2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Macau, China, November 4-8, 2019

MPERL: Hardware and Software Co-design for Robotic Manipulators *

Marcus Pirron[1] and Damien Zufferey[2]

***Abstract*— Building small custom robots is getting democratized thanks to affordable tools like 3D printers and micro controllers. However, it still requires expertise from a wide range of domains: from designing the mechanical parts to writing the code that controls the robot. We present MPERL, a tool to help non-experts build custom robotic manipulators.**

MPERL starts from an abstract description of the robot's kinematic structure which contains information about joints, actuators, and sensors. The structure is refined with an easily manufactured geometry. Furthermore, from the structure MPERL generates control code which can be used to move the robot to a target configuration. We evaluate MPERL on a range of common robotic manipulator architectures, both serial and parallel.

I. INTRODUCTION

Advances in rapid prototyping and manufacturing technology have made building custom robots more accessible and affordable. Low-cost rapid manufacturing tools such as 3D printers and other CNC tools have simplified the creation of mechanical structures. Moreover, inexpensive micro controllers, sensors, and actuators can easily be added to produce functional robots. Custom robotic systems are not anymore limited to industrial settings which can afford high entry costs and high level of expertise. Motivated tinkerers can already build such robots on a limited budget. However, they still need to acquire expertise in multiple domains. As designing and constructing robotic systems draw from many different areas, domain specific knowledge for each of these steps is required: designing the electromechanical components, figuring out how to control the robot, and implementing the code.

We present MPERL (Multipurpose Parallel End effector Robotics Language) a tool that allows non-expert users to create and test robotic manipulators programmatically. These robots can then be manufactured with the help of 3D printers. We focus on simplifying the programming by automatically generating high-level motion primitives which can move a specific element of a robotic manipulators, typically the end effector, to a particular location. MPERL integrates elements from computational design and fabrication, inverse kinematic solvers, and feedback control.

MPERL's input is an abstract view of the general structure of the robot, i.e., what the different components are and how they are connected to each other. MPERL supports any element which generates a rigid motion and comes with default geometry which can be used for fabrication. On the software side, we first perform an analysis to estimate the workspace of the structure and find the singularities within or at the boundary of the workspace. Second, we generate the constraints corresponding to the structure in a form that can be handled by an inverse kinematic solver. Figure 1 summarizes the functionality of MPERL.

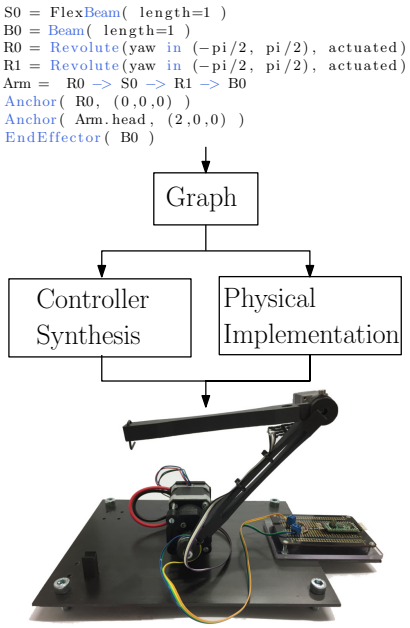


Fig. 1: MPERL workflow

A structure in MPERL starts with anchors to which other components are attached. Anchors have fixed coordinates and cannot move. The other components' positions are constrained by the rigid transformations between the components and the anchors. The components roughly fill the following roles:

- *actuators* can be controlled, e.g. motors;
- *sensors* are controlled by the environment but we can read the components' state;
- *passive* components are controlled by the environment and we cannot read the component's state.

The relative position of components is exposed with parameters which can be static or dynamic. For instance, the angle of an actuated revolute joint is a dynamic parameter while a beam is a prismatic joint with a static parameter.

The sensing elements can give feedback on the motors and

*This work was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (ERC Synergy Grant ImPACT).

[1]Max Planck Institute for Software Systems, Kaiserslautern, Germany, mpirron@mpi-sws.org

[2]Max Planck Institute for Software Systems, Kaiserslautern, Germany, zufferey@mpi-sws.org

also on the robot's overall structure. 3D printed structures are typically made of polymers and can be quite flexible. Origami style structures [1], [2] which have also been used to quickly build customized robots share the same problem. MPERL can embed deflection sensors in the structure at manufacturing time and when a load is applied to the system the sensor reports the deflection back to the controller. The controller can then account for the deflection and adapt the actuation to reach the current end effector target position.

