Code-based Design for Consistent Prototyping, Manufacture and Physical Modeling of Multimodal Robots

Martin Zoula Jan Faigl

Abstract— **The paper introduces `build123things`, a software tool for consistent robot design, manufacturing, and physical modeling. It is motivated by designing robotic systems with multimodal locomotion that require multiple simulation models to evaluate their motion capabilities using multimodal planning techniques. The tool uses a Hierarchical Assembly Graph and an algorithm to select the kinematic root, leveraging the Code-CAD library `build123d` and the OpenCASCADE geometry kernel. It employs explicit reference geometries to overcome the Topological Naming Problem (TNP) of existing mainstream Computer Aided Design (CAD) systems relying on implicit geometry element naming. Rich material annotations and assembly semantics enable use across design, simulation, and manufacturing, narrowing the sim-to-real gap. Its usage is demonstrated in a robotic arm design and a climbing magnetic robot case study, where `build123things` supports multimodal planning by implementing kinematic hierarchy transforms.**

I. Introduction

Robotic system design transforms conceptual ideas into digital models that can be validated in simulation, manufactured into prototypes, and, in turn, a complete production design. It involves assembling modeled components, aided by Computer Aided Engineering (CAE) software that integrates geometry creation, design management, mechanical and operating simulation, and manufacturing procedures [1]. Current research in CAE already focuses on numerous specialized problems, such as trustworthiness and traceability of design interchange [2], domain-specific design [3], or machining code optimization [4], to name a few here.

Articulated mobile robots, like humanoids or crawling systems, present simulation and manufacturing challenges due to their complexity and multimodal locomotion. These robots can use varying traction modes, such as leg switching in hexapod walkers [5] or acrobatic flight phases in humanoids [6]. Our focus, in particular, is on the inchworm-like magnetic robot *MInchW*, which moves by gripping iron beams or slithering as illustrated in Fig. 1. We explore flexible design approaches enabling multimodal planning [7], model-based control [8], and digital twin integration [9].

Besides, we also aim to address the Topological Naming Problem (TNP). Mainstream graphical Computer Aided Design (CAD) systems rely on implicit geometry naming, making them suffer from the TNP, where subsequent procedures lose geometric reference upon parameter change, which is not yet adequately solved [10]. Hence, the resulting designs are prone to various quality issues like syntactical and semantical errors, resulting in the loss of reference geometry elements [11].

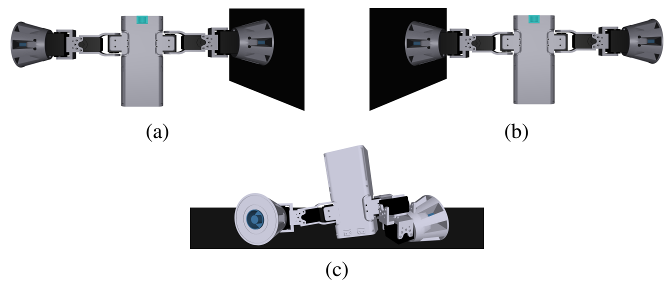


Fig. 1. Example locomotion modes of the *MInchW* robot designed in the proposed `build123things` system. Clockwise from top left: (a) rigidly attached (dry friction) by the "front" leg, (b) by the "rear" leg, and (c) crawling on the ground. As the robot is not perfectly symmetric, different physical model formulations are suitable for each such mode.

Code-CAD (CC) offers a text-based alternative to graphical CAD, using commands to define geometry through Constructive Solid Geometry (CSG) [12]. The CC approach ensures unambiguous, transparent design steps, treating the user as a design programmer. However, existing CC solutions lack systematic reference geometry support. Among existing systems, the `build123d` [13] is a Python library, exposing the OpenCASCADE Boundary Representation (B-REP) geometric kernel [14] in a suitable object-oriented fashion.

We extend [13] into `build123things` software library to support multimodal robotic design while mitigating the TNP. The key contributions are as follows.

1) We propose a methodology to consistently derive robot physical models for particular locomotion modes to support subsequent planning or simulation tasks.
2) We propose mitigation of the TNP by explicit reference geometry management based on the CC approach.
3) We provide an implementation of the above, with clear syntax and semantics, and model parameter management. We demonstrate the usage in a case study of the *MInchW* robot illustrated in Fig. 2.

