Modular ROS-based software architecture for reconfigurable, Industry 4.0 compatible robotic workcells

Mihael Simonič[1], Rok Pahič[1], Timotej Gašpar[1], Saeed Abdolshah[2], Sami Haddadin[2], Manuel G. Catalano[3], Florentin Wörgötter[4], and Aleš Ude[1]

***Abstract*— In this paper we present a novel software architecture for flexible and modular robotic workcells. It aims at providing robot independent, ROS-based programming environment that reflects hardware modularity and enables plug-and-produce connectivity within the workcell both in hardware and software. The distinguishing property of the developed architecture is that it supports the development process at all levels and also enables easy workcell setup and adaptation. Interfaces for the programming of robot movements by kinesthetic teaching and high-level task programming based on FlexBE are provided to enable workcell programming also by non-robotic experts. While the proposed system was developed to facilitate the implementation of automated disassembly solutions, it is of interest also for other industrial processes where high degree of reconfigurability is required, e. g. for automated production of small batch size or one-of-a-kind products.**

I. Introduction

One of the cornerstones of Industry 4.0 is the individualization of products under the conditions of large batch size production [1]. This is still problematic for robot-supported manufacturing because when setting up a traditional robotic cell to automate a specific task, it is necessary to anticipate all possible situations in advance and create a usually complex robotic program. Under such circumstances, every change in the production process requires both hardware and software modifications to the cell. Unless both hardware and software are modular and support easy and fast implementation of changes in the production process, the implementation of such changes results in long down-times, which can last for days or even weeks. Thus for more demanding tasks typical for Industry 4.0 where robot programs change frequently, a flexible robotic system that can be quickly adapted to various circumstances is required. This has to be addressed by enabling quick setup and reconfigurability [2] at both hardware and software level.

To facilitate the implementation of such workcells, we developed a modular software architecture that uses Robot Operating System (ROS) [3] as the backbone. It provides standardized interfaces to robots, sensors, grippers, and peripheral hardware elements and enables plug-and-produce connectivity and communication within the cell. Furthermore, toolchains for quick and efficient setup and programming on the workcell (task-level programming based on FlexBE [4], skill specification and workcell calibration by kinesthetic teaching) are integrated into the overall software architecture. Compared to our previous software architecture [5], [6], which was primarily aimed at automated reconfiguration of robotic workcells, we accommodate a much higher degree of hardware and software modularity within the system proposed in this paper.

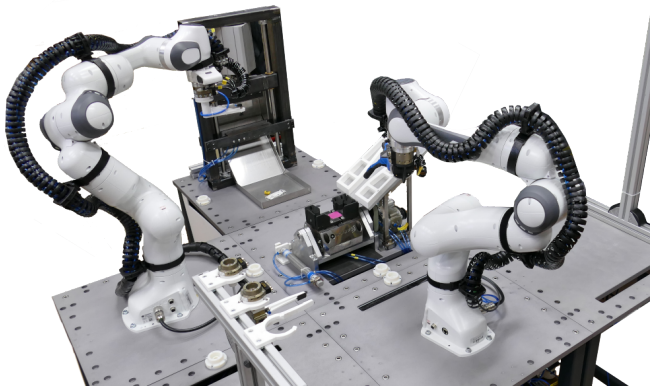

Fig. 1. Reconfigurable robotic workcell for automated recycling of electronic waste. The targeted recycling procedure involves the disassembly of electronic devices to remove batteries.

The proposed design enables that the cell's functionalities can be expanded without disrupting the existing software architecture. This is supported by making use of Docker containers for the integration of new software modules. In the Docker containers, new modules can be deployed with all the necessary libraries without causing conflicts with any preexisting software packages. Developers can thus integrate new software without the need to reprogram any of the existing modules, which also eases the deployment of new hardware in the cell.

The rest of the paper is organized as follows. In Section II we describe the core software architecture of the system and the archetypical hardware module from which the complete workcell can be constructed. In Section IV we present the more advanced module used to mount and control a robot as well as user-friendly tools that are made available for

[1]Department of Automatics, Biocybernetics and Robotics, Jožef Stefan Institute, Ljubljana, Slovenia, e-mail: {mihael.simonic, rok.pahic, timotej.gaspar, ales.ude}@ijs.si

[2]Munich School of Robotics and Machine Intelligence, Technical University of Munich, Germany, e-mail: {saeed.abdolshah, sami.haddadin}@tum.de

[3]Italian Insitute of Technology, Genoa, Italy, e-mail: manuel.catalano@iit.it

[4]Bernstein Center for Computational Neuroscience, Georg-August-Universität Göttingen, Germany, e-mail: worgott@gwdg.de

