Generalized Programming of Modular Robots through Kinematic Configurations

Mirko Bordignon*, Kasper Stoy** and Ulrik Pagh Schultz**

Abstract— **The distinctive feature of modular robots consists in their reconfigurable mechanical structure, as they are assembled on-demand from basic mechatronic units. This implies that kinematic models of the robots need to be computed on a case-by-case basis for each specific assembly, which is a manual and hence time-consuming and error-prone procedure. We propose to automate this process by automatically computing such kinematic models starting from simple descriptions of the modules and their assemblies. This automated computation is supported by our toolchain for programming arbitrary modular robots in arbitrary configurations, presented in this paper.**

We contribute two novel results through this approach. First, a high-level programming language that provides kinematic abstractions for arbitrary modular robots, in contrast to the robot-specific solutions currently available. Second, a programming abstraction to subsume multiple kinematically equivalent robot assemblies into a so-called kinematic configuration, hence eliminating the need to explicitly enumerate and program each of them. These contributions advance current techniques for modular robot programming by demonstrating a tool that a) targets multiple mechanical platforms, offering the first general solution for modular robot programming, and b) raises the abstraction level by allowing users to reason and program in terms of standardized kinematic models that are automatically mapped to physical robot configurations by the toolchain.

I. Introduction

Modular robots are robotic devices assembled on-demand for the task at hand from sets of standard mechatronic building blocks. Such blocks are designed to be dynamically interconnected and detached, similarly to the pieces of a construction set [1]. In comparison with traditional fixed-topology robots, the trade-off consists in increasing versatility at the expense of efficiency. For instance, a robot arm assembled from generic actuator modules would not be as fast and precise as a traditional manipulator (a *specialized* robot), but the same modules could be reused to assemble a completely different kind of device, e.g. a mobile robot. This is arguably an advantage in contexts where versatility is an asset: for example, when it is difficult to anticipate all the possible tasks that might need to be fulfilled and it is impractical, or impossible, to accordingly procure specialized devices on-demand. This is the case in contexts such as outer space and submarine environments, as well as mines or any other place remote or difficult to access. Modular robots can thus serve as mechatronic toolkits for rapid prototyping and deployment of robotic solutions.

A number of hardware platforms and of control strategies have been proposed to pursue this vision [2]. Programming tools bridging the gap between the hardware and control, allowing fast implementation and transfer of controllers to the robot hardware, have however been developed in support of specific systems: Zhang et al. [3] proposed an XML-based scripting language to implement locomotion gaits on chain-type modular robots. Ashley-Rollman [4], De Rosa [5] and coauthors developed declarative languages demonstrated on simulated Claytonics ensembles. We addressed the problem of programming ATRON modules through the DynaRole language and its runtime environment, the DCD Virtual Machine [6], [7]. Nevertheless, no existing method has so far solved the general problem of how to program robots built from arbitrary modular systems. This makes it difficult to transfer results between different robot platforms and causes very limited code reuse, hence negatively influencing the cost and time needed for the development and deployment of robotic systems [8]. In contrast, on other types of robots like e.g., mobile or service robots libraries and application code can be reused, usually through middleware-based solutions that offer standardized interfaces to the underlying hardware [9]: it is well-known how to incorporate information about the mechanical design into the control software for specialized robots, and to create abstractions over such fixed design to promote code portability.

The development of a general method to program modular systems is however hindered by the variable mechanical structure of the assemblies, and further complicated by the different kinematic properties that the modules of each robotic system exhibit. This makes it particularly difficult to exploit mechanical models and incorporate them in control software in terms of abstract interfaces, as such models need to be computed on a case-by-case basis. We have however recently demonstrated a method to automate the generation of such models [10]. In this paper we present two major contributions deriving from its further integration within our software toolchain for modular robot programming. First, we extend the applicability of a module-specific programming language to arbitrary modular robots, demonstrating the first such language targeting multiple modular robotic platforms. Second, we introduce a new programming abstraction to subsume multiple kinematically equivalent robot assemblies, i.e., physically different robot configurations that are kinematically congruent due to symmetries in module design, into a so-called *kinematic configuration*. In addition to offer-

This work was supported by the Danish Council for Independent Research.

*Current affiliation: Digipack Industrial Automation, Italy. Based on work performed while at the Modular Robotics Lab, University of Southern Denmark. mirko.bordignon@ieee.org

**Modular Robotics Lab, Maersk Mc-Kinney Moller Institute, Faculty of Engineering, University of Southern Denmark, Denmark. {kaspers,ups}@mmmi.sdu.dk

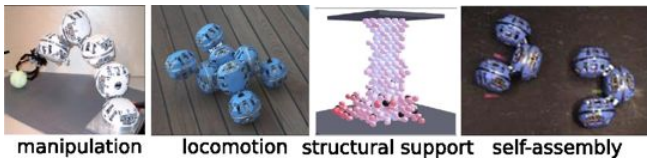
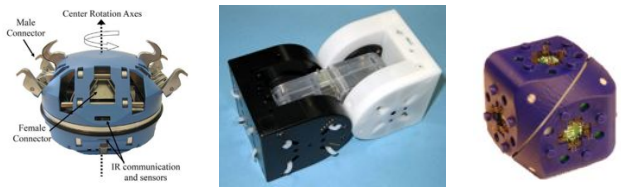


Fig. 1. Examples of several tasks performed by robots built from ATRON modules [13], [14]



(a) ATRON (b) M-TRAN (c) Molecube

Fig. 2. Three different module designs

ing a convenient high-level expressive means to users, this contribution also solves the known problem of identifying and controlling equivalent robot configurations [11], [12]. The rest of the paper proceeds as follows. First, Sec. II reviews M3L, a modeling language that can be used to describe the fundamental kinematic properties of a modular system. Then, Sec. III shows the first part of our contribution, how a programming tool for modular robots is extended to arbitrarily defined systems through M3L, and Sec. IV presents the second part of our contribution, new language features used to define what we call kinematic configurations, with which an abstract kinematic model can be superimposed to the specific robot hardware. Last, Sec. V gives a complete example demonstrating our contributions, and Sec. VI presents our conclusions and compares to related work.