The rest of the paper is organized as follows. The overview of the existing CC solutions and robotic planning is summarized in the following section. Section III introduces our proposed library on an example of a serial manipulator design. A case study on a magnetic climbing robotic system design is in Section IV. The paper is concluded in Section V.

The authors are with the Faculty of Electrical Engineering, Czech Technical University, Technická 2, 166 27, Prague, Czech Republic `{zoulamar|faiglj}@fel.cvut.cz`

The work has been supported by the Technology Agency of the Czech Republic under the project No. TN02000028 and by the European Union under the project ROBOPROX - Robotics and advanced industrial production (reg. no. CZ.02.01.01/00/22_008/0004590).

```python
class Robot (Thing):
    def __init__(self, length:float, a, b) -> None:
        super().__init__(material=PETG())
        # Def. mount point by xyz [mm] and rpy [deg]
        self.leg_front = MountPoint(Location(
          (length, 0, 0), (0, 0, -90) ))
        # Attach with DoF by naming mating mounts
        Revolute(self.leg_front, LegAssembly.mount)
        # Define unambiguous reference geometry
        self.groundplane = fillet(Rectanlge(a, b),3)
```

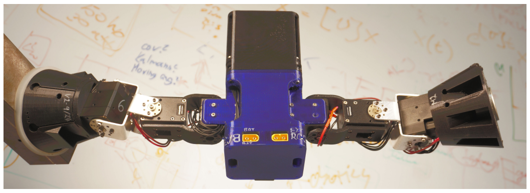


Fig. 2. An example of the assembled physical robot *MInchW* and an example of a robot definition in the proposed `build123things` Code-CAD (CC) system.

II. Related Work

Modern manufacturing relies on digital product management throughout the product lifecycle, aligning with Industry 4.0 goals [15] of automation. Rapid prototyping benefits from rich, annotated data for early flaw detection [16]. Although advanced CAD systems support the lifecycle [11], a gap remains in supporting mobile articulated robotics.

Two geometry paradigms are used in CAD systems: Constructive Solid Geometry (CSG), which builds shapes from primitives using boolean operations [12], and Boundary Representation (B-REP), which defines shapes via their bounding manifolds. Both support parametric design, where the resulting geometry is computed based on limited input parameters. While CSG geometric kernel implementation, such as CGAL [17], encode shapes as anonymous meshes, B-REP kernels, such as OpenCASCADE [14], preserve the geometric semantics, making it preferable for detailed access to the object geometry and topological entities.

The Topological Naming Problem (TNP) arises when even small changes in defining parameters alter or remove resulting geometry, making consistent reference difficult. Since geometry naming is important for user experience [18], it is addressed by heuristics in existing graphical CAD systems; however, it lacks a complete solution [10]. In rigid component assembly, relative positions can be defined by constraint satisfaction programming [19] or directly specified by the designer. These relations form a Hierarchical Assembly Graph (HAG), a directed acyclic graph where nodes represent components and edges define "subcomponent-of" relations, enabling automated assembly planning and execution for manufacturing [20].

A. Existing Code-CAD (CC) Systems and Libraries

We briefly review existing Code-CAD (CC) systems to contextualize our proposed approach. The Onshape CAD system [21] introduces domain-specific language lacking object-oriented features. TinkerCAD [22] and BlocksCAD [23] use Scratch-style [24] graphical programming. FreeCAD enables bidirectional programming [25] in interactive Python sessions to expose geometry manipulation, allowing linking geometry source code and its visualization [18]. Curv [26] uses a custom functional representation approach to the geometry kernel. JSCAD [27] uses JavaScript to implement the CSG paradigm, while ImplictCAD [28] implements a custom geometric kernel in the Haskell language. Besides, field-specific CC systems exist, such as Paramak [29] and STOK-A [30] for tokamak geometry generation, or TiGL [31] for aircraft design with advanced surface modeling.

OpenSCAD [32] uses a custom functional language based on the CSG paradigm with support for parametrized modules. However, global variable declarations limit flexibility, they lack object annotations, and geometry referencing requires duplication. Assemblies rely on basic parametrization without joint semantics, limiting kinematic analysis. Despite the drawbacks, it is found suitable for math and science education [33].

OpenCASCADE-based FreeCAD, with its Python interface, raised a family of Python-based CC solutions. The `build123d` library recently evolved from CadQuery [34], exposing the kernel in a concise, fully object-oriented manner furnished with a rich Python interface. Similarly to the CC systems, the library exposes constructive operations without robust reference geometry handling. Still, we opted to use `build123d` as a modeling backbone for the proposed library as it best fits our needs.