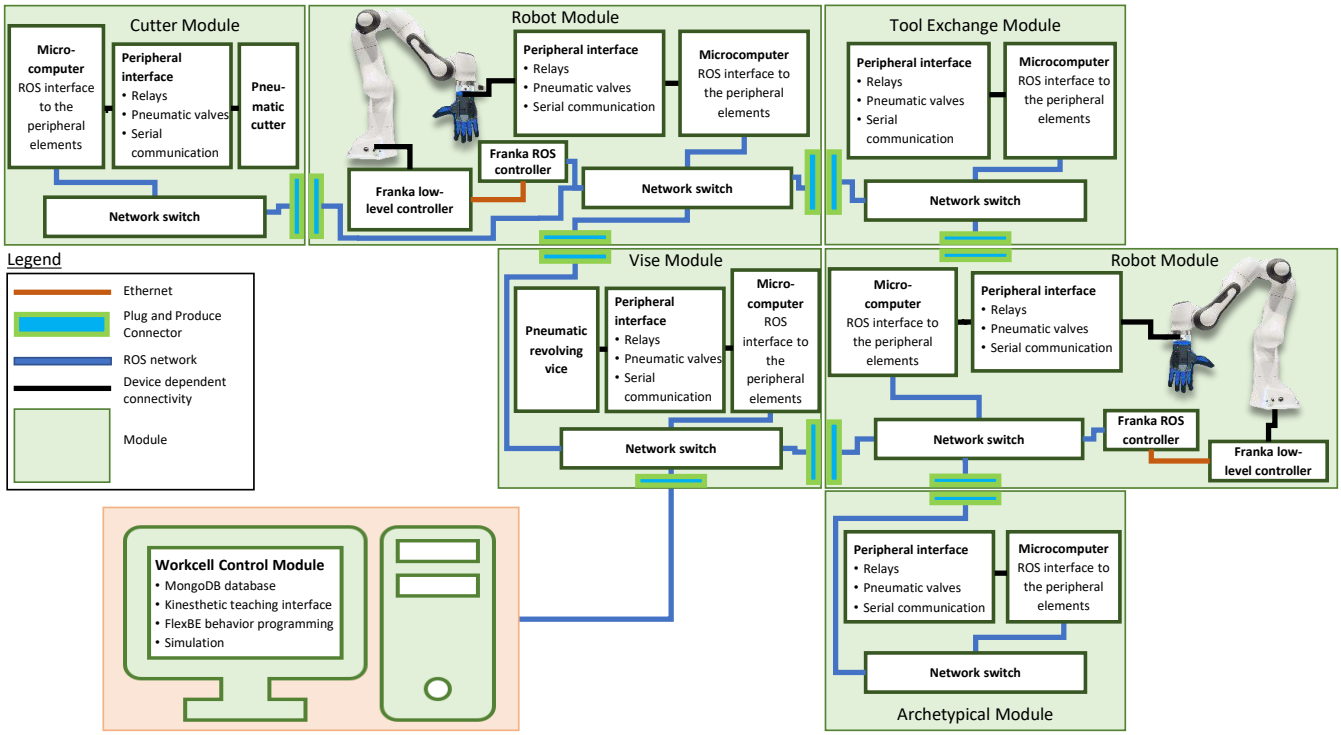

Fig. 2. Software architecture to control the recycling cell from Fig. 1. Except for the "Workcell Control Modue", all other modules correspond to one of the tables from which the working area of the cell is constructed.

easy workcell setup and programming. The application of the developed architecture in the recycling domain is presented in Section IV. It demonstrates the necessity for modular software and hardware design and its applicability to situations where the manufacturing process changes frequently. Finally, in Section V our current implementation and plans for future work are discussed.

II. System Architecture Overview

The purpose of the software architecture is to define all the constituent modules and ensure connectivity between them in the context of data flow. While several software frameworks exist [7], [8], the Robot Operating System (ROS) [3] provides the most widely used framework for the development of robotic software architectures where components exchange data over the shared network. Various tools and features that are available within ROS contribute to realizing the pursued software reconfigurability of the cell [5]. In our case, software modularity and reconfigurability mean that it is possible to expand or adapt the cell's functionalities without disrupting the existing software architecture. It should be possible to develop new software components without the need to reprogram any of the existing ROS nodes (the definition of ROS nodes is provided below). This also eases the development and integration of new hardware components with their own ROS nodes.

Of the many features and tools provided within ROS, we use the following ones to achieve a high degree of software modularity in our system:

- *nodes* – any program (written in any programming language) that has connectivity to the ROS network and can therefore access to and publish data across it (i.e. low-level hardware drivers, high-level state machines, trajectory generation, etc.)
- *topics* – a publish/subscribe table advertised by each ROS node that defines the data that can be provided by the said node, e. g. robot joint states, force-torque sensor data, state of the cutter, etc.
- *messages* – a predefined structure to encapsulate data to be transferred across the ROS network for other nodes to read, e. g. robot joint states are written into *sensor_msgs/JointStates*, which is a predefined ROS message structure that can be sent across the ROS network.
- *parameter server* – used to store various static configuration parameters, e.g. controller gains, camera exposure parameters, kinematic models, etc.
- *services* – a request/response based Remote Procedure Call (RPC) interface that a node can expose in order to trigger short-running tasks that do not require preemption or monitoring from within the ROS network, e. g. visual quality control, pneumatic gripper actuation, tool exchange system lock/unlock, gravity compensation mode toggle, etc.
- *action servers* – similarly to *services*, a request/response RPC exposed by a node. They are, however, used to trigger long-running preemptable tasks from within the ROS network that provide feedback throughout their execution, e. g. robot movement tasks, servo gripper grasping tasks, flexible fixture reconfiguration [9], etc.