II. M3L: Kinematic Modeling of Modular Robots

As previously mentioned, the key aspect of modular robots is their reconfigurable mechanical structure, which can be modified to suit different tasks (Fig. 1). Accordingly, a significant portion of modular robotics research focuses on the mechanical design of the individual modules, exploring the tradeoffs between their manufacturing complexity, the properties of the structures that can be assembled, the possibility to self-reconfigure, etc. This results in a number of module designs that differ in terms of joints and interconnection possibilities, which in turn make it possible to build structures with different properties. For example, robots assembled from each of the module types in Fig. 2 will have adjacent joints whose axes are orthogonal (ATRON), parallel or orthogonal (M-TRAN), parallel, skew or coplanar (Molecube). This depends on the relative placement of connected modules, determined in turn by the specific connection established. More formally, let us consider a single module as a mechanism, i.e., a set of rigid bodies (links), each with an associated frame of reference, whose motion is mutually constrained by kinematic couplings (joints) established between pairs of links, with the frames of the pair's links related by a transformation dependent on the joint type and instant value (e.g. the joint angle). An assembled robot is therefore a mechanism whose links are constrained by such joints and by the fixed transformations induced by inter-module connections, which rigidly set the relative position of the connected links of neighboring modules. It follows that by knowing the geometry of the links, the type and value of the joints, and the fixed transformations induced by the connections, we obtain a complete kinematic description of the assembly. This can be used to calculate the position of each link in a global robot frame given the joint values (forward kinematics), or for example to reason about the structure in terms of relative displacement of the joint axes, its mobility (no. of DOFs), etc. In general, obtaining the kinematic description would be useful when controlling a robot assembled from modules, since we can use it to compute global information about the structure similarly to how this is done for conventional, fixed-topology robots. Furthermore, being able to treat robots assembled from different types of modules under the same formalism is necessary in order to establish a single method for their control, be it in the form of a programming language or of another tool.

An important reason preventing the usage of kinematic and, in general, mechanical models for the high-level control of modular robots is that while the links and joints of the modules can be easily enumerated, the transformations induced by their connections can not. For example, there are 144 possible connections between two Molecube modules, since each of the 6 genderless connectors of a module can be mated to each of the 6 of the second module in 4 possible ways. It is clearly tedious and error-prone to compute those transformations manually, and even then one would need to set up a system to make proper use of the transformations relative to connections together with those induced by the joints in order to calculate the global description. Furthermore, this procedure would then need to be repeated for every different modular robotic system to be modeled. We instead took a different approach relying on model-based code generation: we developed a software tool which takes as input simple descriptions of a modular system and of a robot assembled from it, computes the relevant kinematic information and generates the relative code for use on robot controllers and in simulation. The details about this tool, named the Modular Mechatronics Modelling Language (M3L), can be found in [10], [15]. We briefly overview it here as it is part of the toolchain described in the next section. We use as a running example the M-TRAN system and robot definitions, reported at the end of the paper in Fig. 8.

A. Modeling modules, connections and robots

M3L models comprise *module*, *connection* and *robot* definitions. The essential features of a module are its *links* and *joints*. Links group together multiple *points*, i.e., features of interest that are rigidly interconnected: these could for example be the connectors present in a link. Furthermore, a *hull* can be defined to approximate the physical geometry of a link. The M3L code in Fig. 8 models the M-TRAN module, depicted in Fig. 2(b): as we can notice, rather than to provide e.g. a full, high fidelity CAD-like model, the idea

is to capture its kinematic features. For an M-TRAN module, this amounts to defining its three links (the two cuboids and the bar connecting them) and the two joints coupling each of the cuboids to the bar. Please note how the coordinates can be conveniently defined within an ideal single frame of reference, i.e. as if just by looking at the module "as it is". The definition of connections is the most convenient feature of M3L: instead of explicitly enumerating all possible ways to interconnect two modules (72 in case of M-TRAN), users can define them declaratively by implicitly listing the conditions for a connection to take place. Such conditions are used to unambiguously identify the transformation associated with the connection. The constraint solving algorithms used by the M3L compiler are detailed in [10]: in short, it consists of generating a set of candidate connections, exploiting the given conditions to identify their transformations through a constraints inference mechanism, and pruning those not satisfying the set of initial conditions. This results in a system able to correctly identify the parameters of a modular system from a succinct list of conditions, considerably improving the ease of use. We successfully modeled ATRON, M-TRAN and Molecube modules with M3L: in all cases a description similar in length to the one of Fig. 8 was sufficient (the full models are available in [15]). Once a module and its connections are defined, users can describe an assembly simply by instantiating a number of modules and listing the connections between them, akin to a sequence of steps to physically assemble the robot. Two examples of M3L robots are reported in Fig. 9. Please note that an optional numerical argument is used to select a specific connection from a compiler-generated list, in case more than one is possible between a couple of connectors and the user wants to override the default choice.