B. Code-CAD for Robotics

Beyond simulation and visualization, high-fidelity robot models are essential for machine learning [35] and model-based control [8]. For example, Chignoli et al. [6] demonstrated a humanoid robot with three (acrobatic) control modes (takeoff, flight, and landing), each requiring a distinct dynamic model. Multimodal planning is shown for a humanoid navigating zero gravity using varying grip configurations, with each mode representing a different set of constraints on the robot's movement [7]. Here, a mode is defined as a partially constrained robot configuration.

Similarly, our *MInchW* robot can operate in three locomotion modes: two resembling fixed-base serial manipulators and one as a floating-based system for slithering. Mode transitions, including slippage, remain open challenges. Using the proposed `build123things`, we transform the mechanical design into different simulation setups in MuJoCo [8], demonstrating how the developed system streamlines the otherwise manual process required for multimodal planning with complex robots.

III. Robot Design in the Developed Code-CAD `build123things` Library

The developed `build123things` library follows the Code-CAD principles, where designers define the geometry and assemblies using a programming language. In the `build123things`, as an extension to the `build123d`,

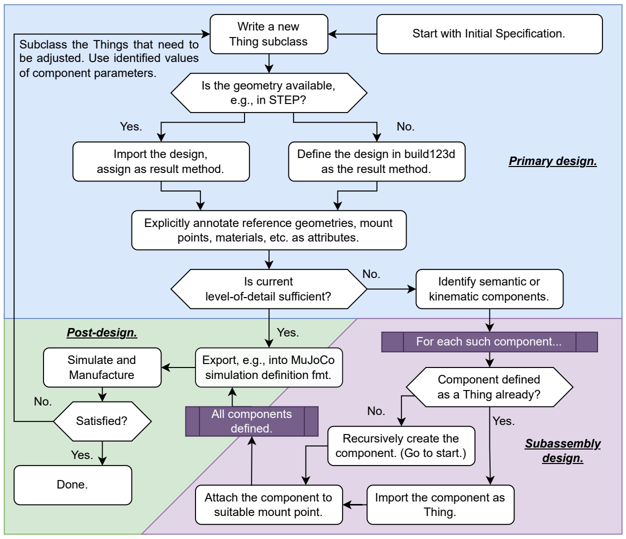


Fig. 3. The suggested design workflow with the proposed library.

Python is used, and designers define all designs as subclasses of the proposed `Thing` class. Each such subclass represents a parametric family of mechanical parts or (sub)assemblies, collectively called components. The family parameters map to the arguments of the constructor, the `__init__` method in Python. Instantiating such components implies a hierarchical assembly of immutable geometries. Once components and their assemblies are defined, the model can be exported for further simulation-based validation and manufacturing, e.g., using 3D printing. The suggested design workflow is summarized in Fig. 3.

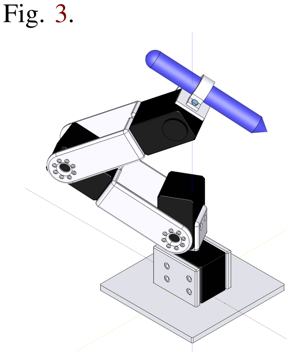


Fig. 4. The robotic manipulator created with the proposed library. Black blocks denote the Dynamixel XM-430 servomotors linked by white 3D-printable fixures, with the end effector being a dummy pen for drawing.

In the rest of the section, the `build123things` library is introduced by an exemplar design of a robotic manipulator with 4 Degrees of Freedom (DoF) in yaw-pitch-pitch-pitch configuration. The manipulator, depicted in Fig. 4, can reach a point in a plane while maintaining the angle of attack of a pen to draw. The presented example also serves as a demonstration of possible `build123things` usage in educational tasks, as the manipulator is fully 3D-printable.

A. Component Design

A component design is introduced by the example of the blue pen in Fig. 4, an elementary component in our manipulator. It is defined as a partially rounded cylinder of length l and diameter d, joined with a translated cone. The component family code can look as follows.

```
class PenDummy( Thing ):
↪ def __init__( self, l = 100.0, d = 15 ):
↪↪ super().__init__( PETG( Color( .1, .1, .9, 0)))
↪↪ self.body = fillet( Cylinder( l, d/2 ) ... ) +
    Location(...) * Cone( d/2, 0, l/10)
↪↪ self.groundplane = Circle( d/2 )
↪ def result(self): return self.body
```

In the object initialization, the component is furnished with the reference geometries (attr. `groundplane`), expressed with respect to (w.r.t.) the component's implicit reference frame. Further, each component declares its authoritative geometrical representation in the `result` method as the geometry or `None` in the case of pure assemblies.

