A Matlab-Internal DSL for Modelling Hybrid Rigid–Continuum Robots with *TMTDyn*

S.M.Hadi Sadati
School of Biomedical Engineering
& Imaging Sciences
Faculty of Life Sciences & Medicine
King's College London
Email: smh_sadati@kcl.ac.uk

Steffen Zschaler
Department of Informatics
Faculty of Natural
& Mathematical Sciences
King's College London
Email: szschaler@acm.org

Christos Bergeles
School of Biomedical Engineering
& Imaging Sciences
Faculty of Life Sciences & Medicine
King's College London
Email: christos.bergeles@kcl.ac.uk

***Abstract*—Hybrid rigid–continuum robot design addresses a range of challenges associated with using soft robots in application areas such as robotic surgery. Design of such robots poses challenges beyond standard rigid-body robots. A fast, reliable, accurate yet simple dynamic model is important to support the design, analysis, and control of hybrid rigid–continuum robots. In our previous work, we developed a modeling package for hybrid rigid–continuum systems, named *TMTDyn*. In this paper, we focus on how we developed an internal domain-specific language (DSL) using Matlab's OO capabilities and the concept of fluent interfaces to improve validation, understandability, and maintainability of the models constructed using *TMTDyn*. We present the language implementation, and discuss some of the benefits and challenges of building a Matlab-internal DSL.**

I. Introduction

Mimicking highly dexterous and deformable biological bodies has been a trending topic of multi-disciplinary research, called soft robotics, using intrinsically soft materials in the form of continuum robotic platforms [1]. Performing delicate tasks [2], high manoeuvrability in unstructured and confined environments [3], [4], [5], dexterous grasping [6], mimicking biological tissue and organs [7], bio-inspired dynamic locomotion [8] such as crawling [9], terrestrial [10] or submerged locomotion [11] are among the promises made by the research in the field. Soft robots are appealing to investigate new design and theoretical concepts such as variable stiffness structures [12], morphological computation [13] and embodied intelligence [14], to simplify the control and sensing tasks through robot embodiment [15], [16].

However, compliance has disadvantages such as uncertain deformations, limited control feedback, reduced control bandwidth, stability issues, underdamped modes, and lack of precision in tasks involving working against external loads [17], [18]. These result in modeling and control challenges for such designs. There is an urgent need for unified frameworks to transfer our well-established knowledge of dynamic system analysis, path planning and control design for rigid-body robots to soft robotic research [19], [20], [21], [22] and to model hybrid rigid–soft-body systems [23], [24]. Such frameworks should be as simple as possible and easy to use to be widely accepted by the ever-growing soft-robotics research community that gathers researchers from different disciplines and backgrounds. They should provide fast computational performance to be suitable for control and design problems of soft systems with large state spaces. To be useful to the community, such frameworks need to be integrable with standard software platforms (*e.g.* C/C++, Matlab, ROS).

In [25], we introduce two new modelling approaches for continuum rods and actuators, a general reduced-order model (ROM), and a discretized model with absolute states and Euler-Bernoulli beam segments (EBA). These models enable us to perform more accurate simulation of continuum rod manipulators as well as extending the solution to modelling 2D and 3D continuum geometries, which is missing in similar recent research [21]. In [26], these models are further explained and implemented in a Matlab software package—*TMTDyn*—providing a new modelling and simulation tool for hybrid rigid–continuum body systems[1].

Our main goal is to make the tasks of deriving the Equation of Motion (EOM) of hybrid rigid–continuum-body robots, performing dynamic-system analysis, state observation, and control-system design more accessible to the interdisciplinary soft-robotics research community and people with limited expertise in dynamic-system modelling. To this end, in this paper we describe a Matlab-internal DSL (Domain Specific Language) serving as a "front-end" for the *TMTDyn* package. This DSL provides the following benefits:

1) *Accessibility:* the DSL offers an intuitive structure for describing a robotic system that is automatically broken down into the parts required by the *TMTDyn* package.
2) *Early validation:* the DSL can offer validation checks at the time of description rather than the time of evaluation, allowing error messages to be more focused and simplifying debugging.
3) *Maintainability:* changes to the structure to be modelled can be made easily and in a structured fashion; where the plain *TMTDyn* package may require careful adjustment of multiple parts of the code, all related changes are closely linked in the DSL.

While DSLs for kinematics and dynamics in robotics have been developed in the past [27], to the best of our knowledge,

[1]https://github.com/hadisdt/TMTDyn

this is the first Matlab-internal DSL specifically targeting hybrid rigid–soft-body systems.

In the remainder of this paper, we first show a motivating example to develop such DSL for robot analysis in Sec. II. We then give some background on concepts relevant to our work in Sec. III before introducing our DSL in Sec. IV and discussing our experience in developing this DSL in Sec. V.

II. Motivating Example

A dynamic system with inertial, compliant and constraining elements can be expressed as a set of lumped (point) masses, usually assumed at the system elements' center of mass (COM) locations, with moments of inertia which are connected with springs / dampers and joints to the adjacent lumped masses. For a continuum system, where usually a system of differential equations describes the system mechanics, a differential format of the lumped-system approach can be employed. To this end, first, the free body diagram of the load balance in a single differential element is drawn, then the lumped-system equivalence of the system is assumed where the parameters are differential terms.