In the developed architecture, each module is connected to the same network in order to broadcast its data and receive information and instructions about what action to perform at any given time. An example implementation is shown in Figure 2. Its constituent components and their integration are described in more detail throughout this paper.

Apart from ensuring software reconfigurability, the proposed architecture allows us to control and monitor all the different modules in the cell as well as the workcell as a whole. The developed system is designed in such a way that each module connects directly to the ROS network. This way we ensure that the data is structured and parsable by all of the software components within the developed system.

An important feature of the proposed architecture is that we can program and exchange information between heterogeneous hardware modules within a single software architecture. Once the developer integrates a new module into ROS, the workcell programmer needs to know only which functionalities the new module exposes to ROS. No special knowledge about hardware-specific software is needed to start programming new workcell applications.

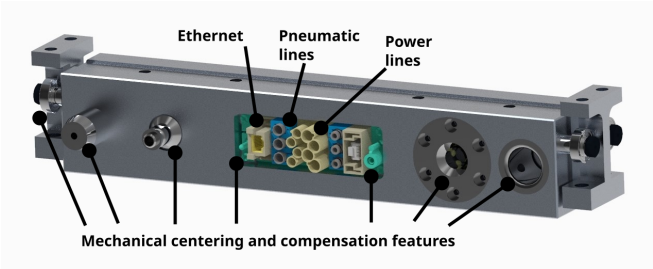

Fig. 3. Plug-and-Produce (PnP) connector that provides connectivity, mechanical coupling, and power (electric and pneumatic)

A. Archetypical workcell module and PnP connectivity

On the hardware side, each module in the proposed cell is built according to the the same archetypical design: a steel frame that provides rigidity, with aluminium work surface that allows for easy mounting of module-specific equipment, e.g. robots, sensors, and auxiliary devices (see also Fig. 1). Each module is further equipped with sufficient computational hardware to run ROS nodes, thus exposing its data and functionalities to the cell's ROS environment. This way the modules can be controlled by the top-level task scheduling software as soon as they are connected to the cell. Some modules may require more than just network connectivity in order to function properly, e. g. compressed air or electric power. To provide such capabilities, modules are connected to each other using Plug-and-Produce (PnP) connectors shown in Fig. 3. More details about hardware design and PnP connectors are provided in [10].

The computational hardware that every module in the cell is equipped with is a Raspberry Pi 4 microcomputer. We mounted the so-called "PoE Hat" on each Raspberry Pi. This allows it to be powered using power over ethernet (PoE) standard, thus reducing the amount of the necessary power supply units. There are other devices that can make use of the PoE connectivity (e.g. cameras). To connect them to the network and at the same time provide them with power, we installed a PoE-enabled network switch on each of the modules.

A Raspberry Pi microcomputer provides us with the ability to pass control signals and read the sensor values by attaching the auxiliary devices (equipment) to the General Purpose Input/Output pins (GPIOs) of the Raspberry Pi. When a new auxiliary device is connected to the GPIOs of the Raspberry Pi, each GPIO in use must be properly configured by a suitable software library. Since the auxiliary devices attached to each individual module needs to work in synchronization with the robots and the auxiliary equipment of other modules, we need to be able to control the equipment globally throughout the cell. Therefore we have prepared a ROS package that wraps the developed software library for configuring and controlling the GPIOs in a ROS node[1]. This way we enable the configuration and control of auxiliary devices through ROS services. The cell programmer no longer needs to deal with the GPIOs but can control and communicate with the auxiliary devices through ROS interfaces.

B. Quick integration of new modules & auxiliary devices

For the operation of each module's auxiliary devices, we implemented two ROS nodes running on each individual module's microcomputer. The first ROS node is "Equipment Server", which configures control of the equipment connected through the GPIOs according to the module configuration file (in human-readable yaml format) and forwards control commands from ROS services to the connected equipment. The second ROS node is "Equipment Manager", which allows a user to modify the module's configuration file via a ROS service.

When the "Equipment Server" node is started, it first reads the individual module's configuration file, which is stored locally on the module's microcomputer, and then configures the required GPIOs. The configuration file contains information which additional equipment is attached to the module and to which GPIOs it is connected. Once GPIOs are configured, the "Equipment Server" creates a separate ROS service for each GPIO to control it. The names of the created ROS services are defined in the configuration file. In operation, the "Equipment Server" accepts the commands sent to its ROS services and controls the equipment connected to the GPIOs accordingly. Currently the "Equipment Server" can handle three types of different GPIO configurations and control interactions. The first possible configuration is a digital output where a service call can set the digital state of the GPIO, making it suitable for controlling devices such as pneumatic valves. The next configuration is a digital input that allows the digital state of devices such as digital sensors to be read on a service call. The last is a configuration that, according to the value in the service call, controls pulse width modulation (PWM) signals that can be used to control devices such as step motors. The node also has