This paper presents MPERL, a tool with a simple front-end language which makes it possible to design robotic manipulator without requiring expert knowledge. MPERL generates a physical implementation of robots and software methods to actuate them by solving the inverse kinematics.

In Section II, we discuss the related research which forms the context for this work. A complete example of the system is presented in Section III. In Section IV we present the core constructs of MPERL and explain how MPERL works. In Section V we present case studies based on common manipulator architecture. Our implementation is available at `https://gitlab.mpi-sws.org/mpirron/mperl`.

II. RELATED WORK

The low cost of rapid prototyping tools has led to a new generation of interactive tools and techniques [2], [3], [4], [5], [6] to help non-expert users create and optimize robotic designs. The closest work to MPERL is the robot compiler by Mehta et al. [1], [7] and the work by Ha et al. [8].

The robot compiler [1], [7] uses a high level structural description, along with a library of components, to generate manufacturable outputs. They target a much wider class of robots. However, on the software side, they generate only stubs to communicate with the hardware. The user has to implement the logic driving a robot. The robot compiler has been extended [9] to use reactive synthesis to generate a finite state machine software controller. While the robot compiler can generate complex temporal behaviors, it does not do complex calculation like inverse kinematics or finding the singularities of the system. MPERL targets only robotic manipulators but generates higher level software functionalities.

Ha et al. [8] use a library of components consisting of joints and links which, together with a user specified input trajectory, to generate robotic system following the input trajectory. Rather then trying to design a structure that achieves a given motion, MPERL generates the software infrastructure to achieve any motion allowed by the structure.

Inverse kinematics has been extensively studied [10], [11] and mature tools are available [12], [13], [14]. We are not developing any new analysis but work on integrating these techniques directly with the system which builds the robots. Available tools focus on serial chains, while the method we use may not be the most efficient, we want to also handle common parallel structures. We expect frequent use of parallel structures to compensate for the lack of rigidity of the 3D printed polymer structure.

Recent works in the area of modular robotics [15], [16], [17] have a similar goal automating the design and programming modular robots. Modular robotics creates new robots by composing predefined modules with known properties and capabilities. The modules, often build by experts, have standardized connections to enable automated generation of assemblies. Assemblies are automatically generated in order to accomplish specialized tasks. Our system is located on a lower level. We focus on composing electromechanical components to build robotic manipulators. We do not yet generate complex behaviors or control but only simple motions.

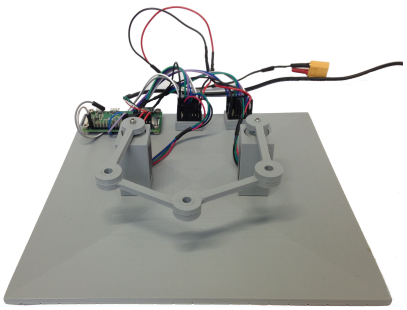


Fig. 2: Dual scara arm as generated by MPERL

Giusti et al. [15], [16] have developed a framework for modular manipulators. From an human demonstration of the task to accomplish, they generate a manipulator by search over the possible robotic assemblies. The search optimizes over both the hardware elements and the software controller. First, assemblies are filtered using kinematic constraints. Then, a controller is generated for each assembly which can accomplish the task. Finally, the assembly which accomplishes the task the fastest is selected.

Jing et al. [17] propose a system for modular self reconfiguring robots. The system consists of an high level mission planner, a design library with configurations and behaviors, and a design and simulation tool. Their robots are made of many copies of a single universal module and different behaviors are achieved through dynamic reconfiguration. Different configurations along with their capabilities have been characterized and the motion planner, given a specification, searches over the space of actions and configurations.

III. EXAMPLE

We use the example of a scara arm to illustrate how to use MPERL to design, manufacture and control robotic systems. Figure 2 shows the MPERL-generated robot. The robot consists of four interconnected 3D-printed beams, two of which are actuated by off-the-shelf stepper motors. Figure 3 shows the source code used to generate the robot structure as well as the control software. The code has been simplified for the sake of clarity.