The following principles guided the design of *TMTDyn*:

- The dynamic motion of a multi-link system is derived where external/input loads, geometrical constraints, rope elements, and soft impacts can be modelled.
- Each element in the system can be assumed as a combination of separable inertial, linear elastic, viscous damping with power law, and external load elements, each with 3D elements.
- The system may have finite or infinite number of elements but must have a finite number of states, forming an Ordinary Differential Equation (ODE) to be integrated numerically over time.
- External / input loads, elastic, damping, geometrical constraint, soft contact, and directional elements (such as string and membrane) are considered as joints between two points on the system but with specific properties to each element type[2].
- 1D continuum elements can be modelled as a finite number of interconnected Euler-Bernoulli elastic beam elements (discretization), or as continuous beam elements with predefined polynomial deformation shape functions (ROM).
- 2D & 3D continuum elements can be modelled as wire meshes in which edges are 1D Euler-Bernoulli beams and connections are point masses.
- Hyperelasticity is not captured directly but can be added by updating an element stiffness matrix in an intermediate step during the numerical simulation.

System kinematics describes the geometric relations between the system elements in terms of rotation and translation. *TMTDyn* derives equations describing the position vector and

TABLE I: The modeling parameters for the experiments with a fabric sleeve around a single link pendulum (M: Measured, C: Calibrated).

Sym.	Value	Metric	Sym.	Value	Metric
m_1[g]	40	M	m_1[g]	36	M
l_{m_1}[mm]	270	M	l_{com_1}[mm]	135	M
$l_{m_{2x}}$[mm]	350	M	$l_{m_{2y}}$[mm]	0.8	M
$l_{m_{2z}}$[mm]	99	M	l_{c_x}[mm]	38	M
l_{c_z}[mm]	30	M	θ_h[deg]	85	M
E[KPa]	5	C	ν	1	C
μ_h[Ns/m]	1e2	C	μ_ϵ[Ns/m]	1e2	C
μ_α[Nms/rad]	1e2	C			

orientation of the local frame attached to each point of the system over time.

As an example, consider a simple model of the dynamic deformation of a fabric sleeve (as a 2D continuum medium) worn on an elbow-like rigid-link pendulum (E2), modelled as a complex hybrid system in our previous work. The results from such a model can be useful for research on wearable sensors. Capturing the dynamics of soft fabrics can provide many benefits to textile-embedded human motion analysis systems, such as those used for computer animation or rehabilitation feedback [28], [29]. Table I presents the fabric and setup dimensions and the simulation parameters. We will use this example throughout this paper. Below, we first show its specification in the current *TMTDyn* package and highlight challenges of the current interface.

A. Pendulum with Fabric Sleeve Setup

A fabric sleeve, made of Jersey fabric, was cut and clamped on a rigid-link pendulum, cut to shape out of ABS clear plastic (Fig. 1). This shape is modelled on a standard sized human arm, and used to simulate the effect of clothing movement given wearer motion. The pendulum was fixed with a 1-DOF (degree of freedom) joint at the top and passively swings. The model was intended to capture the fabric dynamics due to the pendulum's free motion. Three magnetic trackers were used to measure the link COM motion, and deformation of two points on the fabric (s_1, s_2).

1) Modelling Assumptions & Program Input: The fabric can be modelled as a membrane, which is a 2D tension-only continuum geometry that does not withstand bending or compression. This can be done by assuming the fabric as a net of equally distributed masses with connecting linear springs; a lumped-mass approximation of the fabric mesh. We have used a similar method to model a spider web with *TMTDyn* recently [30].

To model the system, we focus on the fabric model and consider the link motion as a passive swinging pendulum from an initial state based on the experimental recording. The fabric deforms when clamped on the link. The overall geometry of the clamped fabric is modelled with FreeCAD software as a wireframe sketch with a 3×5 grid of $n_d = 15$ nodes and 22 edges as in Fig. 1.c. The CAD model is stored in Initial Graphics Exchange Specification (IGES) format to be

[2]This will be discussed more later; our DSL provides dedicated keywords to reduce cognitive load and improve validation.

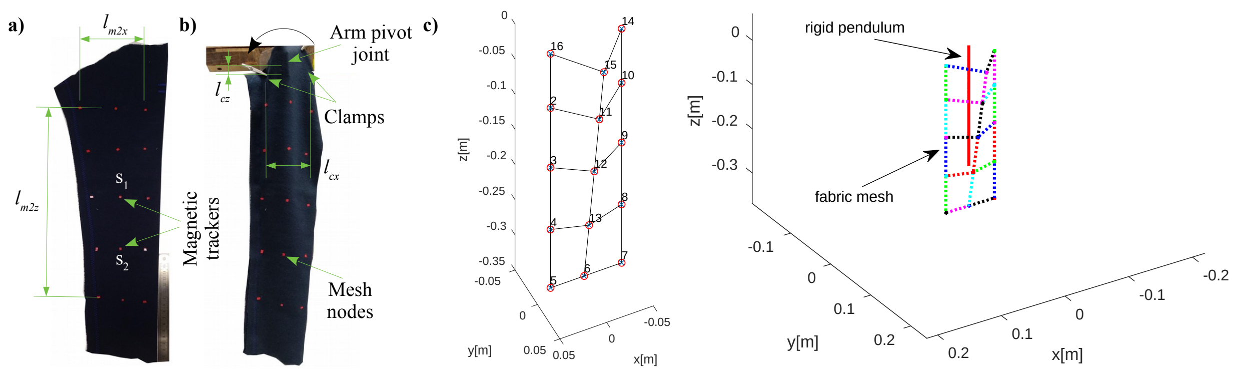

Fig. 1: a) A fabric is cut to form a fabric sleeve around a single link pendulum. b) The link forms a passive pendulum with clamped fabric sleeve. The link is fixed with a 1 DOF joint at the top and two magnetic trackers at nodes s_1 & s_2. Red dots are equivalent to the CAD-file nodes. c) CAD-file wireframe of the sleeve in the clamped configuration (as shown by $mesh\_import.m$ module) and the final animation of the simulation results.

imported into the *TMTDyn* model later. The fabric is clamped at nodes 14 & 16 to the link at position $[\pm l_{c_x}, 0, l_{c_z}]$ (see Fig. 1). Two sets of six constraints are defined to fully fix each of these two nodes to the link.