[1] https://github.com/ReconCycle/raspi_ros

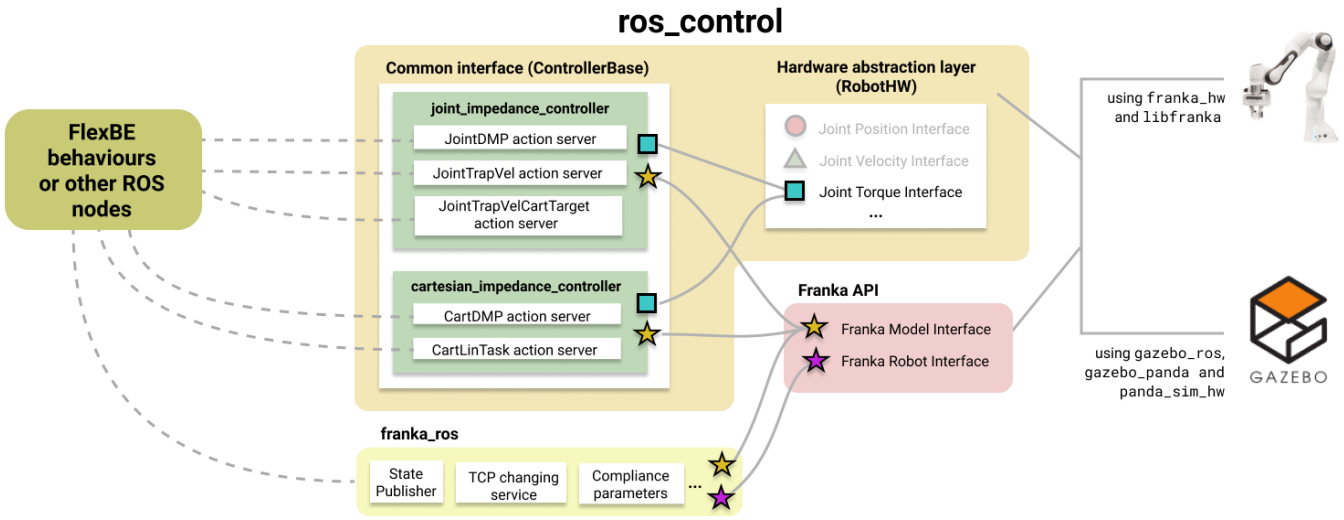

Fig. 4. Integration of controller plugins into the overall ROS arhitecture with ros_control framework.

a restart service that – when triggered – closes all active services, releases the pin's hardware interface, and reads the configuration file again. Then it starts with the newly read configuration.

When switching from one production process to another, we often need to change, add or remove various auxiliary devices attached to the modules. The "Equipment Manager" node allows us in these cases to quickly change the "Equipment Server" configuration according to the changes in the auxiliary equipment. We change the configuration by sending the new desired configuration to the "Equipment Manager" ROS service. When the Equipment Manager receives the new desired configuration, it overwrites the configuration file and restarts the "Equipment Server" by calling the node's restart service. In this way, the "Equipment Server" reconfigures itself according to the new configuration file.

To avoid writing or correcting configuration files manually, we created a ROS package[2] with a more user-friendly approach to handling configuration files. This package contains a client that can communicate with the "Equipment Manager". When the client is started, it opens a terminal window user interface that guides the user through creating or modifying configuration files. At the beginning, the client searches for all "Equipment Managers" from the different modules in its reach and offers the user to select the one s/he wants to configure. In the next steps, the user can choose whether to modify the currently active configuration file or start over with a blank template. According to the selected option, the client then reads the correct configuration file from the "Equipment Manager". When changing the configuration, the user only needs to answer the questions about the various parameter values asked by the terminal guide. When the user is satisfied with the desired configuration, the client automatically changes the configuration file and sends it to the module "Equipment Manager".

C. Docker containers

Although ROS provides a good framework for the development of robotic workcells, setting it up on a single computer still takes some effort and time. Our system is composed of several modules, each with their own computer. Setting up ROS and maintaining all of them would be very time-consuming. Moreover, the transfer of ROS code from one machine to another can be rather difficult. These difficulties arise if the developers do not properly define all the external resources (dependencies) that their code depends on. To address these two issues, we decided to base our development process and overall system on Docker containers.[3]

A Docker container is an isolated environment that is built from a `Dockerfile`. In this file, we specify which Linux distribution the container is based on and what types of dependencies should be installed. The main advantage of this approach is that unlike virtual machines, Docker containers do not emulate the host's hardware but share it. This in turn means that, compared to a virtual machine, a Docker container uses fewer resources. Additionally, once the `Dockerfile` has been written, the image that is built from it will be the same regardless of the platform it runs on. In terms of deploying ROS software on different modules, this means that the developer designs the code in such a way that it runs within the Docker container and thus removes the commonly encountered problem of unmet software dependencies when transferring the code. The details of how the Raspberry Pi integrates into each workcell module are provided in Section II-B.

In terms of network connectivity, Docker containers can communicate between each other just like any other programs, including ROS nodes. This means that different software components running in different Docker containers can exchange data seamlessly. Therefore, using this technology

[2] https://github.com/ReconCycle/raspi-ros-client

[3] https://docs.docker.com/

does not hinder the overall ROS software architecture but it simplifies the set up of modules and the transfer of code.

III. Advanced functionalities and GUIs

A. Robot module

The archetypical workcell module presented in Section II-A is in most cases equipped with relatively simple devices that can be controlled through a microcomputer. However, for more complex equipment such as robots, more advanced control schemes are required. Consequently, both hardware and software must be modified to accommodate such equipment. Our example cell in Fig. 1 contains two robot modules.