B. Model-generated code

After parsing the input models and running the constraints solver to compute the kinematic model, the M3L compiler emits code elements for use in three different contexts: (1) simulation worldfiles for Webots[1] with the modeled robot assembled as prescribed; (2) C components to be used by the DCD Virtual Machine; and (3) an XML file used by the DynaRole compiler to tailor the language to the simulated modules. Regarding the first generated element, it is convenient to generate Webots simulations from the model. First, the specification for single modules need not be written manually, which is advantageous to users without experience in writing the VRML models that the simulator uses. Second, the initial position within the scene of each module of the assembly is automatically determined by exploiting the computed transformations. The automatic positioning facilitates working with modular robots in Webots, since each module would otherwise require manual positioning laboriously computed by composing the transformations from a root module. The C-based execution environment of DCD and

[1]A widespread commercial robot simulator available from Cyberbotics Ltd. `http://www.cyberbotics.com/`

```
role Head extends Module {
 require self.center == $NORTH_SOUTH;
 ...
}
...
role RightWheel extends Wheel {
 require self.connected($WEST) == 1;
 ...
}
```

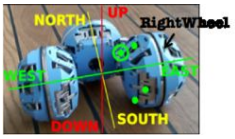


Fig. 3. Role identification through spatial structure constraints in DynaRole (extract from [6])

Dynarole work with these generated worldfiles, facilitating experiments on robots which are not physically available. The details of how Webots simulations are generated can be found in [10]: the simulated world used in Sec. V is generated exactly in this way.

The second and third generated elements are what enable us to extend DynaRole to robots arbitrarily defined using M3L, as illustrated in the next section.

III. Programming Arbitrary Module Types

The combination of a virtual machine and a high-level role-based programming language enables the developer to quickly and efficiently program ATRON modules, as demonstrated using the DCD-VM and DynaRole language [6]. In this paper, we present the generalization of DCD-VM and DynaRole to support arbitrary types of modular robots, building on M3L to generate the module-specific implementation of the virtual machine and compiler from a kinematic module specification. We first analyse the underlying problem in supporting arbitrary modules and then present our solution.

A. Problem analysis

A key DCD-VM feature is the automatic calculation of a module's position within an assembly using relative spatial information received from neighboring modules [16]. The DynaRole language in turn directly supports role-based control [17] by allowing programmers to specify what behaviors are to be activated and in which modules based on this spatial information [6]. Concretely, we use the orientation of the joint axis of an ATRON module together with the number of connected neighbors and their relative direction to declaratively assign functional roles to physical modules. This approach is illustrated in the example program of Fig. 3 where roles for a simple vehicle are assigned based on spatial properties. The user declares the minimal number of spatial constraints needed to assign roles in the given configuration, which is simpler than recognizing arbitrary configurations, and is the approach taken throughout this paper.

The DCD-VM and Dynarole were however limited to being used with the ATRON modular robot since the VM component performing the spatial computations and certain high-level language keywords (e.g., `center`) were manually defined based on the ATRON. For instance, when spatial information was received from a neighboring module, the local spatial information was updated according to hand-written rules that maintained a mapping between the physical connectors and a standardized numbering imposed by the global reference frame to all modules in that specific lattice

Fig. 4. An M-TRAN assembly resembling a quadruped, from `http://unit.aist.go.jp/is/frrg/dsysd/mtran3/`

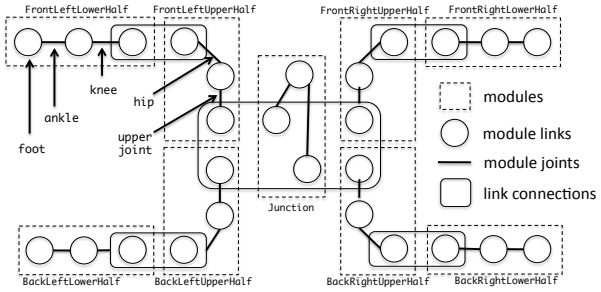


Fig. 5. Composition of the quadruped in Fig. 4, with the roles and labels.

position [10]. This approach has two major problems. First, it is difficult to generalize to other modular robots, as it relies on ATRON features like the presence of only one joint and specific symmetries of its connectors. Second, it worked only for lattice-like modular robots and only for lattice structures (e.g., not for a snake-like ATRON assembly).

B. Solution: M3L-based toolchain customization

Our key insight in DCD-VM and DynaRole is to obtain global geometrical information about the assembled robot and to exploit it to program the robot from a high-level perspective. The kinematic description of the assembly derived from M3L provides the complete global information that we need (for control purposes, in our case). The model-based generated components of the DCD-VM serve this purpose by performing a process akin to a distributed calculation of the robot forward kinematics. An arbitrarily chosen root module broadcasts the transformation which relates it to a global frame. Such a frame can be chosen arbitrarily: it can be the frame of one of the links in the root module, as is usually the case, or it can be an external reference measured through sensors. Connected modules receive and compose the transformation with the one induced by the connection (which was statically computed by M3L from the model) and with those induced between intra-module links by the joints (which are a function of joint values). Every module repeats the process by sending its global transformation to the neighbors for them to likewise compose it and obtain their global transformation, thus establishing a single coordinate system in the whole robot. Cycles and shared successors are handled implicitly by relying on continuous transmission of transformations at regular intervals, similarly to how we propagate state during self-reconfiguration [7]. Alternatively, an underlying communication layer could form a spanning tree or a gradient field which can be used to propagate the transformation information.