The fabric mesh is modelled with lumped masses at the CAD-file wireframe nodes that are interconnected with Euler–Bernoulli (EB) beams. The system states are described with absolute states of the masses where the frames defining the DOF of each mass are defined w.r.t. the system reference frame. The nodes are rigid lumped masses with an equally distributed mass of m_2/n_d. Considering the fabric as a thin plate, the following relation is used to derive the nodes' second moment of inertia as in Eq. 1. where $\odot$ denotes inner product. Links are considered as EB ribbons (beams) connecting the pairs of point masses on the nodes, where the frames defining the ends of a link are defined w.r.t. the local frame of the connecting masses. The beams have linear elasticity $K_\epsilon = \text{diag}(a_c[G, G, E])$ and $K_\alpha = \text{diag}([E, E, G]).\text{diag}(J)$. Here, $J = [l^3_{m2_y} l_{m2_b}, l_{m2_y} l^3_{m2_b}, l^3_{m2_y} l_{m2_b} + l_{m2_y} l^3_{m2_b}]$ is a 1×3 vector consisting of the EB ribbons' cross-section second moments of areas, where $l_{m2_b} = (l_{m2_x} + l_{m2_z})/2$ is the mean width of the ribbons in the x, z-axis directions.

To map the nodes' motion to the beams deformation map, the beams are defined in the local body frames (`xaxis` is not defined) with an initial deformation to comply with the initial geometry of the fabric, as imported from the `'cad.iges'` file, by setting $init = nan$[3].

B. TMTDyn *specification*

The inputs for the *TMTDyn* package to model this setup are shown in Listing 1. *TMTDyn* uses a number of Matlab structs to capture input data. Specifically, the par struct is used to capture parameters controlling how *TMTDyn* derives EOMs and what it does with them. These are mostly boolean flags selecting individual features of the library to be used. The world struct allows specifying global parameters, such as the gravity vector. The body and joint vectors contain structs for modelling the robot geometry. Finally, the mesh struct allows one CAD file to be loaded, describing a set of interconnected lumped masses, which in this example is used to model the fabric mesh.

[3]See the project wiki page at https://github.com/hadisdt/TMTDyn/wiki

C. *Drawbacks of current specification interface*

The current interface is not atypical of Matlab libraries. The use of structs provides a flexible and efficient encoding of the information needed to derive EOMs for hybrid rigid–continuum robots, but it also creates a number of challenges:

1) *Accessibility:* Creating and understanding a model in the *TMTDyn* format can be challenging. One has to have a good understanding of how rotation and translation are encoded in *TMTDyn* both for fixing the relationship between different parts of the system and for describing degrees of freedom that enable the robot to move. There is concept overloading, where the same interface concept is used to describe different things (*e.g.,* 'joint' is an overloaded concept that can describe an actual joint as well as a geometric constraint between two masses and different additional information must be provided for each case).
2) *Validation:* The *TMTDyn* package implements a range of checks to validate the model before computation starts. While these checks will flag up errors, actually identifying and correcting the cause of the problem is not easy. This is compounded by the dynamic nature of Matlab, allowing struct members to be declared on the fly, simply by setting their values. As a result, a small misspelling of one of the keywords will lead to an invalid model, which can be very difficult to debug as the problem is only discovered when attempting to analyse the model. Matlab does not provide any built-in support for analysis for this interface.
3) *Maintainability:* There are some complex interactions between different parts of the specification. For example, the description of DOF requires an entry in the joints

$$I_2 = \frac{m_2}{12n_d} \odot \begin{bmatrix} l_{m2_y}^2 + (\frac{l_{m2_z}}{5})^2 & 0 & 0 \\ 0 & (\frac{l_{m2_x}}{3})^2 + (\frac{l_{m2_z}}{5})^2 & 0 \\ 0 & 0 & (\frac{l_{m2_x}}{3})^2 + l_{m2_y}^2 \end{bmatrix}, \tag{1}$$