We based the software of the robot module on the `ros_control` framework, which provides a hardware abstraction layer (`RobotHW`) that enables standardized access to actuators and comes with a common interface (`ControllerBase`) to write robot-agnostic controllers [11]. The robot middleware is represented by the robot's hardware interface. A reference implementation for the Franka Emika Panda robot is provided by the `franka_hw` ROS package using the `libfranka` library as shown in Fig. 4[4]. It enables compliant control needed in many disassembly tasks.

To enable simulation in Gazebo, we used a community-built Panda model for Gazebo [12] and developed a plugin that mimics the API of the real Panda robot, enabling easy sim-to-real transfer[5]. This way, the trajectory generation and control algorithms can be developed independently of the robot employed in the cell. In our software stack, various trajectory generation algorithms have been implemented to cover the most common robot motion needs:

- joint space point-to-point trajectory with trapezoidal velocity profile [13],
- Cartesian space straight line point-to-point motion & quaternion SLERP trajectory [14] with minimum jerk time evolution,
- joint space point-to-point trajectory with trapezoidal velocity profile, with the initial and final pose provided in Cartesian space and transformed into joint space using inverse kinematics,
- dynamic movement primitive (DMP) in joint space [15],
- Cartesian space DMP [16], [17].

To enable interaction with these plugins, we provided a separate ROS action server wrapper for each of the motion generators (JointTrapVel, JointDMP, JointTrapVelCartTarget, CartDMO, CartLinTask in Fig. 4). Whenever a new movement request arrives to a specific action server (in ROS terminology *action goal*), the underlying motion generator plugin generates either a joint or Cartesian space trajectory. The trajectory points are processed in a parallel real-time safe thread, running at 1 kHZ. In this thread, we rely on joint and Cartesian impedance control to calculate the desired joint torques. Finally, the calculated torques are sent to the low-level robot controller via `RobotHW` interface, provided by the `franka_hw` package for real Panda robot arm or `panda_sim_hw` package for Gazebo simulation, respectively.

The main benefit of using ROS action servers to trigger robot motion is the ability to cancel the request during execution and to get periodic feedback about how the request is progressing. Upon acceptance, the action goal's status is set to active if there are no other action goals, e.g. motions, waiting for execution. If the action goal is preempted, the robot does not immediately enter an emergency state and does not require any restart procedure. The client receives appropriate result messages in order to handle the preemption in its scheme and continue with another action if required/possible. This approach also enables the integration with state machine frameworks for behavior level programming, such as for example FlexBE [4], or integration with different motion planning software stacks such as the widely adopted MoveIt! [18].

B. Persistent data storage with MongoDB

All ROS nodes within the ROS network can access the data on the network by subscribing to *topics* or by reading from the *parameter server*. This is meant to either provide data of the current state of the workcell or some general parameters that can be used. To carry out a specific task, the robot has to move through various configurations in its workspace. The data required to generated these motions need to be stored as persistent data and read during the disassembly process. These configurations need to be stored as persistent data and read during the process. It is also required that we are able to modify these data as the need arises, e.g. when the final pose of the robot at one step of the disassembly process changes.

To meet all these requirements, we decided to store the motion configuration data in the MongoDB database. We integrated this database in our ROS software architecture by using the already existing `mongodb_store` ROS package.[6] In our setup, the MongoDB database runs on the ROS master computer. All the data are saved as named entries into the MongoDB database. For point-to-point movements, the initial and final robot configurations are saved, whereas complex trajectories are saved as parameters of dynamic movement primitives (DMPs) [15], [16]. It then becomes possible to define a high-level action sequence that reads these named entries (single configurations or DMPs) from the database and moves the robot accordingly. The high-level disassembly sequences are programmed using FlexBE (see Section III-C). The poses and trajectories are saved in the database as ROS messages corresponding to each type of movement. Having these trajectories saved as named entries enables quick reconfiguration in terms of changing robot motion. It is sufficient to overwrite the entry in the database with a modified motion to update the disassembly sequence without changing the high-level disassembly sequence program.