Concretely, the calculation is performed by C code: the M3L compiler generates constants for the coordinates of geometrical features and for the fixed transformations induced by intermodule connections, and it generates functions to compute intra-module transformations (see [10], [15]). The C code requires floating point support but otherwise has no specific architectural dependencies, making it suitable for practically all resource-constrainted microcontrollers. Specific constraints or different programming languages can easily be supported by implementing more backends for the compiler. Thanks to model-based generated code we thus obtain a computational substrate providing global information about the robot, which in turns supports DynaRole queries on its spatial structure. An XML file generated by M3L extends the DynaRole language by exporting the links and joints of a module as keywords. For instance, the position of Joint 1 of an M-TRAN module, defined in the M3L model (Fig. 8), can be set from within a DynaRole program by `self.JOINT_1_SET_POSITION(90)`. (Please note that we use uppercase notation as a convention to distinguish model-generated keywords from the standard DynaRole ones). Making the DCD-VM able to perform spatial computations on arbitrary modules and extending DynaRole with the keywords needed to control them is enough to be able to use these tools, originally developed for ATRON, on other modular systems. However, in order to truly exploit the expressive power that the mechanical model offers, we needed to add a few more features to DynaRole, as documented next.

IV. Arbitrary Kinematic Configurations

A. Problem analysis

The ATRON mechanical design has two features that inherently simplified the DynaRole abstractions used to describe robot configurations of interest. First, each module has only one joint, therefore in order to identify a joint within an assembled robot it is sufficient to assign a role to the module containing it. Second, the two links connected by the joint are identical and symmetrically placed, so that actuating the joint in a certain direction (i.e., clockwise or counterclockwise) is invariant under a 180° rotation of the module. To appreciate both these features, we can consider the three-modules vehicle of Fig. 3. Due to the first feature, a DynaRole program can define the `LeftWheel` and `RightWheel` roles for the left and right modules, and then simply use their joints as the wheel actuators without having to specify further conditions. In this sense, on ATRON the concepts of module and joint are conflated in one entity. Due to the second feature, we can observe that it is not important whether the wheel is attached to the rest of the structure through its north or south link, since in both cases we can operate the actuator in a certain direction, e.g. clockwise for the right wheel, and the effect will be the same, that is, the wheel will make the vehicle proceed forward. In contrast, we

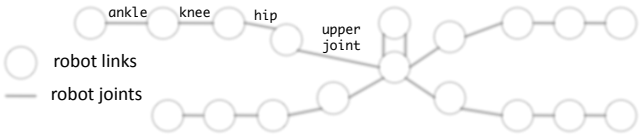


Fig. 6. Kinematic graph of the quadruped in Figure 4, derived from the graph in Figure 5 by considering multiple links connected by rigid inter-module connections as a single link

can examine the M-TRAN assembly resembling a quadruped in Fig. 4. Each leg is formed by two modules, which we can call `UpperHalf` and `LowerHalf`. Provided that we identify the modules corresponding to the two roles within each leg, a first problem is to map functional joints to their physical counterparts. For example, let the `ankle` be the joint connecting the link touching the ground (i.e., the `foot`) to the rest of the structure. The existing DynaRole spatial abstractions (direction of joint axes and number of connected modules) are not sufficient to map the ankle to one of the two joints within the LowerHalf. In other terms, differently from ATRON there is no one-to-one correspondence between modules and joints to be implicitly exploited. Furthermore, even if we identified the physical joint to operate when actuating the ankle, operating it consistently to produce a certain movement relatively to the rest of the structure (e.g., moving the foot backwards) is not invariant over multiple physical configurations resembling the pictured one. This is due to the fact that the foot cuboid and the connecting bar are neither identical nor symmetrically placed with respect to the joint. Given the same actuator input, the movement of the foot with respect to the bar is inverted when connecting the LowerHalf module to the structure in the two possible ways mapping the same physical joint to the ankle (see Fig. 4).

B. Solution: labels and kinematic conventions

The shortcoming highlighted by the M-TRAN quadruped consist in the fact that we can map functional to physical entities only at module granularity, and in the absence of methods to define kinematic conventions between functional parts. We can address the first problem by exploiting *labels*, that we first envisioned in [18] and now implemented. The idea is to name module components and then to use them in the control logic instead of referring to e.g. specific joints. In this way we can establish a functional mapping within the single modules. Let us analyze how the quadruped is composed by discussing the graph in Fig. 5. We can first distinguish the two modules composing each leg as a `LowerHalf` and an `UpperHalf` modules, and conventionally assign the name `Junction` to the module connecting the left and right legs. The functional parts of interest to be abstracted in order to control the assembly are highlighted in Fig. 5: within each leg, there are four joints that we want to identify based on invariant properties on the kinematic structure. By considering the structure of the overall robot pictured in Fig. 6, we can observe that, in addition to its belonging to the `UpperHalf` or `LowerHalf` half of a leg, each joint edge has a specific number of adjacent joints, i.e. number of edges sharing a common link vertex with it: 1 for the ankle joint, 2 for the knee and the hip, 6 for the upper joint. Please note that the bar of the central M-TRAN module is not directly connected to the legs, and is therefore not part of the link resulting from their connection. We can exploit the adjacency property and hence identify the joints of interest independently from the specific physical configuration of the assembly. For the quadruped, within the lower halves of each leg we can map the relevant joints to ankle and knee using the new `label` type and the `self.joints` function, whose first argument is the orientation in the global frame of the joints that we want to identify and whose second argument is the number of adjacent joints that they are required to have:

```
label ankle = self.joints($ANY,1);
label knee = self.joints($ANY,2);
```