Listing 1: *TMTDyn* package input for the sleeve fabric models clamped to a rigid-link pendulum, structure-based user interface. Model labels are as in Fig. 1.

```
par.derive = 0 ; % derive TMT EOM
par.derive_mex = 0 ; % use Matlab codegen
par.simdyn = 2 ; % dynamic simulation with C-mex files
par.post_process = 1 ; % post-process using user-specified code
par.anim = 1 ; % animate the results
syms E $\nu$ $\mu_\epsilon$ $\mu_\alpha$ $\theta_{h_0}$ $\mu_h$ ; % symbolic variable definition for derivations
par.sym = [ E, $\nu$, $\mu_\epsilon$, $\mu_\alpha$, $\theta_{h_0}$, $\mu_h$ ];
par.var = [ 5e3, 1, 1e2, 1e2, 1.48, 1e2 ] ; % values for symbolic variables in simulation as in Table
    1.

world.g = [0,0,-$g$] ; % gravity

body(1).m = $m_2$ ; % pendulum rigid link
body(1).l_com = [0, 0, -$l_{m_1}$]; % pendulum COM

joint(1).second = 1 ;
joint(1).tr.rot = [ 2 , inf ] ; % 1 DOF rotation around y-axis
joint(1).dof.init = $\theta_{h_0}$ ; % pendulum initial angle
joint(1).dof.damp.visc = $\mu_h$ ; % pendulum joint viscous damping

% Import mesh geometry:
mesh.file_name = 'cad.iges'; % CAD-file name
mesh.tol = 1e-3; % geometry import tolerance
mesh.tr.trans = [0,0,$l_{c_z}$]; % mesh geometry initial position/orientation
mesh.tr.rot = [2,$\theta_{h_0}$];
mesh.body.m = $m_2/n_d$ ; % equally distributed fabric mass over the nodes
mesh.body.I = $I_2$ ; % Describing the mesh absolute DOF with mesh.joint(1):
mesh.joint(1).tr.trans = [ inf, inf, inf ] ; % masses absolute state as system DOFs
mesh.joint(1).tr.rot_type = 'non_unit_quat' ; % orientation representation type
mesh.joint(1).tr.rot = [ inf, inf, inf, inf ] ; % non-unit quaternion
mesh.joint(1).dof(4).init = 1 ; % quaternion initial value

% Describing the mesh EB beam connections with mesh.joint(2):
mesh.joint(2).spring.coeff = [ diag( $K_\epsilon$ ), diag( $K_\alpha$ ) ] ; % linear elasticity of beams
mesh.joint(2).spring.init = nan ; % beam initial state from system geometry
mesh.joint(2).damp.visc = [ $\mu_\epsilon$, $\mu_\alpha$ ] ; % linear viscous damping
mesh.joint(2).damp.power = $\nu$ ; % damping power law

% Fabric clamps:
joint(2).first = 1 ;
joint(2).second = 16 ; % clamp at node 16 based on mesh file plot
joint(2).tr.trans = [ $l_{c_x}$, 0, -$l_{c_z}$ ] ;
joint(2).fixed = ones(1,3) ; % only the Cartesian location is constraint (free relative rotations)
joint(3).first = 1 ;
joint(3).second = 14 ; % clamp at node 14 based on mesh file plot
joint(3).tr.trans = [ -$l_{c_x}$, 0, -$l_{c_z}$ ] ;
joint(3).fixed = ones(1,3) ;
```

vector to be coordinated with a number of entries in the DOF vector (one for each `inf` transformation in the joint specification). The connection is made based on the index in both vectors. However, because for every joint there may be more than one DOF entry, identifying the correct DOF entry for a given joint entry is a non-trivial task with a high cognitive load. This is particularly problematic when making changes to the model, where a small change to a joint can inadvertently cause the two sets of index to become out of sync, leading to an invalid model that is very difficult to debug.

III. Background

In this section, we provide brief background on concepts relevant to our work. Specifically, we briefly discuss domain-specific languages (DSLs) and the differentiation of internal and external DSLs, the idea of fluent interfaces for implementing internal DSLs, and Matlab's approach to object-orientation as this will be required for the development of a fluent language.

Domain-Specific Languages: are computer languages developed specifically to capture problems in a specific application domain. DSLs allow domain experts to express their goals, problems or requirements in terms they are familiar with, while ensuring that these expressions can be meaningfully interpreted (and often 'executed' in some form) by a computer. This is useful because it encapsulates details required at the level of abstraction at which a computational process actually works and shields domain experts from them, leading to a more effective division of labour in a cross-disciplinary group of experts. For our purposes, we wish to hide the details of how EOMs are computed and provide a language that is close to how someone aiming to solve a particular problem with the help of a novel robot thinks about the robot's structure.

DSLs can be internal or external [31]. The former are developed directly embedded in an existing 'host' language—often a general-purpose language such as Ruby, Java, or C++. The latter are developed as independent languages which are subsequently interpreted or compiled for execution. Robot design uses a rich set of existing languages and tools, most notably Matlab and C++. Adding yet another standalone language to this mix is a hard sell: existing languages and tools come with a rich ecosystem of libraries and support infrastructure, which would need to be adapted and translated at great cost to be interoperable with a newly introduced external DSL. Instead, an internal DSL embedded in Matlab offers the right trade-offs for this problem, despite the well-known limitations of internal DSLs.

Fluent interfaces: are the typical approach to API design used in developing internal DSLs [31]. The key idea is to use a set of interacting classes, whose methods can be chained together into call sequences that read like keywords in a new language. This is achieved by making each method return either the object it was called on or a new object representing a subordinate language scope. Carefully designing the names of methods so that chains of method invocations can be read like sentences increases the usability of the language.

Object-orientation in Matlab: Fluent interfaces rely on using object-oriented concepts for API design to enable method chaining. Matlab is originally not an object-oriented language, but does provide object-oriented features. Some idiosyncrasies of the Matlab approach to object orientation must be taken into account when implementing fluent interfaces: By default, objects are passed by value in Matlab rather than by reference. This makes developing methods that return a sub-scope while updating the current scope difficult. Fortunately, objects of classes that sub-class the `handler` class are always passed by reference. Matlab also doesn't support statements that continue over multiple lines of text. Instead, an ellipsis (...) must be placed at the end of each line. This will add some syntactical clutter to our internal DSL.

IV. A Matlab-internal DSL

Listing 2 shows our motivating example expressed in our new DSL. The DSL is a Matlab-internal language built using fluent interfaces [31]. For each new context (indicated in the listing by levels of indentation), we have implemented a separate builder class providing the keywords available to the user at this level. The entire specification starts by using `tmtdyn()`, which creates an instance of the root `tmtdyn` builder class. From `tmtdyn()`, three keywords are available: `simulation()` to specify simulation parameters, `world()` to specify general world parameters[4], and `robot()` to start the definition of the robot structure. Finally, users use `run()` to run the analysis based on the specification provided.

Rather than providing a detailed account of every line of Listing 2 (many of which are hopefully self-explanatory), we will focus on highlighting some of the key design principles behind our DSL:

1) Using Matlab OO to support fluent interfaces;
2) Improving accessibility through specialised keywords;
3) Improving maintainability by co-locating the definition of transformations and additional information about degrees of freedom; and
4) Improving validation using runtime checks for method availability.