[4] `https://github.com/frankaemika/franka_ros`

[5] `https://github.com/smihael/panda_sim_hw`

[6] `https://github.com/strands-project/mongodb_store`

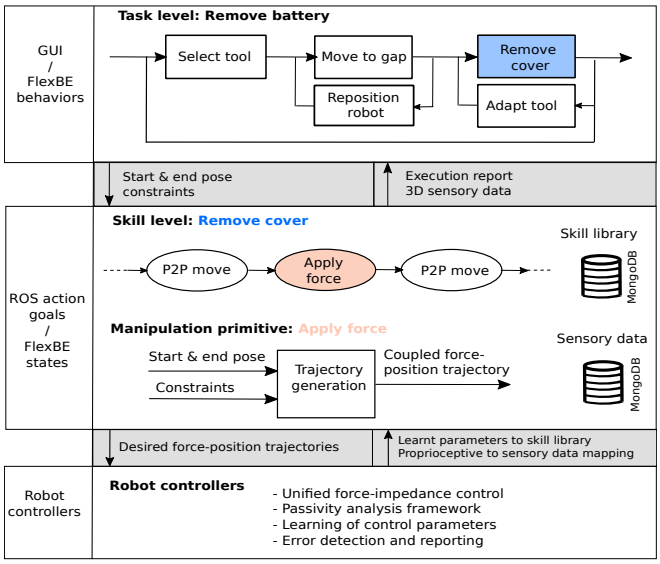

Fig. 5. Hierarchical task specification and adaptation in the proposed system

C. Integration of FlexBE state machine engine

Manually coding a program that sends instructions in a specific order to all of the components of the cell requires a deep know-how of both the programming language and the underlying system. To facilitate this process, we integrated FlexBE in our workflow. FlexBE is a high-level behavior engine that supports creating, executing and monitoring complex robot behaviors [4] as state machines. It also provides a graphical user interface where each state is represented by a square and the transition between them are represented by arrows. This provides the user with a graphical interface to change the state machine sequence by simply adding or removing states and creating connections between them. Additionally, states can be grouped into sub-behaviors and concurrent containers, where the latter provides means to execute several states in parallel, which can significantly improve cycle times.

We have created multiple custom-made FlexBE states that are used to execute robot trajectories, control the peripheral machinery, manipulate the process data, etc.[7] Each FlexBE state is associated with a service or action server to perform the following operations:

- Reading from and writing to the skill library (MongoDB database), e.g. to store data acquired by kinesthetic teaching and to read data to initialize the desired robot movements (using `mongodb_store`),
- Controlling and monitoring the execution of robot motions through Action Servers (c.f. Section III-A),
- Controlling the peripheral equipment, e.g. grippers, pneumatic vise, etc. using tools presented in Section II-B, and
- Sensor data acquisition and processing, e.g. vision pipeline.

[7] https://github.com/ReconCycle/reconcycle_states

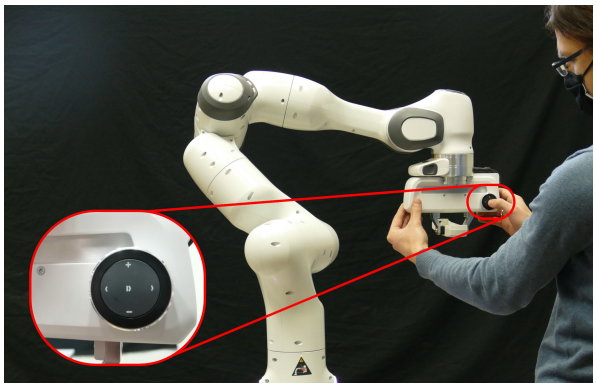

Fig. 6. Bluetooth media controller is mounted on the robot arm to provide an easy-to-use interface to the Helping Hand GUI.

Using these states, we can develop complex high-level behaviors while hiding the low-level implementation details from the user as shown in Fig. 5.

D. Helping Hand GUI for kinesthetic teaching

The definition of robot motions is a difficult and time consuming process unless proper tools are provided. Programming by Demonstration (PbD) provides a methodology to define robot motions in a natural way rather than by coding [19]. To capture trajectories, kinesthetic guidance [20], [21] is commonly used because it enables the demonstration of tasks directly in the robot's workspace by physically guiding it along the desired path, thus avoiding the correspondence problem typical for PbD.

To integrate kinesthetic teaching into the overall software infrastructure, we developed a ROS package called Helping Hand[8]. The Helping Hand GUI enables the recording of individual poses or robot configurations as well as the recording of continuous motion sequences to specify smooth trajectories. The first functionality is commonly used for fast calibration of the workcell by guiding the robot to the fixtures attached to the individual modules that form the workcell (see Fig. 1). The obtained robot configuration is used to define the relative transformation from the robot base to the fixture. This way we can quickly obtain the poses of all workcell modules in the robot coordinate frame. Afterwards, all other existing data can be retrieved in any of the calibrated coordinate frames with the required transformations applied automatically.

To provide a user-friendly interface for kinesthetic guidance, we equipped the Panda robots with buttons that connect to a computer via a Bluetooth media button (Fig. 6). A ROS package[9] was developed that reads the button key-press events and triggers the Helping Hand GUI events described above via ROS topics.

IV. Application Example

To demonstrate that the proposed framework can be applied for advanced tasks, we present the implementation

[8] https://github.com/tgaspar/helping_hand
[9] https://repo.ijs.si/msimonic/keypress_monitor