A. Robot Design

The scara arm consists of four beams, which are connected to each other with revolute joints. The two extremities of the arm are connected to a base with revolute actuators. The revolute joint at the center of the arm is designated as the end effector. In Figure 3, lines 1-5 initialize the

```
B0 = Beam( length=1 )                                    1
B1 = Beam( length=1 )                                    2
R0 = Revolute(yaw in (-pi/2, pi/2), actuated)            3
R1 = Revolute( )                                         4
R2 = Revolute( )                                         5
Arm1 = R0 -> B0 -> R1 -> B1 -> R2                        6
Arm2 = Clone( Arm1, preserve=R2 )                        7
DualScara = Merge(Arm1, Arm2)                            8
Anchor( R0, (0,0,0) )                                    9
Anchor( Arm2.head, (2,0,0) )                             10
EndEffector( R2 )                                        11
```

Fig. 3: MPERL code for a dual scara arm

building blocks for one half of the robot — two beams with length 1 and three revolute joints. The angular motion of the actuator is restricted to the interval $[-\pi/2, \pi/2]$. If no specific constraints on the length or angle are given, default bounds are used. Line 6 uses the $\rightarrow$ operator to connect these elements into a chain. Since the arm is symmetric, line 7 duplicates one half and line 8 connects the two halves to form the dual scara arm. R2 is not duplicated, so the two chains meet there. R0 and the corresponding duplicated part are actuated, while the remaining revolute joints do not have any motors attached and follow passively. Lines 9-11 determine where the end effector is located and which parts of the robot arm are connected to the ground.

B. Robot Controller

The code in Figure 3 describes the structure of the robot. This description contains the necessary information to compute the kinematics (including singularities) of the robot. Both, forward and inverse kinematics are supported. MPERL generates an Actuate method which takes as argument a mapping from actuator parameters to values and a Move method moves the end effector to a position given as argument, thereby solving the inverse kinematic equations to get values for the revolute actuators.

C. Physical Implementation

Along with the controller synthesis, our system generates suitable 3D parts to manufacture the robot. Figure 2 shows a physical implementation with 3D printed parts. A Raspberry Pi controls the robot and a micro controller is attached to each motor. The micro controllers implement a standardized interface and are connected to a common message bus.

IV. THE MPERL LANGUAGE

A. Preliminaries

In the scope of this work, we consider only kinematic models. Positions are points in $\mathbb{R}^3$, displacements are vectors in $\mathbb{R}^3$. Mechanical components are modeled as rigid bodies and the relation between them are rigid motions. We use the special euclidean group $SE(3)$ for rigid movement. Rigid transformations preserve distance and orientation.

Elements of $SE(3)$ are represented by 4×4 homogeneous transformation matrices with multiplication as the group operation. Points extend to $(x_1, x_2, x_3, 1)^T$ and vectors to $(a, b, c, 0)^T$. We can therefore write: $\begin{pmatrix} R & r \\ 0_{1\times3} & 1 \end{pmatrix} \in \mathbb{R}^{4\times4}$ where $R \in \mathbb{R}^{3\times3}$, $r \in \mathbb{R}^3$, and R satisfies $R^T R = RR^T = I$ and $|R| = 1$. Matrices for common rigid transformations like rotations and translations can be found in text book [10].

We use $\mathbb{B}$ for the boolean values *True* and *False*.

B. Primitive Components

Robots are created by composing small building blocks (*Primitive Components*) together to form larger structures. These primitive components fall roughly into the following categories: anchors, beams, joints, actuators, and sensors.

Anchors are used to connect a robotic system to the world frame. An anchor specifies an absolute position and orientation. Furthermore, anchors cannot move and do not have any parameter.

Beams are static connections between components. Beams have static parameters, for instance, the length. Varying the parameters results in different robots. However, the parameters need to be fixed before the robot can be made. We use the term beam for any component that satisfy these conditions. So a fixed rotation is also a "beam".

Joints are connections with dynamic parameters. In the simplest form, the parameters are neither visible nor controlled. The position of the joint is constrained by the other elements in the system and, if under-constrained, the environment picks the joint configuration. If a motor is connected to the joint then it becomes an *actuator* and the controller can set the actuator's configuration. *Sensors* are joints which configuration can be read by the controller.

Abstractly, components are described by their signature, a triple (P, C, S) where P is a set of parameters, C is a set of constraints imposed upon the joint ($P \mapsto \mathbb{B}$), and S is a map from P to $SE(3)$ which describes the state of the component.

The set of parameters P consists of active parameters P_a and passive parameters P_p, i.e. $P = P_a \uplus P_p$. Active parameters can be set explicitly, e.g. the angular value of an actuator, and passive parameters are set by the environment. The value of passive parameters is implicitly existentially quantified.