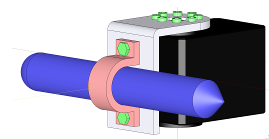


Fig. 5. End Effector (EE) is made of a white L-shaped fixture. The Ω-shaped cap (red) holds the pen (blue) in between; both components are subassemblies of the EE. The servomotor model (black) is a reference object (not part of the EE assembly). The green screws are servomotor subcomponents, translated to the EE reference frame using the expression syntax to extrude holes in the created fixture precisely.

The EE component is to attach the defined pen to the rotor of the Dynamixel XM-430 servomotor with an L-shaped fixture screwed to the rotor and fastened by an Ω-shaped cap, see Fig. 5. In the constructor `__init__`, we can use an existing library model of the servomotor and place it in the local reference frame for further reference by assigning it as an attribute.

```
self.servo_ref= XM430().move(Location(...)) # black
```

The library handles component movement symbolically via the `TransformResolver` object. The moved object is wrapped in `TransformResolver`, hijacking the attribute access to transform attribute values to the current local reference frame automatically. The language-level expression maps to the spatial transformation as illustrated by the green screw in Fig. 5 and is defined as follows.

```
self.servo_ref.screw_right_bottom_front(5) # green
```

Note, the `build123d` offers a selector logic to access topological faces or vertices of a geometry. However, relying on such features is prone to the TNP as there is generally no guarantee about the selection result or ordering. Hence, we suggest defining reference geometries as original entities dependent directly on input parameters to mitigate the TNP. A designer may create reference geometry by assigning arbitrary attributes except for the following reserved attributes.

- The `__parameters__` attribute stores values passed to `__init__`, facilitating the `adjust` functionality.
- The `__material__` attribute defines appearance and mechanical properties of the `result`.
- The `mass`, `volume`, and `matrix_of_inertia` methods determine respective quantities.

The servomotor is used to define a sweeping curve and to cut openings for screws in the fixture. The sweep operation may be ill-conditioned (geometry kernel error), or the result might collide with the servomotor while rotating (semantic error). Such risks are prevented by language features like the `assert` or `try` statements, encapsulating the invalid parameter handling in the class.

```
assert thick > 0 and lngth > servo_ref.rotor_radius
curve = Polyline(self.servo_ref...position, ...
     position + Vector(0, cl, 0), ... ) # tmp val
profile = ... * Rectangle(self.servo_ref.p.
     rotor_radius_1 * 2 + thickness * 2, ...) # tmp
try: self.body = sweep(profile, curve, transition=
     ROUND) + extrude(Circle(...),...) # ref val
```

The cap component is defined similarly and rigidly attached to the EE. Matching screw holes are subtracted using the expression transform.

```
body -= self.cap_pen.screw_2.body_hull
```

Finally, the EE declares a mount point to attach to the robot.

```
self.mount = MountPoint(self.servo_ref.rotor_center
     .location * MOUNTING_LOCATION)
```

Existing model definitions may be altered in two ways. First, subclassing may reduce or change the component family parameter space. Hence, designers can favor generic designs from which specialized cases are derived. The second method is cloning a component instance using `adjust` while changing the named parameters. Thus, one may obtain, e.g., a slightly bigger version of the servomotor to facilitate clearances that account for manufacturing inaccuracy.

```
xm430_with_clearance = XM430().adjust(
    rotor_radius_2__add = 0.2,
    width__add = 0.2, height__add = 0.2)
```

The design process can be summarized as using the language and library features to define the result and reference geometries in a salient way that is robust to parameter change.