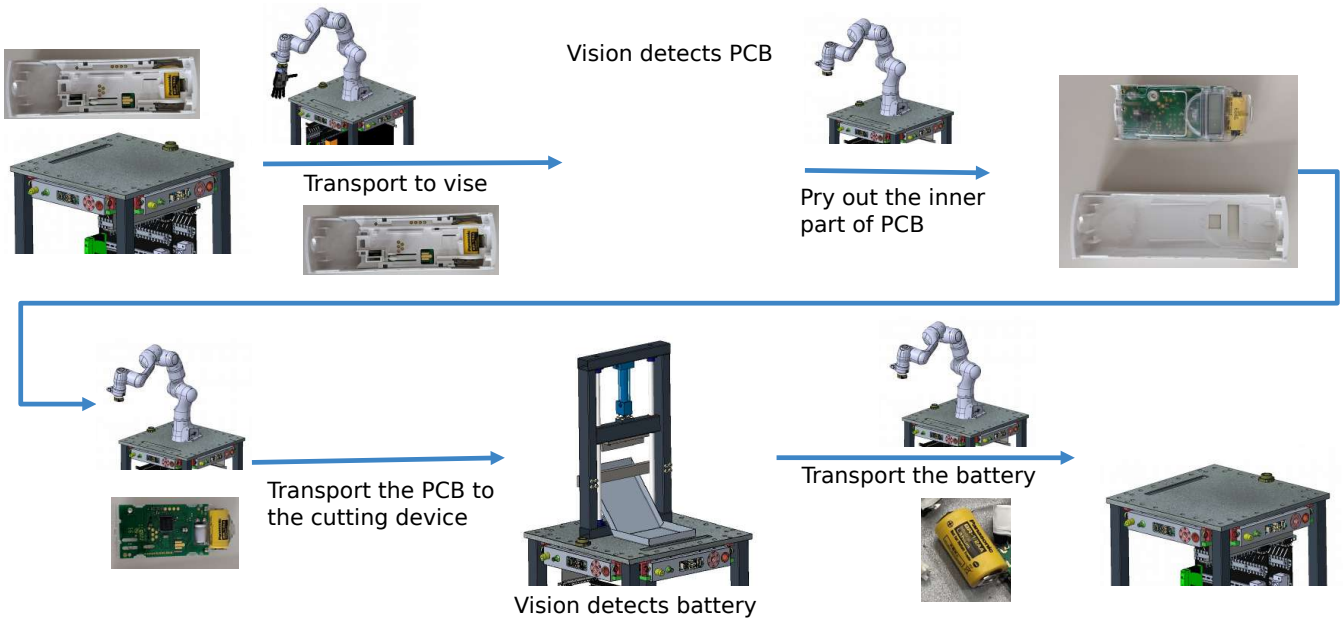

Fig. 7. The automatic disassembly workflow for a heat cost allocator

of a disassembly task in the context of electronic waste recycling, which is still largely dominated by manual labor. In recycling, the robot needs to deal with many different products exhibiting different states of damage. Therefore, robotic workcells for the recycling of electronic devices need to allow flexible adaptation to be able to successfully disassemble individual devices.

Using the proposed architecture, we were able to implement the disassembly of heat cost allocators (HCA). The main objective of this example disassembly task was to remove the PCB containing the soldered battery from the plastic housing and then cut the battery from the extracted PCB.

We first analysed the disassembly as carried out by a human operator. This analysis served as the basis for specifying the automated workflow and the modules of the workcell to carry out the required operations. The steps are shown in Fig. 7. We equipped the archetypical modules with a pneumatic-driven vise and cutter and provided two robot modules. The other two modules are passive and are used for tool exchange and as an entry point for devices that need to be disassembled (see Fig. 1). The integration of these hardware elements was easy using the tools presented in Section II-B.

In the current implementation, the first robot starts by picking up the HCA using qb SoftHand Research gripper (based on the Pisa/IIT SoftHand [22]) and places it into the pneumatic vise, which clamps the housing firmly so that the second robot can use the lever tool to break the PCB out of the housing. When the PCB is released, the vise is rotated upside down, allowing the PCB to fall onto the tray below. When the vise returns to the upwards position, the tray extends out from underneath it, allowing for the unobstructed pick-up of PCB in the next step. At the same time, vise jaws are opened, allowing the first robot to pick up the empty housing using qb SoftHand and transfer it to the designated container. After breaking of the PCB from the plastic housing, the second robot changes its tool. It leaves the lever tool in the tool storage and takes the vacuum gripper to pick up the circuit board from the tray and places it into the cutter. When the battery is cut from the PCB, the second robot uses the same vacuum tool to pick up the battery and places it into the container designated for the removed batteries.

The high-level specification of the disassembly process was programmed as a FlexBE behavior, while the robot configurations and trajectories needed to execute the required robot operations were acquired by kinesthetic teaching, as described in Section III-D.

The result was a modular workcell with an accompanying program to disassemble HCAs, which together enable quick adaptation of both hardware and software to different HCAs. A video showing the current implementation of the disassembly process is available as an attachment to this paper.

V. Conclusion and Future Work

In this paper we presented a new ROS-based software stack for flexible robot workcells suitable for agile and self-adaptable production systems as required by the Industry 4.0 paradigm. By following the modular software and hardware paradigm, we are able to support both fast development and deployment of new software and hardware module couples (*developer* tasks) as well as fast setup and reconfiguration of the workcell to prepare specific manufacturing processes (*programmer* tasks).

From the developer point of view, the functionalities of the cell can be expanded without the need to reprogram any of the software modules that control the existing cell.

Most new modules can be developed by simply defining an appropriate configuration file for the archetypical module, where the configuration file specifies the format of ROS messages that can be received by the module over the ROS network and how they should be interpreted to control any auxiliary devices mounted on the module via GPIOs. If an existing module should be expanded with new capabilities, this can be achieved by integrating new ROS nodes in Docker containers, thus avoiding any conflicts with the existing software in the cell.