The constraints model the limits on an element's motion, e.g. how much a revolute joint can turn. In general, constraints are predicates over the set of parameters. In particular, we assume that every parameter has an explicit lower and upper bound. These bounds are required to reason about robotic systems using automated solvers.

For example, given a component (P, C, S) and a target rigid motion T, finding whether the component can induce this motion reduces to solving: $\exists P.\, S(P) = T \wedge C(P)$.

We provide an initial sets of components including revolute, prismatic, and spherical joints. MPERL can be extended with user defined components.

C. Composing and Manipulating Elements

Primitive components are connected together to form a robotic system. These connections result in a graph of

interconnected components. A robotic system is modeled as labeled graph where the vertices are components represented by a local frame, the edges are the connections between components, and the labeling function maps the vertices to the type of the component (revolute, prismatic, etc.) and its properties (anchors, effector). A robotic system can contain other robotic systems; thus, MPERL can directly manipulate and compose whole graphs like the merge operation that we have seen in Figure 3.

MPERL has the following constructs to build robotic systems (RS):

```
RS ::= Primitive Component
     | RS -> RS
     | Clone RS
     | Decompose RS
     | Merge RS RS
```

The primitives are any joint natively supported by MPERL or provided by the user. To simplify the creation of more complex structures, we support the composition and decomposition of graphs into serial chains, and the cloning/merging of robotic systems.

Once a system is build, we can give specific roles to certain elements: anchor and end effectors.

```
Modifiers ::= EndEffector RS
            | Anchor RS Coordinate
```

Properties poll the robotic system and its components. For any system, we can get the parameters and the constraints acting on the part considered. If the system is a chain, we can also get the transformation matrix between the start and the end of the chain. Furthermore, we can get the Jacobian of that transformation.

```
Properties ::= TransformationMatrix RS
             | Jacobian RS
             | GetConstraints RS
             | GetParameters RS
```

Actions include commands for manipulating robotic system, most notably actuating parts (forward kinematics) and moving the end effector (inverse kinematic).

```
Action ::= Move RS Coordinate
         | Actuate RS Parameter Value
```

We discuss how the inverse kinematic is handled in the next subsection. It is worth mentioning that we can use the action commands during the development to simulate the robotic systems and in combination with the properties commands we can inspect the robot's state in specific configurations.

D. Generating the Software

The main benefit of using MPERL is not so much for realizing the physical structure but in getting some baseline software functionality implemented out-of-the-box. Not only can we generate commands to set some actuator values but we also provide the more user friendly move command where the user specifies a target position for the end effector and MPERL computes the actuators values to reach the target by solving the inverse kinematics.

1) Forward and Inverse Kinematics: For the graph structure of a robotic system, we can extract information to solve the forward and inverse kinematics. The forward kinematics computes the position of the structure given actuator values and the inverse kinematics computes the parameter values for a target configuration. The difficulty for these problems depends on the structure of the graph.

Chains, i.e., paths where all the vertices have degree 2 except the start and end with degree 1, are simpler to handle. Chains are also called serial structures. For graphs composed of a single chain, the forward kinematics is easy and boils down to matrix multiplication.

On the other hand, the inverse kinematics and both the forward and inverse kinematics for parallel structures are much more difficult. These problems cannot, in general, be solved analytically and we rely on numerical solvers and optimization tools for non-linear systems of equations. Below, we discuss how to find the parameters for a single configuration. For trajectories, we compute the configurations for points along the trajectories and interpolate the configurations between these points.

a) Preprocessing the Graph: The first step is to normalize and reduce the graph. This has two parts. (i) *Splitting the anchors.* As the anchors are fixed, having multiple elements connected to a single anchor is semantically equivalent to having these elements connected to their own copy of the anchor. This transformation may increase the number of vertices in the graph but preserves the edges. The goal of this splitting is to increase the serial parts of the graph. (ii) *Collapsing chains.* Sequences of joints compose nicely and can be simplified. Two joints (P_1, C_1, S_1) and (P_2, C_2, S_2) are simplified to $(P_1 \uplus P_2, C_1 \wedge C_2, S_1 \cdot S_2)$ removing the intermediate node altogether. In this step, the graphs for parallel structures may become multigraphs.

b) Serial Structures: We first explain how to deal with chains, i.e. serial structures. Later, we will use the kinematic equations for chains as the basis for handling parallel structures.