Similarly, within the upper halves definition we can identify the hip and the upper joint as:

```
label upper_joint = self.joints($ANY,6);
label hip = self.joints($ANY,2);
```

Details on how labels are implemented can be found in [15]. We can now write statements like:

```
self.jointRotateDegrees(knee,90);
```

However, this solves only the first of the two problems that we detected.In order to define kinematic conventions between functional parts, we can extend labeling to links by exploiting a second property evident from the graph in Fig. 6: the different degree of each link vertex (i.e., the number of joints connected to it). Within a lower half module we can thus declare the foot as:

```
label foot = self.links(1);
```

Given the ability to identify a joint (the ankle) and the link connected to it (the foot), we can define the joint axis in the global reference frame:

```
self.setJointAxis(ankle, foot, $EAST_WEST);
```

This statement indicates that when the axis of the ankle joint is oriented in the prescribed direction, positive joint values correspond to a positive displacement of the foot link. Since the joint is revolute, this means that actuating it with positive values has to rotate the foot counterclockwise around the defined direction. Relating this convention to the actual values needed to operate the physical actuators is possible given the M3L joint definition: as we can see from Fig. 8, this contains the axis direction around which each link coupled by the joint rotates counterclockwise when the actuator is fed positive values, and the compiler actually checks that the two axes are collinear and opposite. In this sense, M3L is a language for description of mechatronics toolkits, since it is used to model the mechanics but also the control electronics operating it. (Again, readers can refer to [15] for details.) Being able to identify the joint and links of interest and to set kinematic standards to operate the joints enable us to write a DynaRole program for the M-TRAN quadruped, as shown in the next section.

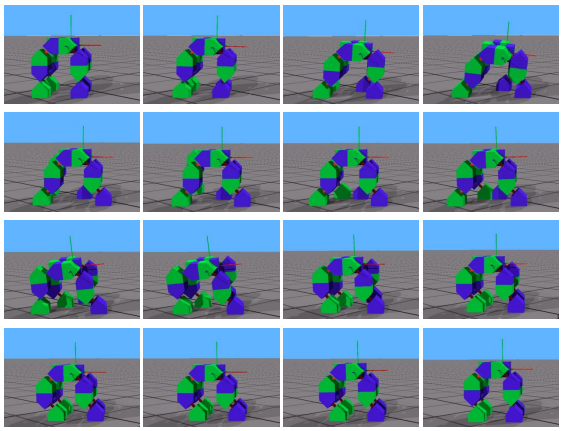


Fig. 7. The simulated quadruped performing a single gait cycle

For robots where many joints share the same neighbor configurations, such as a snake configuration, the role identification feature of DynaRole can be used to incrementally assign roles to each module in the robot, based on the state of its neighboring modules, e.g. starting with a "head" acting as origin, then giving all remaining modules the role of either a "body segment" or a "tail module" [17], [6].

V. Complete Example: M-TRAN quadruped

We now provide an example of using the complete toolchain to program a modular robot. The same toolchain has been used to program simulated [16] and physical ATRON modules [6], we now use it to program simulated M-TRAN modules. Concretely, the M3L model of Fig 8 is used to automatically customize the DCD-VM and DynaRole compiler to enable programming the quadruped-like M-TRAN assembly of Fig. 4, which we simulate using Webots (Fig. 7). Specifically, we combine the following DynaRole features: (1) the identification of modules through roles using spatial constraints; (2) the identification of joints within modules through labels using connectivity properties; and (3) the establishment of kinematic conventions using labels and spatial constraints. For our experiments, we use two different robot models both resembling the quadruped, listed in Fig. 9. First, please note that through the use of the `frame` field we fix the global reference frame to the passive link of module `m5`: the three reference axes, centered in the middle module, are also visible in the simulation (Fig. 7). Please also note that in order to use the DCD-VM and therefore DynaRole with Webots, the M3L compiler generates wrapper functions to interface with the Webots-provided C libraries.

First, we define roles for each module. The central one, connecting the left and the right side which we call `Junction`, is the only one connected to four other modules:

```
role Junction extends Module {
  require (sizeof(self.connected) == 4);
}
```

The upper halves of each leg are all connected to three modules, two in one direction and one in another. We therefore define an abstract role encapsulating this information, together with the label definitions that we will use later:

```
abstract role LegUpperHalf extends Module {
  connected_2_dir;
  connected_1_dir;
  label upper_joint = self.joints($ANY,6);
  label bar = $LINK_CONNECTING_BAR;
  label hip = self.joints($ANY,2);
  require (sizeof(self.connected(connected_2_dir))==2);
  require (sizeof(self.connected(connected_1_dir))==1);
}
```

We then define four roles extending `LegUpperHalf`, one for each leg (for simplicity, only one is shown here):

```
role FrontLeftUpperHalf extends LegUpperHalf {
  connected_2_dir = $SOUTH;
  connected_1_dir = $EAST;
}
```

The lower halves are connected to one module and include the ankle and knee joints:

```
abstract role LegLowerHalf extends Module {
  label foot = self.links(1);
  label bar = $LINK_CONNECTING_BAR;
  label ankle = self.joints($ANY,1);
  label knee = self.joints($ANY,2);
  require (sizeof(self.connected) == 1);
}
```