From the programmer point of view, the provided software toolchains enable fast and efficient programming at all three levels defined in Fig. 5: control, skills, and behaviors. New real-time controllers can be developed by exploiting `ros_control` framework, where the controllers can be tested in the Gazebo simulation. The programming of new skills as well as workcell calibration are supported by a user-friendly interface for kinesthetic guidance. Finally, FlexBE behavior engine has been integrated into the software stack to easily specify complex sequences of actions.

While the presented workcell implementation primarily targets the field of electronic waste recycling, it can also assist the manufacturing industry that targets small production batches where shifts in demand occur frequently.

ACKNOWLEDGMENT

This work has received funding from the EU's Horizon 2020 grant ReconCycle (grant agreement no. 871352).

REFERENCES

[1] K.-D. Thoben, S. Wiesner, and T. Wuest, ""Industrie 4.0" and smart manufacturing – A review of research issues and application examples," *International Journal of Automation Technology*, vol. 11, no. 1, pp. 4–16, 2017.

[2] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, and H. V. Brüssel, "Reconfigurable manufacturing systems," *CIRP Annals*, vol. 48, no. 2, pp. 527–540, 1999.

[3] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. Sebastopol, CA: O'Rilley Media, 2015.

[4] P. Schillinger, S. Kohlbrecher, and O. von Stryk, "Human-robot collaborative high-level control with application to rescue robotics," in *IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, Sweden, 2016, pp. 2796–2802.

[5] T. Gašpar, M. Deniša, P. Radanovič, B. Ridge, T. R. Savarimuthu, A. Kramberger, M. Priggemeyer, J. Rossmann, F. Wörgötter, T. Ivanovska, S. Parizi, Ž. Gosar, I. Kovač, and A. Ude, "Smart hardware integration with advanced robot programming technologies for efficient reconfiguration of robot workcells," *Robotics and Computer-Integrated Manufacturing*, vol. 66, art. no. 101979, pp. 1–17, 2020.

[6] T. Gašpar, B. Ridge, R. Bevec, M. Bem, Ž. Gosar, I. Kovač, and A. Ude, "Rapid hardware and software reconfiguration in a robotic workcell," in *18th International Conference on Advanced Robotics (ICAR)*, Hong Kong, 2017, pp. 229–236.

[7] N. Vahrenkamp, M. Wächter, M. Kröhnert, K. Welke, and T. Asfour, "The robot software framework ArmarX," *it - Information Technology*, vol. 57, no. 2, pp. 99–111, 2015.

[8] P. Fitzpatrick, E. Ceseracciu, D. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, pp. 42–49, 2014.

[9] T. Gašpar, I. Kovač, and A. Ude, "Optimal layout and reconfiguration of a fixturing system constructed from passive stewart platforms," *Journal of Manufacturing Systems*, vol. 60, pp. 226–238, 2021.

[10] P. Radanovič, J. Jereb, I. Kovač, and A. Ude, "Design of a modular robotic workcell platform enabled by plug & produce connectors," in *20th International Conference on Advanced Robotics (ICAR)*, Ljubljana, Slovenia, 2021.

[11] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo, "ros_control: A generic and simple control framework for ROS," *The Journal of Open Source Software*, vol. 2, no. 20, p. 456, 2017.

[12] S. Sidhik, "panda_simulator: Gazebo simulator for Franka Emika Panda robot supporting sim-to-real code transfer," Apr. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3747459

[13] K. M. Lynch and F. C. Park, *Modern Robotics; Mechanics, Planning and Control*. Cambridge University Press, 2017.

[14] K. Shoemake, "Animating rotation with quaternion curves," *SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 245–254, 1985.

[15] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors," *Neural Computation*, vol. 25, no. 2, pp. 328–373, 2013.

[16] A. Ude, B. Nemec, T. Petrič, and J. Morimoto, "Orientation in Cartesian space dynamic movement primitives," in *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China, 2014, pp. 2997–3004.

[17] L. Koutras and Z. Doulgeri, "A correct formulation for the orientation dynamic movement primitives for robot control in the Cartesian space," in *Proc. Conference on Robot Learning (CoRL)*, Osaka, Japan, 2019, pp. 293–302.

[18] D. Coleman, I. A. Şucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a MoveIt! case study," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 3–16, 2014.

[19] R. Dillmann, "Teaching and learning of robot tasks via observation of human performance," *Robotics and Autonomous Systems*, vol. 47, no. 2-3, pp. 109–116, 2004.

[20] D. Lee and C. Ott, "Incremental kinesthetic teaching of motion primitives using the motion refinement tube," *Autonomous Robots*, vol. 31, no. 2-3, pp. 115–131, 2011.

[21] M. Simonič, T. Petrič, A. Ude, and B. Nemec, "Analysis of methods for incremental policy refinement by kinesthetic guidance," *Journal of Intelligent & Robotic Systems*, vol. 102, art. no. 5, pp. 1–19, 2021.

[22] M. G. Catalano, G. Grioli, E. Farnioli, A. Serio, C. Piazza, and A. Bicchi, "Adaptive synergies for the design and control of the Pisa/IIT SoftHand," *The International Journal of Robotics Research*, vol. 33, no. 5, pp. 768–782, 2014.