After normalization, if the graph simplifies to a chain we have two anchor nodes connected by an edge which is the composition of all the joints along the way. Let us call X and Y the two positions and orientations of the anchors and (P, C, S) the composed joint. The inverse kinematics amounts to solving $\exists P.\ X \cdot S(P) = Y \wedge C(P)$. Not every solver supports constraints of that form and we often transform the equality constraints into an optimization problem. $X \cdot S(P) = Y$ is transformed into minimizing $|X \cdot S(P) - Y|$. The constraints in $C(P)$ are also turned into distances and added to the objective function. The constraints have a solution iff the corresponding optimization problem has a minimum of 0. With a solution for the values of P, we keep only the values from the active parameters. Notice that the solution may not be unique. We will discuss later the limitations of this approach.

c) Parallel Structures: To handle parallel structures, we reduce the problem to finding a common solution for multiple chains at the same time. For the inverse kinematics, the

end effector gets a fixed position assigned and, therefore, it becomes an anchor in the graph. Splitting that node is often sufficient to turn the graph into a collection of chains. These types of architecture are *implicit chains*.

For systems which do not decompose into chains, the first step is to find a minimal set of simple paths which covers all the edges in the graph. We do this using a greedy algorithm. Assume that the graph get decomposed into n simple paths called P_i with $1 \leq i \leq n$. We use these path to generate the constraints.

First, for each path P_i which starts and ends with anchors, we generate the same constraints as for chains. Second, for each vertex v in the graph which is not an anchor, we generate constraints to make sure all the paths going through v agree on the position of that vertex. More precisely, for each pair of paths P_i, P_j going through v, we compute the partial path constraints from one of the anchor to v. Let us call these positions $P_i \downarrow v$ and $P_j \downarrow v$. The additional constraint $P_i \downarrow v = P_j \downarrow v$ makes sure the solutions for individual chains are consistent in the global structure.

Solving the inverse kinematics requires finding a solution to the conjunction of all the constraints generated in the two steps described above. Implicit chains are easier to handle for the inverse kinematics solver as each chain can be solved independently. Otherwise, we need to solve the constraints over the entire graph. While we generate a number of constraints polynomial in the size of the graph constraints solvers are exponential in the number of parameters.

d) Limitations: The kinematic equations include parameters which we cannot actuate. Therefore, actuating the system may not give the expected result. To warn the user of potential problems we compute the degrees of freedom using the Grübler-Kutzbach criterion and report an error if it does not match the number of active parameters. This approach can give false positives though. A second limitation is that we do not check for self-intersection. The kinematic model is generated from the abstract graph and the actual geometry is added later. Furthermore, we allow the user to provide a custom geometry for the components.

e) Parameter Synthesis: While the inverse kinematics is most often used to compute values for the dynamic active parameters of a robotic system, MPERL can also use inverse kinematics to automatically synthesize values for the static parameters of a robotic system. Given a graph where the static parameters of beams are not assigned and a list of target points. We can solve for values of the static parameters such that the system can reach all the target points with the same static parameters values.

Consider a serial scara (roughly one half of the arm in Figure 3), a target point of $(\sqrt{2}/2, \sqrt{2}/2, 0)$, unspecified lengths of the beams, and unspecified angular values of the joints. The move command returns a possible solution which contains possible values for the length of B0 and B1:

```
B0_length = 0.765;   B1_length = 0.999;          1
R0_yaw = 1.963; R1_yaw = -1.964; R2_yaw = 0;     2
```

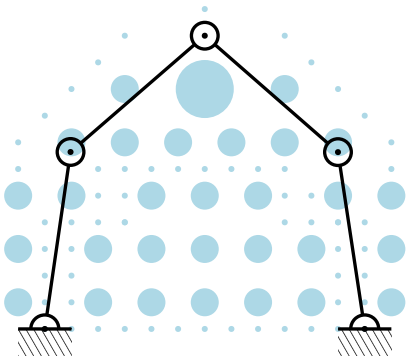


Fig. 4: Singularities for a dual scara arm. The size of the dots represent distance to singularities.

E. Workspace and Singularities

We follow trajectories by computing the joints configuration for points along the trajectories and interpolate between these configurations. This only works if the robotic system is not in a singular configuration, i.e. a configuration in which the system looses some degree of freedom and cannot operate as expected. These singularities happen at the boundary of the workspace, e.g. when a serial arm is fully extended (*boundary singularity*), or inside the work space, e.g. if two or more axis of motion are aligned (*internal singularities*).

