohlbrecher, A. Stumpf, A. Romay, P. Schillinger, O. von Stryk, and D. C. Conner, "A comprehensive software framework for complex locomotion and manipulation tasks applicable to different types of humanoid robots," *Frontiers in Robotics and AI*, vol. 3, p. 31, 2016.

[9] P. Fankhauser, C. D. Bellicoso, C. Gehring, R. Dub, A. Gawel, and M. Hutter, "Free gait - an architecture for the versatile control of legged robots," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, Nov 2016, pp. 1052–1058.

[10] C. Zhou, X. Wang, Z. Li, and N. Tsagarakis, "Overview of Gait Synthesis for the Humanoid COMAN," *Journal of Bionic Engineering*, vol. 14, no. 1, pp. 15–25, 2017.

[11] J. A. Castano, Z. Li, C. Zhou, N. Tsagarakis, and D. Caldwell, "Dynamic and reactive walking for humanoid robots based on foot placement control," *International Journal of Humanoid Robotics*, vol. 0, no. 0, p. 1550041, 2015.

[12] C. Zhou, Z. Li, X. Wang, N. Tsagarakis, and D. Caldwell, "Stabilization of Bipedal Walking Based on Compliance Control," *Autonomous Robots*, vol. 40, pp. 1041–1057, 2016.

[13] A. Rocchi, E. M. Hoffman, D. G. Caldwell, and N. G. Tsagarakis, "Opensot: A whole-body control library for the compliant humanoid robot coman," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 6248–6253.

[14] C. Zhou, C. Fang, X. Wang, Z. Li, and N. Tsagarakis, "A Generic Optimization-based Framework for Reactive Collision Avoidance in Bipedal Locomotion," in *IEEE Conference on Automation Science and Engineering*, Fort Worth, TX, USA, August 2016, pp. 1026–1033.

[15] P. Kryczka, P. Kormushev, N. G. Tsagarakis, and D. G. Caldwell, "Online regeneration of bipedal walking gait pattern optimizing footstep placement and timing," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, Sept 2015, pp. 3352–3357.

[16] J. A. Castano, C. Zhou, Z. Li, and N. Tsagarakis, "Robust model predictive control for humanoids standing balancing," in *2016 International Conference on Advanced Robotics and Mechatronics (ICARM)*, Aug 2016, pp. 147–152.

[17] J. Castano, C. Zhou, P. Kryczka, and N. Tsagarakis, "MPC Strategy for Dynamic Stabilization of Preplanned Walking Gaits," in *IEEE-RAS International Conference on Humanoid Robots*, Birmingham, UK, November 15-17 2017, pp. 618–623.

[18] I. d. N. Oren Ben-Kiki, Clark Evans, *YAML Aint Markup Language (YAML^{tm}) Version 1.2*, 3rd ed., oct 2009, retrieved 09/2016.

[19] D. van Heesch, *doxygen, Manual for version 1.5.3.*