A. Using Matlab OO to support fluent interfaces

For each scope, we have defined a new Matlab class using the Builder design pattern [32]. The `tmtdyn().run()` method then extracts all information accumulated and invokes the appropriate *TMTDyn* functions as configured in the `simulation()` settings.

All builder classes follow the fluent-interface design pattern. Methods return a reference to self (or a fresh sub-builder if a new scope is opened) so that they can be easily chained using dot notation. Method names have been carefully chosen to achieve natural and intuitive readability of chained method calls. A technical challenge is that, by default, all Matlab objects are passed by value. As a result, a method like the one below from class `tmtdyn` will not work as expected:

```
function robot = robot(self, name)
  self.the_robot = robot_builder(self, name);
  robot = self.the_robot;
end
```

In particular, the update to `self` will be lost as soon as the `robot()` method returns. Fortunately, this counter-intuitive behaviour can be easily fixed by ensuring all classes extend the `handle` system class. Instances of `handle` are always passed by reference.

The use of separate classes for different scopes restricts the 'keywords' available at each point in a specification: only the methods defined in the current builder class can be invoked. Calling a method not defined in the current scope will lead to an error when executing the Matlab program. While this is an important feature for validation purposes (see below), it can also cause problems. In particular, in a naïve language implementation, users would have to explicitly close each scope to obtain a reference to the containing scope, for example by chaining an, otherwise meaningless,

[4] Currently, *TMTDyn* only supports defining the gravity vector

Listing 2: Motivating example expressed using our internal DSL

```
tmtdyn()...
  .simulation()...
    .var($E$, 5e5)...
    .var($\nu$, 1)...
    .var($\mu_\epsilon$, 1e2)...
    .var($\mu_\alpha$, 1e2)...
    .var($\theta_{h_0}$, 1.48)... % in radians
    .var($\mu_h$, 1e2)...
    .derive_eom()...
      .use_mex()...
      .optimize_code()...
    .analysis()...
      .dynamic_sim('m_file', 0, 1)... % simulation for t=0:1 s
    .post_process()...
      .animate()...
      .run_user_code()...
  .world()...
    .g([0, 0, $-g$])...
  .robot('fabric_pendulum')...
    .body('arm')...
      .with_mass($m_2$)...
      .with_center_of_mass_at([0, 0, $-l_{m_1}$])...
    .connected()...
      .with_transformation_from()...
        .rot_y()...
          .initial_value($\theta_{h_0}$)...
          .parallel_damper()...
            .viscosity($\mu_h$)...
  .mesh('fabric')...
    .from_file('exp/exp2.iges', 1e-3)...
    .with_transformation()...
      .rot_y($\theta_{h_0}$)...
      .trans_z($-l_{c_z}$)...
    .with_node('fabric')...
      .with_mass($m_2/n_d$)...
      .with_inertia($I_2$)...
      .connected()...
        .with_transformation_from()...
          .translation([inf, inf, inf])...
          .rot_non_unit_quat([inf, inf, inf, inf])...
            .dof(1)...
              .initial_value(1)...
    .with_edge('fabric_links')...
      .beam_stiffness()...
        .coefficient([diag($K_\epsilon$)', diag($K_\alpha$)'])...
        .initial_state_from_configuration()...
      .beam_damping()...
        .viscosity([$\mu_\epsilon$*ones(1,3), $\mu_\alpha$*ones(1,3)])...
        .power($\nu$)...
  .constraint('clip_constraint_1')...
    .from_body(1)...
    .with_transformation_from()...
      .translation([$l_{c_x}$, 0, $-l_{c_z}$])...
    .to_body(16)...
    .fixed_directions([1, 1, 1])...
  .constraint('clip_constraint_2')...
    .from_body(1)...
    .with_transformation_from()...
      .translation([$-l_{c_x}$, 0, $-l_{c_z}$])...
    .to_body(14)...
    .fixed_directions([1, 1, 1])...
  .run();
```

`.end()` call. Each scope object contains a private property called `the_source`, which references the containing scope element, so `end()` would be implemented like this:

```
function source = end(self)
  source = self.the_source;
end
```

Calling `end()` would then return the containing scope, so that subsequent chained method calls would be able to use the methods defined in that scope again. However, this would create a lot of syntactic clutter, which is generally undesirable in language design. Fortunately, Matlab provides basic meta-programming capabilities, which enable us to intercept the routing of method calls. We use this to implicitly close scopes when a method from a containing scope is invoked. To do so, each builder class overrides the `subsref` method, responsible for method lookup:

```
function varargout = subsref (self, S)
  try
    [varargout{1:nargout}] = ...
        builtin('subsref', self, S);
  catch
    self.onLeaveContext();
    [varargout{1:nargout}] = ...
        builtin('subsref', self.the_source, S);
  end
end
```

This first attempts to look up any requested method in the current object. If the method is not found there, it is looked up in the containing scope, which can be found via property `the_source`. Here, `onLeaveContext()` is a method in each builder class to do context-specific cleanup.

Automatically closing scopes like this makes the DSL more concise, but can also lead to problems like the well-known "dangling else problem" [33]: If the same keyword is available in two nested scopes, which one does the user mean? In our DSL, we avoid this problem by using distinct keywords where similar concept are available in different nested scopes. For example, both joints and general DOFs can have springs and dampers attached. A `joint_builder` describes a connection between two frames and can have stiffness and damping element to restrict the relative motion of these two frames in the form of a parallel spring–damper. Any `dof_builder`, defining a system state, can have a parallel spring–damper system. To solve the problem of scopes we chose different names for the methods that define these spring–damper systems in each scope; both are translated to the `.spring` and `.damper` sub-field of the struct-based interface. For example, a spring element is named `beam_stiffness` in the `joint_builder` class and `parallel_spring` in the `dof_builder` class. Apart from resolving issues in scoping, these names also better describe the purpose and functionality of the elements in their respective context.