B. Manipulator Assembly

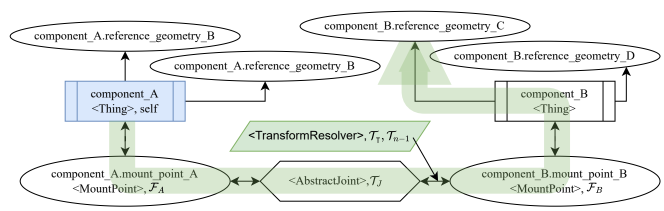
The manipulator itself comprises a base link with yaw-motion servomotor, shoulder-link, humerus-link, forearm-link, and EE. Following the concept of hierarchical assembly, each named component is defined as a subassembly of the previous ones. For instance, the shoulder link's `__init__` can be defined as follows.

```
# Fixture to rigidly bind yaw and 1st pitch servos.
self.connector = MountPoint(bd.Location())
Rigid(self.connector, Connector_Yaw_Pitch().origin)
# 1st pitch servo, part of shoulder assembly.
self.servo = MountPoint(self.connector.
     servo_pitch_mount)
Rigid(self.servo, XM430().top)
# Subassembly with all links up to end effector.
self.humerus = MountPoint(self.servo.rotor_center)
self.joint = Revolute(self.humerus, LinkHumerus().
     mount_by_braces)
self.joint.set(42) # Joint handle sets joint transf
# Shoulder may be mounted to base yaw servo.
self.base = MntPoint(slf.connector.servo_yaw_mount)
```

The `MountPoint`-type attribute defines an auxiliary reference frame $\mathcal{F}$ w.r.t. the implicit component reference frame. Henceforth, `AbstractJoint` instance (such as `Rigid(...)` or `Revolute(...)`) symbolically aligns components A and B by their respective mount points, adding an arbitrary, possibly parametrized, intermediate joint transformation $\mathcal{T}_J$. Formally, the alignment is read as

$$\mathcal{T}_{AB} = \mathcal{F}_{n-1} \times \mathcal{F}_A \times \mathcal{T}_J \times \mathcal{T}_\mathsf{T} \times \mathcal{F}_B^{-1},$$

where $\mathcal{F}_{n-1}$ accumulates the transformation from expression evaluation, and $\mathcal{T}_\mathsf{T}$ is the fixed transformation with the rotation defined by Euler angles (180°, 0°, 90°), and zero translation. The fixed transformation aligns the z-axes of the reference frames parallel but opposing, constituting a standardized way of the mount points orientation compatible with other conventions like the Denavit-Hartenberg notation. Both symbols $\mathcal{T}$ and $\mathcal{F}$ may be interpreted as homogeneous coordinate transforms. The designer may enjoy expression-transformer semantics for the reference placement as the attribute access uses the `TransformResolver` object; see illustration depicted in Fig. 6.




```
reference_geometry_C_in_self_reference_frame =
     self.mount_point_A.reference_geometry_C
```

Fig. 6. Two components arranged via `AbstractJoint` and respective `MountPoints`. An expression-transform (in green and below diagram) yields a `TransformResolver` that ultimately interprets the `reference_geometry_C` w.r.t. the implicit frame of `component_A`.

Mount points distinguish a single outbound or multiple inbound assembly slots. Hence, they define the Hierarchical Assembly Graph (HAG) where components are nodes and edges are determined implicitly by investigating which mount points are actually occupied with `AbstractJoint` instances. `Thing` instance memoization ensures that each component family instance is stored in the memory only once, making the design analysis easy. Extensible operations on assemblies are supported via the `walk` enumerator method, implementing HAG traversal with optional heed for hierarchy. Such operations include exporting procedures, visualizing the HAG using the `graphviz` library [36], or preparing the `mjcf` MuJoCo [8] model description files.

Complete manipulator code and more examples are part of the library [37]. Note that the current library implementation is not performance-optimized, sacrificing computation speed to implement the required features and validate the proposed solution. As opposed to the integrated CAD philosophy advocated in [1], the proposed library is a modular software supporting robust data interchange. At the current state, the library validates the proposal for ease of CC-based designs. The library's core module comprises around 1000 code lines and around five-fold when including miscellaneous code.

IV. Case Study on Magnetic Crawler Robot

The proposed library was used to design and prototype the *MInchW* robot intended to inspect iron structures. The robot is equipped with two coil-regulated permanent magnets with the nominal pull of 300 N and actuated by six Robotis Dynamixel X-series servomotors. Power can be drawn from six built-in 18650 Lithium-ion cells, promising over an hour of uptime. The resulting robot weights 2.08 kg and spans 48.5 cm. The illustration of the robot assembly is depicted in Fig. 2, and a simplified Hierarchical Assembly Graph (HAG) of the final assembly is visualized in Fig. 7. The final robot prototype is shown in Fig. 2.