We can precisely define each lower half due to its connection to the corresponding upper half:

```
role FrontLeftLowerHalf extends LegLowerHalf {
  require
   (sizeof(self.connected($ANY,$$FrontLeftUpperHalf))==1);
}
```

Having identified the nine modules of interest and having appropriately labeled joints and links as explained Sec. IV, we can now proceed with the third part, i.e. the definition of kinematic conventions. For upper halves this amounts to:

```
  startup standup() {
    self.setJointAxis(upper_joint, bar, $EAST_WEST);
    self.setJointAxis(hip, bar, $EAST_WEST);
  }
```

and analogously for the lower halves:

```
  startup set_joint_axes() {
    self.setJointAxis(ankle, foot, $EAST_WEST);
    self.setJointAxis(knee, bar, $EAST_WEST);
  }
```

Having identified the joints of interest and having set standard operational conventions, our DynaRole program identifies a kinematic configuration, which the DCD-VM maps to the physical robot and its actuators. We can now exploit it to implement, for instance, a walking gait. We first define utility commands on the legs to be used as building blocks for the gait. On the upper halves we write a sequence which raises the thigh while keeping it normal to the ground in order not to unbalance the robot. This sequence will be used in the gait to temporarily lift the leg, so as to allow the lower half to move:

```
  command move_thigh() {
    self.jointRotateDegreesNonBlocking(upper_joint,
        raise_thigh_deg);
    self.jointRotateDegrees(hip,raise_thigh_deg);
    self.jointRotateDegreesNonBlocking(upper_joint,
        lower_thigh_deg);
    self.jointRotateDegrees(hip,lower_thigh_deg);
  }
```

(The specific values to operate the joints differ for front/back and left/right legs and are thus specified as fields in the

subclasses). Please note the use of `NonBlocking` variants: while the standard function to set the actuators blocks until the movement has been performed, the non blocking version allows us to actuate multiple joints at the same time.

On the lower halves we define three commands: one to project the robot forward by bending the knee and the ankle; one to move the foot forward, and one to recover a straight posture after a single gait cycle:

```
command from_straight_to_proj_forward() {
  self.jointRotateDegreesNonBlocking(knee,-45);
  self.jointRotateDegreesNonBlocking(ankle,45);
}
command move_foot() {
  self.jointRotateDegreesNonBlocking(knee,90);
  self.jointRotateDegreesNonBlocking(ankle,-90);
}
command from_proj_backward_to_straight() {
  self.jointRotateDegreesNonBlocking(knee,-45);
  self.jointRotateDegreesNonBlocking(ankle,45);
}
```

Last, we can define the `gait` behavior, reported in Fig. 10, which invokes the command defined on the leg modules and results in the quadruped walking with the gait pictured in Fig. 7. The complete program, omitted here for space reasons, can be found in [15].

In order to verify that the program is indeed working in the same way for a physically different but kinematically equivalent configuration, we tested it on both the robots listed in Fig. 9. Inspecting the connections list of the two robots, we can notice that the left and right parts of the quadruped are swapped with respect to the junction, and that one robot has two of the lower leg halves connected with a further 180° rotation (as previously stated, the optional numerical parameter after the connector couple explicitly specifies a connection other than the default one). As expected, the simulated robot showed the same behavior: the kinematic configuration, defined in the DynaRole program through joints identification and setup of kinematic conventions, is correctly mapped to the two different topologies thanks to the indirection layer provided by the DCD-VM. This solves the problem outlined in [11] and [12], i.e., devising a way to transparently control physically different but kinematically *congruent robot topologies* (as [11] calls them) in order to perform a given task.

VI. Perspectives

To the best of our knowledge, the software toolchain presented in this paper is the first one providing a high-level programming language that is applicable to arbitrarily defined modular robots: other existing solutions, as reviewed in Sec. I, are tailored to specific robots. The core idea enabling the generalization of our toolchain to different platforms is the incorporation of explicit mechanical modeling. Specialized robots have fixed mechanical structure and therefore their control software can implicitly incorporate it, be it that of a mobile base or a robot arm. The single modules composing a modular robot have fixed mechanical structure as well, so by implicitly incorporating such structure and explicitly modeling the interconnections, software can be made portable across different assemblies, provided that they are made from the same type of modules. By making mechanical modeling a first class concept in the toolchain, we made software portable both across different assemblies and across systems with different basic modules, achieving truly general applicability. Furthermore, mechanical modeling allowed us to indirectly solve two long-standing problems in modular robotics research, namely the recognition of specific robot configurations and the control of functionally equivalent configurations. Park et al. [12] tackled the first problem by matching configurations detected at runtime to known ones through three different methods. The proposed methods however are too computationally expensive to run on the resource-constrained devices on-board the modules, limiting the scope of applicability of the methods, and are furthermore not integrated within a software toolchain. Everist et al. [11] proposed a theoretical framework to adapt gait tables to physically different but functionally equivalent ("congruent") robot configurations. Again, the method is computationally expensive and is not directly usable as a means of programming.

References

[1] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]," *IEEE Robotics and Automation Magazine*, vol. 14, no. 1, pp. 43–52, March 2007.

[2] K. Stoy, D. Brandt, and D. J. Christensen, *An Introduction to Self-Reconfigurable Robots*. MIT Press, 2010.