MPERL uses Singular Value Decomposition to find the direction towards which the manipulator is most difficult to move, and therefore, to get the closeness to a singularity [18]. More precisely, we compute the Minimum Singular Value (MSV) [19], [20] as follows. The Jacobian of the robotic system can be written as $J = U\Sigma V^T$ where U is a $m \times m$ diagonal matrix, V a $n \times n$ diagonal matrix, and Σ the $m \times n$ diagonal matrix consisting of the singular values σ_i. The MSV is then given my $\min(\sigma_1, \ldots, \sigma_m)$.

During initial setup of the robotic system, we map the manipulator's workspace to find singular regions. If a trajectory passes close to or through a singular region, an error is reported. MPERL is also able to iteratively refine the map.

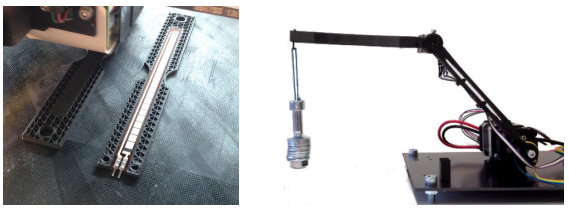
The singular regions are mapped by dividing the working space into a grid and computing the singular value decomposition at the grid's connecting points. When getting near a possible singularity, the grid can be refined only in this area. This has the same limitation as the inverse kinematics. We currently only support chain based robotic systems like the scara arms. As computing singularities is expensive, the computation is done once while the system is being designed and then stored. Then, when computing trajectories, we can lookup whether a trajectory traverses singular regions. Figure 4 shows the singularity map for the dual scara arm.

F. Sensors and Feedback

Sensors can by embedded in beams (see Figure 6a) or on actuators and become extra nodes in the graph. A sensor reports the error between the expected configuration and the current state. In the case of an actuator, this is the distance to the set point and, for structural elements, this is the

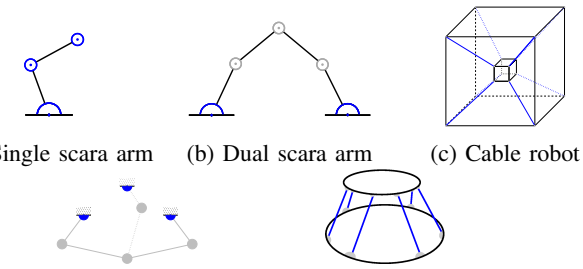
```
S0 = FlexBeam( length=1 )
B0 = Beam( length=1 )
R0 = Revolute(yaw in (-pi/2, pi/2),actuated)
R1 = Revolute(yaw in (-pi/2, pi/2),actuated)

Arm =  R0 -> S0 -> R1 -> B0
Anchor( R0, (0,0,0) )
Anchor( Arm.head, (2,0,0) )
EndEffector( B0 )
```

Fig. 5: MPERL code for a single scara arm with flex sensor



(a) Flex sensor embedded in beams during printing (b) Single scara arm where the lower beam includes sensors

Fig. 6: Embedding sensors in the robot structure



(a) Single scara arm (b) Dual scara arm (c) Cable robot

(d) Delta robot (e) Gough-Stewart platform

Fig. 7: Schematic representation of tested robotic systems. Blue denotes active components, gray passive components and black the remaining structural components

deflection. For example, if the sensed position is equal to the target position, the sensor's configuration matrix is the identity matrix I; otherwise it reflects the error. At runtime the sensors are polled, the corresponding transformation matrices are updated, and the inverse kinematics solver uses this information to update the actuation values.

Listing 5 gives the source code for a single scara arm, which automatically adjusts load induced deflection. The *FlexBeam* consists of a beam embedding a load cell. As soon as the system polls the sensor and notices a deviation, it automatically engages the actuated joint in R0 to counter the difference in position. Figure 6b shows the hardware implementation of the system. As soon as the weight is attached or removed from the arm, the position of the arm will change. The load cell senses the change in position and the system corrects for it by actuating the revolute actuator at the base of the arm. We currently use a simple proportional feedback controller as more complex controller such as PID are unstable in parallel structures. We leave the use of more advanced controller as future work.

V. IMPLEMENTATION

A. Implementation

We implemented MPERL in Python. The equations and constraints are manipulated using the Sympy library [21] and, to solve kinematics equations, we provide support for the dReal SMT solver [22], least squares optimization[1], and our own implementation of cyclic coordinate descent [23].