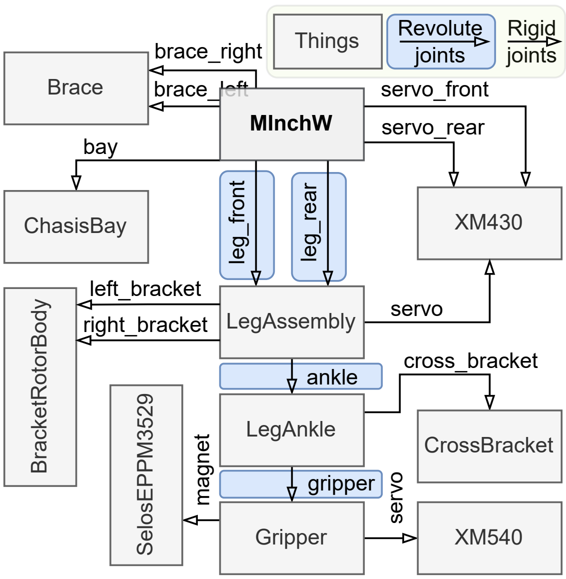


Fig. 7. The Hierarchical Assembly Graph (HAG) of the MInchW robot, without the electronics and fasteners. Nodes (rectangles) identify a component used in the final robot assembly. Edges represent the "is-subassembly-of" relation between components. Each edge represents a local spatial transformation that aligns the two components by specifying particular mount points.

The root element of the primary design is the middle section of the robot (the grey body in Fig. 8), while the legs are attached relative to the root as sub-assemblies with their local

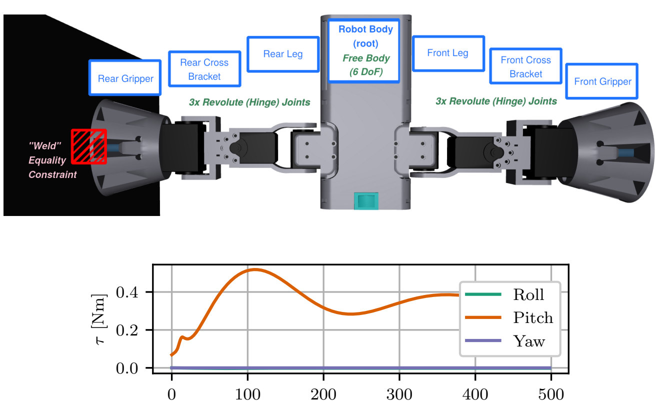


Fig. 8. Default kinematic structure used to simulate rigid attachment via the "weld" constraint (top) includes parasite dynamics (bottom) that require around 500 simulation steps to settle.

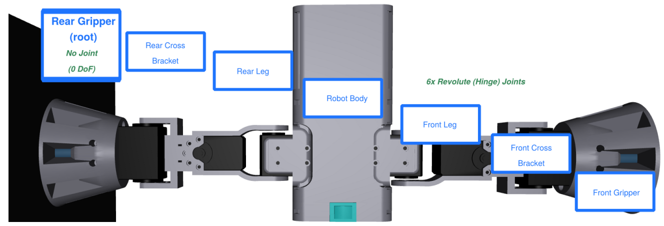


Fig. 9. Transforming the kinematic representation by selecting the kinematic root in the environment-attachment point leads to an equivalent of a serial manipulator and allows using standard, efficient inverse dynamics solvers. Our proposed system is able to perform such a kinematics transform.

hierarchy. Turning the hierarchy directly into a simulation model is useful for the wiggling locomotion mode (Fig. 1c), where the central robot element acts as a free body with articulated legs. Simulating iron-attached locomotion requires rigidly fixing a magnet in space. With the original kinematic hierarchy, it can be done using soft constraints, such as the "weld" constraint in MuJoCo. However, such constraints introduce parasite model dynamics, requiring hundreds of simulation steps to settle a static simulation scenario; see the plot in Fig. 8. The `build123things` enables automatic reformulation of the kinematic tree by selecting a root in the attachment location, enabling solving forces and dynamics of the attached robot with a single standard inverse dynamics computation as illustrated in Fig. 9.

Both cases were simulated in the MuJoCo [8]; the `build123things` provided a `XML` file with the robot structure and complementary `STL` mesh files. The file was then included in another file that defined the simulation parameters and scene. Hence, we are able to produce a set of consistent *MInchW* dynamic models, each suitable for a particular operating configuration or the so-called mode in the sense of multimodal planning [7].