B. Improving accessibility through specialised keywords

In the structs-based approach, all transformations need to be specified using axis index and value, vectors, quaternions etc. This creates additional cognitive load when reading a specification as the reader has to constantly back translate to a more intuitive representation. In our DSL, we still offer these ways of specifying transformations, but also provide specialised operators for typical transformations, that allow for better readability. For example, to specify a rotational degree of freedom around the y-axis, one simply says `.rot_y()`. Similarly, to specify a fixed rotation around the same axis, one simply provides a value parameter to the method call: `.rot_y(.5)`. Rotations and translations can be mixed arbitrarily. The DSL will automatically generate an appropriate number of `tr` records in the structs that are passed to the underlying *TMTDyn* library at the end. This is done by combining sequences of rotations (translations) and creating new records whenever a translation is followed by a rotation.

Another such example is the specification of the initial state of the mesh. Remember from Sect. II that we used an initial value of `nan` to specify that the mesh should initially be deformed to be connected to the arm according to the overall configuration. This encoding clearly loses the intuition. Instead, on Line 46 of Listing 2, we use a bespoke keyword `initial_state_from_configuration` to express the same thing in a more intuitive manner.

C. Improving maintainability

In the structs-based interface, the definition of what degrees of freedom exist (by using `inf` values for elements in transformation vectors) and the specification of their properties (damping, spring properties, *etc*) are contained in two separate arrays, requiring users to track complicated links between two sets of indexes. This makes changing the code very error prone: adding or removing a degree of freedom in a transformation somewhere in the structural specification means tracking down the corresponding index in the `dof` vector and updating it and all subsequent indexes accordingly. Mistakes made in this process are very difficult to spot and correct; often the only sensible way of fixing a problem is to reconstruct the `dof` vector from scratch.

In our DSL, we choose a different approach: DOF details are given directly in a sub-scope of a transformation-specification that declares a new degree of freedom. For example, Lines 25–28 in Listing 2 declare that the robot arm can be freely rotated around the y-axis (DOF declaration, previously defined in `body.tr.rot`) and immediately specify the behaviour of the arm when rotated in this way (previously defined in `dof` vector). As a result, maintainability is improved, because users no longer need to maintain consistency between two separate vector-index ranges. Instead, the correct vectors are generated from the DSL specification.

D. Improving validation

Our fluent DSL improves validation in two ways:

1) *Better use of Matlab checking mechanisms:* Matlab is a very dynamic language. Variables do not need to be declared, but can be used straightaway. While this can have many benefits, it also means that a small typo in a name silently creates a new variable (or struct member) rather than setting a required property. As a result, *TMTDyn* will produce incorrect results, but this can be very hard to spot and debug. In contrast, with a fluent DSL only the methods explicitly defined in a scope are available to be used. Any typos will be picked up by Matlab when it tries to invoke the method and an error will be thrown at this point, helping identify the cause of the problem instantly. This can also be used to ensure consistency constraints are satisfied. For example, joints in a *TMTDyn* model can either connect two bodies or can indicate where a body is connected to the base. In the former case, two bodies must be specified, whereas in the latter case only one body is required. We provide the `joint()` keyword to define a standard joint while using the `connected_from()` keyword for defining joints that connect to the base. Both return a type of joint builder, but only the one returned by `joint()` has a method for defining the `from_body()`. For `connected_from()`, the source body is automatically set from the specification context.
2) *Localised consistency checks:* In the existing *TMTDyn* library all consistency checks are only undertaken once the complete struct has been defined. This makes it difficult to provide error messages clearly locating the source of a problem. Because every declaration is done through a method call, we can distribute consistency checks throughout the model creation. For example, we are easily able to check the correct format of any vector provided at the point that it is defined.

V. Discussion

The DSL we have presented in this paper is the current endpoint of a language-design journey. We originally started our collaboration on a Matlab package for deriving EOMs for primarily rigid-body robots. This package had already some of the features of the current *TMTDyn* package, but had a much more basic interface, where every aspect of the system to be modelled was captured in a different vector, with no meaningful naming conventions, validation, etc. We initially experimented with an external DSL—written in Xtext [34]—that allowed fairly comfortable high-level specification of robot structures with clean syntax and good tool support, including an amount of in-editor error checking and validation[5]. While using an external DSL enabled very clean syntax and some fairly powerful features, including the easy specification of model variants, this external DSL struggled to be accepted by users and we eventually gave it up. The main challenge was that the external DSL required users to become familiar

[5]The last version of this DSL we explored can still be found at https://github.com/szschaler/RigidBodies/

with a separate tool set, which was perceived as a hurdle too high. However, the design of that original DSL informed the redesign of the current, struct-based interface of *TMTDyn*. While this improves on the original interface, and in parts already reads like a DSL, it leaves substantial challenges to accessibility, maintainability, and validation as we have discussed in this paper. This is also supported by informal feedback received from some of the users of that interface, who highlighted the difficulty of specifying basic relative rotations (for which our DSL now has introduced dedicated keywords) or complex geometry (which can now be imported directly using IGES format). There was also feedback indicating that the struct-based interface requires substantial initial training; we hope that the DSL-based approach has improved this situation. With the current, Matlab-internal DSL, we feel we are getting closer to a design sweet spot that balances reuse of existing tooling infrastructure against the strengths of DSLs in improving accessibility, maintainability, and validation.