Nodes in the graph can be either actual physical devices or virtual nodes (loop devices). As an actual physical device, each actuated or sensing building block consists of a standardized interface, running on an Atmega micro processor and the component which does the acting or sensing, e.g. a motor or a rotary sensor. Communication between nodes are done using UART or SPI. As a virtual device, we provide an implementation as Python classes. It is also possible to mix virtual and physical classes, e.g. to have a simulation of a robotic system, which incorporates real-world sensor data. We support all lower kinematic pairs and also provide some support for higher kinematic pairs and wrapping pairs.

To generate physical parts from the structural description, we provide parameterized OpenSCAD[2] descriptions, which can be user modified, e.g. different beam designs.

[1]We use scipy.optimize.least_squares.

[2]https://www.openscad.org/

B. Evaluation

We test MPERL on the single and dual version of the scara system [24], a redundantly constraint Cable Robot [25] with 8 cables, a Delta robot [26], and a Gough-Stewart platform [27]. Figure 7 shows the overall structure of these examples.

Table I gives statistics about these examples. We report the size of the graph representation of the robotic systems and the number of parameters as measure of the complexity of the examples. Table I also reports the setup time of the example in MPERL, which is the time for the system to initialize a new system: building the graph, traversing and checking for constraints, and generating the kinematic equations (but not solving them). The setup is needed once or when the structure of the robot changes. We run MPERL on a single core of an Intel i7-6920HQ (2.9Ghz) with Debian Linux.

Table II shows the time to calculate forward and inverse kinematics for different robotic systems. We take the average for 20 different arbitrary but valid configurations (forward kinematics) and for arbitrary but reachable points in the working space (inverse kinematics). For the inverse kinematics, we compare the three solvers but we use only dReal for the forward kinematics.

Table III summarizes the time needed to compute the singularities. Starting from the center, the work space is evenly divided in each direction. For the single and dual scara arm, this results in an initial 81 cells, for the Cable Robot, Delta Robot and the Gough Stewart Platform, we sample 541 cells. Figure 4 shows the initial grid for the dual scara system. For each point, we calculate the singular value decomposition of the robotic system (serial chains),

TABLE I: Overview of examined robotic systems. $|P_p|$ and $|P_a|$ denotes the number of passive and active parameters

Robotic System	Type	Size of graph	$\|P_p\|$	$\|P_a\|$	Setup $[ms]$
Single Scara Arm	Serial	7 Nodes	2	3	316
Dual Scara Arm	Parallel	13 Nodes	8	2	641
Cable Robot	Parallel	40 Nodes	48	8	917
Delta Robot	Parallel	15 Nodes	4	3	612
Gough Stewart	Parallel	30 Nodes	36	6	1366

TABLE II: Evaluation of the kinematics solvers

Robotic System	Inverse kinematics $[ms]$			Forward kinematics $[ms]$
	dReal	LS	CCD	
Single Scara Arm	52	124	439	44
Dual Scara Arm	156	448	918	138
Cable Robot	594	955	825	217
Delta Robot	367	822	1287	177
Gough Stewart	516	755	640	273

which gives the closeness to singularities. If a target point is near a singularity, the cell containing can be further divided and the map gets more refined. Refinement steps add to the computation time but increase the usable work space with a better approximation of the singular regions

TABLE III: Computation of the singularity map

Robotic System	Initialization $[s]$	Refinement step $[s]$
Single Scara Arm	67	64
Dual Scara Arm	34	30
Cable Robot	621	615
Delta Robot	432	402
Gough Stewart	489	463

VI. CONCLUSION

We have presented MPERL, its capabilities, how to use it, and how it handles common manipulator architectures.

In the future, we plan to extend it and connect it to a wider ecosystem. We plan to support a wider range of components including legs and wheels taking inspiration from the work of Schulz et al. [2] and Ha et al. [8]. Currently, we work only with kinematic models and would like to consider dynamic effects as well. Finally, we plan to connect to PGCD [28], [29] a tool for coordinating the actions of multiple robots instead of working with a single robot at the time.

REFERENCES

[1] A. M. Mehta, J. DelPreto, B. Shaya, and D. Rus, "Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications," in *IROS*, 2014.

[2] A. Schulz, C. R. Sung, A. Spielberg, W. Zhao, R. Cheng, E. Grinspun, D. Rus, and W. Matusik, "Interactive robogami: An end-to-end system for design of robots with ground locomotion," *I. J. Robotics Res.*, vol. 36, no. 10, 2017.