The design comprises 26 large components, omitting fasteners like screws, cables, and electronics. We also used `build123things` to generate `STL` files as a basis for Fused Deposition Modeling manufacturing; such an operation took, using the non-optimized implementation of the `build123things`, about 15 s on the Intel Core i7-10700 processor and about 412 MB of memory. During rapid prototyping, some of the parts turned out to be of different weights than initially anticipated, specifically the 3D printed parts with low infill. It is addressed by subclassing respective `Things` and overriding weight or inertia attributes as summarized in Fig. 3. Hence, the `build123things` exporter produces a consistent dynamic model for the MuJoCo [8] software by expanding the native hierarchical assembly.

Future work: Based on experience with the developed `build123things` and regarding the related work, the possible future work is summarized as follows. The open fundamental feature is the representation of the internal component structure as indicated in [12], which affects modeling fidelity. We identified a need to seamlessly represent the variable level of detail where both detailed designs with small components like screws or discrete electronic parts may be efficiently kept alongside the macroscopic designs involving only compound rigid bodies with enumerated matrices of inertia. The library currently lacks features like non-HAG assemblies, generalized part interfaces, tolerance analysis, and measurement unit analysis. Assembly cycle detection and verification of the referencing geometries through joints are also desirable features.

V. Conclusion

We present `build123things`, a novel Code-CAD library suitable not only for designing articulated mobile robots with multimodal locomotion. It mitigates the Topological Naming Problem through explicit reference geometry naming and supports a hierarchical assembly model with explicit joint semantics and rich annotations. Components are easily parametrized in Python, and models can be exported for simulators with flexible kinematic root selection to enable multimodal planning. The library usage and its efficiency are demonstrated in the *MInchW* robot case study. Besides, as a text-based system, `build123things` supports standard version control. It advances Code-CAD by integrating robust design semantics, mitigating the Topological Naming Problem, and implementing proven yet overlooked principles absent in similar systems. Additionally, arbitrary kinematic root rebasing enables seamless model export for robotic simulators, supporting multimodal planning and validating diverse locomotion modes.

References

[1] U. Rembold and R. Dillmann, Eds., *Computer-Aided Design and Manufacturing*, 2nd ed., ser. Symbolic Computation. Springer, 1986.

[2] T. Hedberg Jr., M. Helu, S. Krima, and A. Barnard Feeney, "Recommendations on ensuring traceability and trustworthiness of manufacturing-related data," National Institute of Standards and Technology (NIST), Tech. Rep., 2020.

[3] X. Dai and Y. Hong, "Fabric mechanical parameters for 3D cloth simulation in apparel CAD: A systematic review," *Computer-Aided Design*, vol. 167, p. 103638, 2024.

[4] R. B. Käsemodel, A. F. de Souza, R. Voigt, I. Basso, and A. R. Rodrigues, "CAD/CAM interfaced algorithm reduces cutting force, roughness, and machining time in free-form milling," *The International Journal of Advanced Manufacturing Technology*, vol. 107, no. 3–4, p. 1883–1900, 2020.

[5] P. Čížek, M. Zoula, and J. Faigl, "Design, construction, and rough-terrain locomotion control of novel hexapod walking robot with four degrees of freedom per leg," *IEEE Access*, vol. 9, pp. 17 866–17 881, 2021.

[6] M. Chignoli, D. Kim, E. Stanger-Jones, and S. Kim, "The MIT humanoid robot: Design, motion planning, and control for acrobatic behaviors," in *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*, 2021, pp. 1–8.

[7] Z. Kingston and L. E. Kavraki, "Scaling multimodal planning: Using experience and informing discrete search," *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 128–146, 2023.

[8] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2012, pp. 5026–5033.

[9] F. Tao, B. Xiao, Q. Qi, J. Cheng, and P. Ji, "Digital twin modeling," *Journal of Manufacturing Systems*, vol. 64, pp. 372–389, 2022.

[10] S. H. Farjana and S. Han, "Mechanisms of persistent identification of topological entities in CAD systems: A review," *Alexandria Engineering Journal*, vol. 57, no. 4, pp. 2837–2849, 2018.

[11] C. González-Lluch, P. Company, M. Contero, J. D. Camba, and R. Plumed, "A survey on 3D CAD model quality assurance and testing tools," *Computer-Aided Design*, vol. 83, pp. 64–79, 2017.

[12] C. D. Toth, J. O'Rourke, and J. E. Goodman, Eds., *Handbook of discrete and computational geometry*, 3rd ed., ser. Discrete Mathematics and Its Applications. Productivity Press, 2017.