An internal DSL in Matlab has many benefits, not least the ability to reuse Matlab's rich and flexible mathematical expression language. However, Matlab also makes it difficult in some regards to create a seamless language experience from a fluent interface. Most annoyingly, Matlab requires continuation markers (...) to indicate a line of code that continues on the next line of input. This can add substantial syntactic clutter. Fluent interfaces are often praised for their discoverability [31] as they integrate nicely with code-completion functionality offered by modern development environments. Unfortunately, discoverability is limited in a Matlab-based fluent interface because the Matlab language is highly dynamic, making it near impossible to predict statically what methods might be invoked in a particular place. However, using fluent interfaces at least provides a fail-fast capability that reports any mistyped keywords as soon as the code is executed.

VI. Conclusions

We have presented a Matlab-based DSL enabling the specification of hybrid rigid–continuum robots so that EOMs for these robots can be derived automatically by the *TMTDyn* package. The new DSL improves accessibility, maintainability, and validation of robot models using our approach. The examples from this paper are available online[6].

This is not the first DSL developed for the robotics domain. In [27], Nordmann *et al.* provide a detailed survey of the growing landscape of literature in this field. They also define a set of dimensions enabling the classification of existing and new DSLs. We classify our work in these terms as follows:

- *Functional dimension.* We are focusing on kinematics and dynamics. While there are a good number of DSLs in this space already, ours is the first supporting hybrid systems and soft robots.
- *Process stage.* We cover a number of process stages as listed by [27]:
  1) *Capability building* has been implemented: full equations of motion are derived for any system specified in our DSL;
  2) *Platform building* is partially addressed through the support for importing mesh definitions and IGES files enabling modelling of platforms such as continuum arms or deformable 2D structures.

[6]https://github.com/hadisdt/TMTDyn_hll

In future work, we will explore further useful features to be added to the DSL. For example, more complex configuration patterns could be incorporated as pre-defined keywords for typical recurring model elements (spherical joints, for example) that would further improve the efficiency of specifying robot models in our Matlab package. We will apply this new DSL to model further examples, which will help further refine and improve the modelling capabilities offered. We also plan to undertake an empirical study to evaluate and further improve the usability of our DSL in terms of productivity and reliability.

Acknowledgements

S.M.H. Sadati and C. Bergeles are funded by ERC Starting Grant No. 714562.

References

[1] D. Rus and M. T. Tolley, "Design, fabrication and control of soft robots," *Nature*, vol. 521, no. 7553, pp. 467–475, 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14543

[2] M. Cianchetti, T. Ranzani, G. Gerboni, T. Nanayakkara, K. Althoefer, P. Dasgupta, and A. Menciassi, "Soft Robotics Technologies to Address Shortcomings in Today's Minimally Invasive Surgery: The STIFF-FLOP Approach," *Soft Robotics*, vol. 1, no. 2, pp. 122–131, 2014. [Online]. Available: http://online.liebertpub.com/doi/abs/10.1089/soro.2014.0001

[3] J. Burgner-Kahrs, D. C. Rucker, and H. Choset, "Continuum Robots for Medical Applications: A Survey," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1261–1280, Dec. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7314984/

[4] M. Cianchetti and A. Menciassi, "Soft Robots in Surgery," in *Soft Robotics: Trends, Applications and Challenges*, 1st ed., ser. Biosystems & Biorobotics. Springer International Publishing, 2017, vol. 9, pp. 75–85. [Online]. Available: http://link.springer.com/10.1007/978-3-319-46460-2_10

[5] I. D. Walker, H. Choset, and G. S. Chirikjian, "Snake-Like and Continuum Robots," in *Springer Handbook of Robotics*. Cham: Springer International Publishing, 2016, pp. 481–498. [Online]. Available: http://link.springer.com/10.1007/978-3-319-32552-1_20

[6] R. K. Katzschmann, A. D. Marchese, and D. Rus, "Autonomous Object Manipulation Using a Soft Planar Grasping Manipulator," *Soft Robotics*, vol. 2, no. 4, pp. 155–164, Dec. 2015. [Online]. Available: http://online.liebertpub.com/doi/abs/10.1089/soro.2015.0013

[7] L. He, N. Herzig, S. d. Lusignan, and T. Nanayakkara, "Granular Jamming Based Controllable Organ Design for Abdominal Palpation," in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Jul. 2018, pp. 2154–2157.

[8] M. Wehner, R. L. Truby, D. J. Fitzgerald, B. Mosadegh, G. M. Whitesides, J. A. Lewis, and R. J. Wood, "An integrated design and fabrication strategy for entirely soft, autonomous robots," *Nature*, vol. 536, no. 7617, pp. 451–455, Aug. 2016. [Online]. Available: http://www.nature.com/nature/journal/v536/n7617/full/nature19100.html

[9] S. I. Rich, R. J. Wood, and C. Majidi, "Untethered soft robotics," *Nature Electronics*, vol. 1, no. 2, p. 102, 2018.

[10] I. S. Godage, T. Nanayakkara, and D. G. Caldwell, "Locomotion with continuum limbs," *IEEE International Conference on Intelligent Robots and Systems*, pp. 293–298, 2012.