[3] Y. Zhang, A. Golovinsky, M. Yim, and C. Eldershaw, "An XML-based Scripting Language for Chain-type Modular Robotic Systems," in *Proc. 8th Conf. on Intelligent Automous Systems (IAS-8)*, Amsterdam, The Netherlands, March 10-13 2004, pp. 729–738.

[4] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A Declarative Approach to Programming Ensembles," in *Proc. 2007 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07)*, San Diego CA, USA, October 29 - November 2 2007, pp. 2794–2800.

[5] M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, "Programming Modular Robots with Locally Distributed Predicates," in *Proc. 2008 IEEE Int. Conf. on Robotics and Automation (ICRA'08)*, Pasadena CA, USA, May 19-23 2008, pp. 3156–3162.

[6] M. Bordignon, K. Stoy, and U. P. Schultz, "A Virtual Machine-based Approach for Fast and Flexible Reprogramming of Modular Robots," in *Proc. 2009 IEEE Int. Conf. on Robotics and Automation (ICRA'09)*, Kobe, Japan, May 12-17 2009, pp. 4273–4280.

[7] U. P. Schultz, M. Bordignon, and K. Stoy, "Robust and reversible execution of self-reconfiguration sequences," *Robotica*, vol. 29, pp. 35–57, 2011.

[8] D. Brugali and E. Prassler, "Software Engineering for Robotics," *IEEE Robotics and Automation Magazine*, vol. 16, no. 1, pp. 9–15, March 2009.

[9] R. T. Vaughan, B. P. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proc. 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03)*, Las Vegas NV, USA, October 27-31 2003.

[10] M. Bordignon, U. P. Schultz, and K. Stoy, "Model-based Kinematics Generation for Modular Mechatronic Toolkits," in *Proc. 9th ACM SIGPLAN/SIGSOFT Int. Conf. on Generative Programming and Component Engineering (GPCE'10)*, Eindhoven, The Netherlands, October 10-13 2010.

[11] J. Everist, F. Hou, and W.-M. Shen, "Transformation of Control in Congruent Self-Reconfigurable Robot Topologies," in *Proc. 2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'06)*, Beijing, China, October 9-15 2006, pp. 612–618.

[12] M. Park, S. Chitta, A. Teichman, and M. Yim, "Automatic Configuration Recognition Methods in Modular Robots," *Int. Journal of Robotics Research*, vol. 27, no. 3-4, pp. 403–421, March 2008.

```
module MTRAN:

    point c1: (point _1: coords=(-1, 1, 1)) (point _2: coords=(-2, 0, 1)) (point _3: coords=(-1,-1, 1))\
      (point _4: coords=( 0, 0, 1)) coords=(-1, 0, 1), connector,gender="male",extended=[bool]
    point c2: (point _1: coords=(-2, 1, 0)) (point _2: coords=(-2, 0,-1)) (point _3: coords=(-2,-1, 0))\
      (point _4: coords=(-2, 0, 1)) coords=(-2, 0, 0), connector,gender="male",extended=[bool]
    point c3: (point _1: coords=(-1, 1,-1)) (point _2: coords=( 0, 0,-1)) (point _3: coords=(-1,-1,-1))\
      (point _4: coords=(-2, 0,-1)) coords=(-1, 0,-1), connector,gender="male",extended=[bool]

    point c4: (point _1: coords=( 1, 1,-1)) (point _2: coords=( 2, 0,-1)) (point _3: coords=( 1,-1,-1))\
      (point _4: coords=( 0, 0,-1)) coords=( 1, 0,-1), connector,gender="female",extended=false
    point c5: (point _1: coords=( 2, 1, 0)) (point _2: coords=( 2, 0, 1)) (point _3: coords=( 2,-1, 0))\
      (point _4: coords=( 2, 0,-1)) coords=( 2, 0, 0), connector,gender="female",extended=false
    point c6: (point _1: coords=( 1, 1, 1)) (point _2: coords=( 0, 0, 1)) (point _3: coords=( 1,-1, 1))\
      (point _4: coords=( 2, 0, 1)) coords=( 1, 0, 1), connector,gender="female",extended=false

    link active_block: (point c_1: coords=(-2, 1,-1)) (point c_2: coords=(-2,-1,-1)) (point c_3: coords=(-2,-1, 1))\
      (point c_4: coords=(-2, 1, 1)) grouping=(c1,c2,c3), hull=convex(c1.point,c2.point,c3.point,active_block.point)
    link passive_block: (point c_1: coords=( 2, 1, 1)) (point c_2: coords=( 2,-1, 1)) (point c_3: coords=( 2,-1,-1))\
      (point c_4: coords=( 2, 1,-1)) grouping=(c4,c5,c6), hull=convex(c4.point,c5.point,c6.point,passive_block.point)
    link connecting_bar: (point ab_c: coords=(-1, 0, 0)) (point pb_c: coords=( 1, 0, 0)) (point c_3: coords=(0,0.2,0))\
      (point c_4: coords=(0,-0.2,0)) grouping=(ab_c,pb_c), hull=convex(connecting_bar.point)

    axis j1_ab_rot_axis:  origin=(-1, 0, 0), direction=( 0, 0, 1)
    axis j1_cbar_axis:    origin=(-1, 0, 0), direction=( 0, 0,-1)
    axis j2_pb_rot_axis:  origin=( 1, 0, 0), direction=( 0, 0, 1)
    axis j2_cbar_axis:    origin=( 1, 0, 0), direction=( 0, 0,-1)
    joint joint_1: type=revolute,value=[-pi/2,pi/2],pair=((active_block, j1_ab_rot_axis),(connecting_bar,j1_cbar_axis))
    joint joint_2: type=revolute,value=[-pi/2,pi/2],pair=((passive_block,j2_pb_rot_axis),(connecting_bar,j2_cbar_axis))

connection MTRAN_to_MTRAN: members=( MTRAN a, MTRAN b ), conditions=( (a.point p_a coincident b.point p_b)\
    and (p_a._1 coincident p_b.point) and (p_a._2 coincident p_b.point) and (p_a._3 coincident p_b.point)\
    and (p_a._4 coincident p_b.point) and ( (p_a.gender=="male" and p_a.extended==true and p_b.gender=="female")\
      or (p_b.gender=="male" and p_b.extended==true and p_a.gender=="female") ) )
```