[13] R. Maitland, "build124d," cited on 2025-05-02. [Online]. Available: https://github.com/gumyr/build123d

[14] "OpenCASCADE Technology," cited on 2025-02-02. [Online]. Available: https://dev.opencascade.org/doc/overview/html/index.html

[15] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry 4.0," *Business & Information Systems Engineering*, vol. 6, no. 4, p. 239–242, Jun. 2014.

[16] A. Matta, D. R. Raju, and K. Suman, "The integration of CAD/CAM and rapid prototyping in product development: A review," *Materials Today: Proceedings*, vol. 2, no. 4–5, pp. 3438–3445, 2015.

[17] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Pion, and S. Schirra, "2D and 3D linear geometry kernel," in *CGAL User and Reference Manual*. CGAL Editorial Board, 2023, cited on 2024-02-29. [Online]. Available: https://doc.cgal.org/5.6/Manual/packages.html#PkgKernel23

[18] A. Mathur, M. Pirron, and D. Zufferey, "Interactive programming for parametric CAD," in *Computer graphics forum*, vol. 39, no. 6. Wiley Online Library, 2020, pp. 408–425.

[19] R. Sodhi and J. U. Turner, "Towards modelling of assemblies for product design," *Computer-Aided Design*, vol. 26, no. 2, pp. 85–97, 1994.

[20] Y. Pane, M. H. Arbo, E. Aertbeliën, and W. Decré, "A system architecture for CAD-based robotic assembly with sensor-based skills," *IEEE Transactions on Automation Science and Engineering*, vol. 17, no. 3, pp. 1237–1249, 2020.

[21] "FeatureScript introduction," cited on 2024-02-02. [Online]. Available: https://cad.onshape.com/FsDoc/

[22] "Codeblocks – Tinkercad," cited on 2024-02-02. [Online]. Available: https://www.tinkercad.com/codeblocks

[23] "BlocksCAD," cited on 2024-02-02. [Online]. Available: https://www.blockscad3d.com/editor/

[24] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computating Education*, vol. 10, no. 4, nov 2010.

[25] J. F. Gonzalez, D. Kieken, T. Pietrzak, A. Girouard, and G. Casiez, "Introducing bidirectional programming in constructive solid geometry-based CAD," in *ACM Symposium on Spatial User Interaction*, 2023, pp. 1–12.

[26] "doug-moen/curv: a language for making art using mathematics," cited on 2024-02-15. [Online]. Available: https://codeberg.org/doug-moen/curv

[27] "JSCAD – JavaScript CAD," cited on 2024-01-23. [Online]. Available: https://github.com/jscad/OpenJSCAD.org

[28] "ImplictCAD.org," cited on 2024-01-23. [Online]. Available: https://implicitcad.org/

[29] J. Shimwell, J. Billingsley, R. Delaporte-Mathurin, D. Morbey, M. Bluteau, P. Shriwise, and A. Davis, "The Paramak: automated parametric geometry construction for fusion reactor designs." *F1000Research*, vol. 10, no. 27, 2021.

[30] A. Gabrijel and A. Čufar, "STOK-A tool for parametric modeling of simple tokamaks," in *International Conference Nuclear Energy for New Europe*, 2022, pp. 1008.1–1008.10.

[31] M. Siggel, J. Kleinert, T. Stollenwerk, and R. Maierl, "TiGL: An open source computational geometry library for parametric aircraft design," *Mathematics in Computer Science*, vol. 13, no. 3, p. 367–389, 2019.

[32] "OpenSCAD – The Programmers Solid 3D CAD Modeller," cited on 2024-02-02. [Online]. Available: https://openscad.org

[33] J. Horvath and R. Cameron, *3D Printed Science Projects Volume 1: Ideas for Your Classroom, Science Fair, or Home*. Apress, 2024.

[34] "CadQuery/cadquery: A python parametric CAD scripting framework based on OCCT," cited on 2024-03-12. [Online]. Available: https://github.com/CadQuery/cadquery

[35] A. Plaat, W. Kosters, and M. Preuss, "High-accuracy model-based reinforcement learning, a survey," *Artificial Intelligence Review*, vol. 56, no. 9, p. 9541–9573, Feb. 2023.

[36] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[37] "The build123things Repository," cited on 2024-09-12. [Online]. Available: https://github.com/zoulamar/build123things