[3] L. Zhu, W. Xu, J. Snyder, Y. Liu, G. Wang, and B. Guo, "Motion-guided mechanical toy modeling," *ACM Transactions on Graphics (TOG)*, vol. 31, 2012.

[4] B. Thomaszewski, S. Coros, D. Gauge, V. Megaro, E. Grinspun, and M. Gross, "Computational design of linkage-based characters," in *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2014)*, 2014.

[5] V. Megaro, B. Thomaszewski, M. Nitti, O. Hilliges, M. Gross, and S. Coros, "Interactive design of 3d-printable robotic creatures," *ACM Trans. Graph.*, vol. 34, no. 6, 2015.

[6] N. Bezzo, A. M. Mehta, C. D. Onal, and M. T. Tolley, "Robot makers: The future of digital rapid design and fabrication of robots," *IEEE Robot. Automat. Mag.*, vol. 22, no. 4, 2015.

[7] A. M. Mehta, J. DelPreto, and D. Rus, "Integrated codesign of printable robots," *J. Mechanisms Robotics*, vol. 7, no. 2, 2015.

[8] S. Ha, S. Coros, A. Alspach, J. M. Bern, J. Kim, and K. Yamane, "Computational design of robotic devices from high-level motion specifications," *IEEE Transactions on Robotics*, vol. 34, no. 5, 2018.

[9] A. M. Mehta, J. DelPreto, K. W. Wong, H. Kress-Gazit, and D. Rus, "Robot Creation from Functional Specifications," in *Springer Proceedings in Advanced Robotics*, 2017.

[10] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[11] S. R. Buss, "Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods," 2009.

[12] H. Bruyninckx, "Open robot control software: the OROCOS project," in *ICRA*. IEEE, 2001.

[13] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, 2010.

[14] A. Aristidou and J. Lasenby, "FABRIK: A fast, iterative solver for the inverse kinematics problem," *Graphical Models*, vol. 73, no. 5, 2011.

[15] A. Giusti, M. Zeestraten, E. İçer, A. Pereira, D. G. Caldwell, S. Calinon, and M. Althoff, "Flexible automation driven by demonstration: Leveraging strategies that simplify robotics," *IEEE Robotics & Automation Magazine*, vol. PP, pp. 1–1, 05 2018.

[16] A. Giusti and M. Althoff, "On-the-fly control design of modular robot manipulators," *IEEE Transactions on Control Systems Technology*, vol. 26, no. 4, pp. 1484–1491, July 2018.

[17] G. Jing, T. Tosun, M. Yim, and H. Kress-Gazit, "Accomplishing high-level tasks with modular robots," *Auton. Robots*, vol. 42, no. 7, 2018.

[18] G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*. Prentice Hall Professional Technical Reference, 1977.

[19] C. A. Klein and B. E. Blaho, "Dexterity measures for the design and control of kinematically redundant manipulators," *The International Journal of Robotics Research*, vol. 6, no. 2, pp. 72–83, 1987.

[20] V. Klema and A. Laub, "The singular value decomposition: Its computation and some applications," *IEEE Transactions on Automatic Control*, vol. 25, no. 2, pp. 164–176, 1980.

[21] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, 2017.

[22] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Automated Deduction - CADE-24*, vol. 7898. Springer, 2013.

[23] L. . T. Wang and C. C. Chen, "A combined optimization method for solving the inverse kinematics problems of mechanical manipulators," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 4, 1991.

[24] H. Makino, "Assembly robot," Patent US4 341 502A, 1979.

[25] J. Albus, R. Bostelman, and N. Dagalakis, "The nist robocrane," *Journal of Robotic Systems*, vol. 10, 1993.

[26] R. Clavel, "Device for the movement and positioning of an element in space," Patent US4 976 582A, 1985.

[27] D. Stewart, "A platform with six degrees of freedom," *Proceedings of the Institution of Mechanical Engineers*, vol. 180, no. 1, 1965.

[28] G. B. Banušić, R. Majumdar, M. Pirron, A.-K. Schmuck, and D. Zufferey, "PGCD: Robot programming and verification with geometry, concurrency, and dynamics," in *ICCPS*. ACM/IEEE, 2019.

[29] R. Majumdar, M. Pirron, N. Yoshida, and D. Zufferey, "Motion session types for robotic interactions (brave new idea paper)," in *European Conference on Object-Oriented Programming, ECOOP 2019*, ser. LIPIcs, A. F. Donaldson, Ed., vol. 134, 2019, pp. 28:1–28:27.