[11] M. Cianchetti, M. Calisti, L. Margheri, M. Kuba, and C. Laschi, "Bioinspired locomotion and grasping in water: the soft eight-arm OCTOPUS robot," *Bioinspiration & Biomimetics*, vol. 10, no. 3, p. 035003, May 2015. [Online]. Available: http://stacks.iop.org/1748-3190/10/i=3/a=035003?key=crossref.7e7a029ec68cfb24c606d395db7d7611

[12] M. A. McEvoy and N. Correll, "Shape-Changing Materials Using Variable Stiffness and Distributed Control," *Soft Robotics*, Oct. 2018. [Online]. Available: https://www.liebertpub.com/doi/abs/10.1089/soro.2017.0147

[13] K. Nakajima, H. Hauser, T. Li, and R. Pfeifer, "Exploiting the Dynamics of Soft Materials for Machine Learning," *Soft Robotics*, Apr. 2018. [Online]. Available: https://www.liebertpub.com/doi/full/10.1089/soro.2017.0075

[14] ——, "Information processing via physical soft body," *Scientific Reports*, vol. 5, p. 10487, May 2015. [Online]. Available: https://www.nature.com/articles/srep10487

[15] R. M. Füchslin, A. Dzyakanchuk, D. Flumini, H. Hauser, K. J. Hunt, R. H. Luchsinger, B. Reller, S. Scheidegger, and R. Walker, "Morphological Computation and Morphological Control: Steps Toward a Formal Theory and Applications," *Artificial Life*, vol. 19, no. 1, pp. 9–34, Nov. 2012. [Online]. Available: https://doi.org/10.1162/ARTL_a_00079

[16] T. G. Thuruthel, Y. Ansari, E. Falotico, and C. Laschi, "Control Strategies for Soft Robotic Manipulators: A Survey," *Soft Robotics*, vol. 5, no. 2, pp. 149–163, Apr. 2018. [Online]. Available: https://www.liebertpub.com/doi/10.1089/soro.2017.0007

[17] L. Blanc, A. Delchambre, and P. Lambert, "Flexible Medical Devices: Review of Controllable Stiffness Solutions," *Actuators*, vol. 6, no. 3, p. 23, Jul. 2017. [Online]. Available: http://www.mdpi.com/2076-0825/6/3/23

[18] M. Cianchetti, T. Ranzani, G. Gerboni, I. De Falco, C. Laschi, and A. Menciassi, "STIFF-FLOP surgical manipulator: Mechanical design and experimental characterization of the single module," in *IEEE International Conference on Intelligent Robots and Systems (IROS)*. Tokyo, Japan: IEEE, 2013, pp. 3576–3581.

[19] A. D. Kapadia, I. D. Walker, D. M. Dawson, and E. Tatlicioglu, "A Model-based Sliding Mode Controller for Extensible Continuum Robots," in *Proceedings of the 9th WSEAS International Conference on Signal Processing, Robotics and Automation*, ser. ISPRA'10. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2010, pp. 113–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=1807817.1807840

[20] F. Renda and L. Seneviratne, "A Geometric and Unified Approach for Modeling Soft-Rigid Multi-Body Systems with Lumped and Distributed Degrees of Freedom," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 1567–1574.

[21] F. Renda, F. Boyer, J. Dias, and L. Seneviratne, "Discrete Cosserat Approach for Multisection Soft Manipulator Dynamics," *IEEE Transactions on Robotics*, pp. 1–16, 2018.

[22] C. Della Santina, D. Lakatos, A. Bicchi, and A. Albu-Schäffer, "Using Nonlinear Normal Modes for Execution of Efficient Cyclic Motions in Soft Robots," *arXiv:1806.08389 [cs]*, Jun. 2018, arXiv: 1806.08389. [Online]. Available: http://arxiv.org/abs/1806.08389

[23] S. Sadati, L. Sullivan, I. Walker, K. Althoefer, and T. Nanayakkara, "Three-Dimensional-Printable Thermoactive Helical Interface With Decentralized Morphological Stiffness Control for Continuum Manipulators," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2283–2290, Jul. 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8288847/

[24] L. Paternò, G. Tortora, and A. Menciassi, "Hybrid Soft–Rigid Actuators for Minimally Invasive Surgery," *Soft Robotics*, Oct. 2018. [Online]. Available: https://www.liebertpub.com/doi/full/10.1089/soro.2017.0140

[25] S. Sadati, A. Shiva, L. Renson, C. Rucker, K. Althoefer, T. Nanayakkara, C. Bergeles, H. Hauser, and I. Walker, "Reduced Order vs. Discretized Lumped System Models with Absolute and Relative States for Continuum Manipulators," in *Robotics: Science and Systems*, Freiburg, Germany, 2019, p. 10.

[26] S. Sadati, S. E. Naghibi, A. Shiva, M. Brendan, L. Renson, M. Howard, C. Rucker, K. Althoefer, T. Nanayakkara, S. Zschaler, C. Bergeles, H. Hauser, and I. D. Walker, "*TMTDyn*: A Matlab package for modeling and control of hybrid rigid–continuum robots based on discretized lumped system and reduced order models," (under review). [Online]. Available: https://bit.ly/2XvcgiI

[27] A. Nordmann, N. Hochgeschwende, D. Wigand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 75–99, Jul. 2016.

[28] B. Michael and M. Howard, "Activity recognition with wearable sensors on loose clothing," *PLOS ONE*, vol. 12, no. 10, p. e0184642, Oct. 2017. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0184642

[29] ——, "Gait Reconstruction From Motion Artefact Corrupted Fabric-Embedded Sensors," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1918–1924, Jul. 2018.

[30] S. M. H. Sadati and T. Williams, "Toward Computing with Spider Webs: Computational Setup Realization," in *Biomimetic and Biohybrid Systems*, ser. Lecture Notes in Computer Science. Springer, Cham, Jul. 2018, pp. 391–402. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-95972-6_43

[31] M. Fowler and R. Parsons, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Professional Computing Series. Addison Wesley Professional, 1995.

[33] A. F. Kaupe, "A note on the dangling else in ALGOL 60," *Commun. ACM*, vol. 6, no. 8, pp. 460–462, Aug. 1963. [Online]. Available: http://doi.acm.org/10.1145/366707.367585

[34] S. Efftinge, J. Köhnlein, and S. Zarnekow, "Xtext language development framework," http://www.eclipse.org/Xtext/, last visited 06 June, 2018.