Fig. 8. M-TRAN M3L model (used to customize tools for the M-TRAN robot)

```
robot quadruped: modules=(MTRAN m1,MTRAN m2,MTRAN m3,MTRAN m4,MTRAN m5,MTRAN m6,MTRAN m7,MTRAN m8,MTRAN m9),
 connections=((m1.c2,m2.c5),(m2.c2,m3.c5),(m3.c2,m4.c5),(m2.c3,m5.c4),(m3.c6,m5.c3,3),(m6.c2,m7.c5),
              (m7.c2,m8.c5),(m8.c2,m9.c5),(m7.c1,m5.c6),(m8.c4,m5.c1,35)),
 frame=(m5.passive_block.(axis: origin=(1,0,0), direction=(1,0,0)), m5.passive_block.(axis: origin=(1,0,0),
        direction=(0,1,0)), m5.passive_block.(axis: origin=(1,0,0), direction=(0,0,1)) )

robot equivalent_quadruped: modules=(MTRAN m1,MTRAN m2,MTRAN m3,MTRAN m4,MTRAN m5,MTRAN m6,MTRAN m7,MTRAN m8,MTRAN m9),
 connections=((m1.c2,m2.c5),(m2.c2,m3.c5),(m3.c2,m4.c5,19),(m2.c1,m5.c6),(m3.c4,m5.c1,35),(m6.c2,m7.c5,19),
              (m7.c2,m8.c5),(m8.c2,m9.c5),(m7.c3,m5.c4),(m8.c6,m5.c3,3)), ...
```

Fig. 9. M3L models of equivalent M-TRAN quadrupeds (used for simulated experiments in Sec. V)

.

```
  behavior gait() {
    self.sleepds(20);
    /* start projecting the body forward */
    FrontLeftLowerHalf.from_straight_to_proj_forward();
    FrontRightLowerHalf.from_straight_to_proj_forward();
    BackLeftLowerHalf.from_straight_to_proj_forward();
    BackRightLowerHalf.from_straight_to_proj_forward();
    self.sleepcs(25);
    /* extend the front right leg */
    FrontRightUpperHalf.move_thigh();
    FrontRightLowerHalf.move_foot();
    self.sleepds(10);
    /* retract the back left leg */
    BackLeftUpperHalf.move_thigh();
    BackLeftLowerHalf.move_foot();
    self.sleepds(15);
    /* retract the back right and front left legs */
    BackRightUpperHalf.move_thigh();
    BackRightLowerHalf.move_foot();
    self.sleepcs(25);
    FrontLeftUpperHalf.move_thigh();
    FrontLeftLowerHalf.move_foot();
    self.sleepds(20);
    /* stand straight again */
    FrontLeftLowerHalf.from_proj_backward_to_straight();
    FrontRightLowerHalf.from_proj_backward_to_straight();
    BackLeftLowerHalf.from_proj_backward_to_straight();
    BackRightLowerHalf.from_proj_backward_to_straight();
  }
```

Fig. 10. DynaRole gait behavior for the quadruped of Fig. 9 (controls any physical or simulated robot in a similar kinematic configuration)

[13] E. H. Østergaard, K. Kassow, R. Beck, and H. H. Lund, "Design of the ATRON lattice-based self-reconfigurable robot," *Autonomous Robots*, vol. 21, no. 2, pp. 165–183, September 2006.

[14] D. Brandt, D. J. Christensen, and H. H. Lund, "ATRON Robots: Versatility from Self-Reconfigurable Modules," in *Pro. 2007 IEEE Int. Conf. on Mechatronics and Automation (ICMA'07)*, Harbin, China, August 5-8 2007, pp. 26–32.

[15] M. Bordignon, "Elements of a Software Development Ecosystem for Modular Robotics," Ph.D. dissertation, University of Southern Denmark, Odense, Denmark, 2010, available at http://www.mmmi.sdu.dk/~mirko/thesis.pdf.

[16] U. P. Schultz, "Distributed Control Diffusion: Towards a Flexible Programming Paradigm for Modular Robots," in *Proc. 1st Int. Conf. on Robot Communication and Coordination (RoboComm'07)*, Athens, Greece, October 15-17 2007.

[17] K. Stoy, W.-M. Shen, and P. Will, "Using Role Based Control to Produce Locomotion in Chain-Type Self-Reconfigurable Robots," *IEEE/ASME Transactions on Mechatronics*, vol. 7, no. 4, pp. 410–417, December 2002.

[18] U. P. Schultz, M. Bordignon, D. J. Christensen, and K. Stoy, "Spatial Computing with Labels," in *Proc. SASO'08 Spatial Computing Workshop (SCW'08)*, Venice, Italy, October 20 2008.