2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

Domain-specific languages for kinematic chains and their solver algorithms: lessons learned for composable models

Sven Schneider*[,1,2] and Nico Hochgeschwender[1] and Herman Bruyninckx[2,3,4]

***Abstract*— The Unified Robot Description Format (URDF) and, to a lesser extent, the COLLAborative Design Activity (COLLADA) format are two of the most popular domain-specific languages (DSLs) to represent kinematic chains in robotics with support in many tools including Gazebo, MoveIt!, KDL or IKFast. In this paper we analyse both DSLs with respect to their *structure* and *semantics* as seen by tools that produce or consume such representations. For the former, we notice a tight coupling of various unrelated domains like kinematics and dynamics with visualisation, control or even specific simulators. For the latter, a key insight is that both DSLs target human developers and leave important design decisions like the choice of joint attachment frames implicit or hidden in the documentation.**

The lessons learned from this analysis guide us to an improved interchange format by designing *composable*, loosely coupled models with *complete* metamodels that unambiguously define the model semantics. We substantiate our findings with concrete examples. Furthermore, we compose solver algorithms on top of the kinematic chain representation. As a consequence of the above analysis and decomposition we can systematically apply structure- and semantics-conserving model-to-code transformations to those algorithms.

I. Introduction

URDF [1] is a robot description language with support in tools like Gazebo [2], RViz [3], MoveIt! [4] or KDL [5]. Users benefit from it by having to learn only a single language and then reuse their robot specifications among those tools. For tool developers the format provides an interface that clearly defines which features they must support and a library to easily consume a URDF document. The uptake of URDF in various tools clearly indicates the need for a common description format and justifies the overhead that the language developers have invested into URDF's design.

URDF features some design choices that foster *composability*, that is to add new meaning representations without any changes to the already represented meaning. We analyse URDF from a structural and from a semantics perspective while providing some examples but mostly counterexamples of composability, i.e. anti-patterns (AP). We exemplify our observations along the excerpt in Listing 1. A `teletype font` refers to specific keywords in that example.

*a) Underutilize identifiers (**AP1**):* From a structural perspective, the listing shows *(i)* how a `joint` element composes two links by referring to — with what we call a symbolic pointer — a `parent` and `child` link; and *(ii)* how an identifier or `name` (here, of the two links) is a necessary condition for composability. The same mechanism enables URDF's extensions to attach transmissions or joint state to joints and sensors to links. In the same way the Semantic Robot Description Format (SRDF) [6] annotates parts of a kinematic chain, for example, by identifying a subset of links or joints to belong to the "left" or the "right" arm. The first, structural anti-pattern to inhibit composability and hence the further evolution of URDF is the *inconsequent* application of identifiers: elements nested inside `robot`, `link` and `joint` lack names and hence are not addressable.

Listing 1: Excerpt of a kinematic chain model in URDF.

```
<robot name="robot1">
  <link name="link1">
    <inertial>
      <mass value="1"/>
      <inertia ixx="100" ixy="0" ... />
    </inertial>
    <visual>...</visual>
    <collision>
      <geometry><mesh filename="example.dae"/></geometry>
    </collision>
    <gazebo>...</gazebo>
  </link>
  <joint name="joint1" type="revolute">
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <parent link="link1"/>
    <child link="link2"/>
    <limit effort="30" velocity="1.0" lower="-2.1" .../>
    <safety_controller k_position="15"
                       soft_lower_limit="-2.0" .../>
  </joint>
</robot>
```

*b) Single-conformance (**AP2**):* In an Extensible Markup Language (XML) [7] document any tag, such as `robot` or `link`, represents *the* type of an element. We call this single-conformance, that is, the element must satisfy the structural and semantic constraints associated with that single type. The joint in URDF is an example that instead requires multi-conformance. URDF deliberately introduces multi-conformance but must rely on heterogeneous syntax: the `joint` tag and the `revolute type` attribute.

*c) Artificial structural constraints (**AP3**):* As a tree-structured format, URDF has exactly one root element (`robot`) that "closes" the composition hierarchy to the top. Yet, the more general logical model is a graph that does not feature a generic root. Moreover, URDF artificially limits the supported models: `damping`, `friction` and the safety controller (`k_position`) represent linear models where non-linear models are often more realistic or suitable. Moreover, coordinate representations of orientations are restricted to roll-pitch-yaw Euler angles (`rpy`), whereas the inertia matrices must be represented by individual entries

* Corresponding author, `sven.schneider@h-brs.de`
[1] Dept. of Computer Science, Bonn-Rhein-Sieg UoAS, Germany.
[2] Dept. of Mechanical Engineering, KU Leuven, Belgium.
[3] Dept. of Mechanical Engineering, TU/e Eindhoven, Netherlands.
[4] Flanders Make, Belgium.

2023 IEEE International Conference on Robotics and Automation (ICRA) | 979-8-3503-2365-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICRA48891.2023.10160474

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

(ixx, ixy, ...). Converting to URDF's convention (for example, from rotation matrices) is an error-prone activity for users that would better be delegated to a computer. Further, URDF supports different levels of detail by *(i)* "soft" (soft_lower_limit) and "hard" (limit) position limits for control or monitoring; as well as *(ii)* lumped inertial, low-detail collision and high-detail visual shape representations. Yet, the resolution, i.e. the number of levels, is fixed in URDF. We also find that *(i)* at most one measurement of any physical quantity such as a pose (origin), mass or inertia is supported; and *(ii)* any such measurement must have exactly one coordinate representation, that is a choice of reference coordinate system and physical units.

*d) Tight coupling (**AP4**):* While all the previous choices *are* valid, they are not the only ones. The final structural problem amplifies the previous challenges: URDF couples a variety of unrelated domains deep down in the composition hierarchy of the root element and each of those domains consist of widely variable models themselves: a link contains elements related to dynamics (inertia), visualization (visual), motion planning or simulation (collision) and even simulator-specific tags (gazebo). Similar observations apply to the joint element.

*e) Incomplete metamodel (**AP5**):* From a semantics perspective, the language designers have taken the additional step of formally representing some cardinality and data type constraints in an XML schema [8]. Yet, this schema definition prohibits vendors from adding custom elements or tags to a URDF document and hence is inconsistent with the manually-implemented URDF parser. Beyond those formal constraints, URDF is only easy to use for human developers as they can read in the documentation that, for instance, distances (xyz) should be specified in metres, whereas rpy orientations are usually measured in radians and follow the fixed-frame convention. Similarly, a human quickly notices the different meanings of strings: scalars (ixx or ixy), ordered lists of coordinates (xyz and rpy), reference to a file (filename) or symbolic pointers (link). But for a computer all this information remains unavailable because URDF's metamodel, that is a computer-readable version of the documentation, does not exist.

*f) Implicit assumptions (**AP6**):* Furthermore, URDF prematurely composes and hence over-constrains some concepts. Neither effort nor velocity are hard limits of actuators. Electric motors may be overdriven temporarily to provide short bursts of force, whereas a limit on velocity physically emerges due to friction or is artificially imposed by a motor controller. Additionally, URDF couples a joint, a pure motion constraint, to its control (safety_controller): only actuators can be controlled.

Finally, URDF omits symbolic representations such as the frames of pose (origin) relations, i.e. each quantity is defined merely via its coordinates. The Geometric Relations Semantics [9] demonstrates how composing both representations helps developers to identify common programming errors. Another consequence of those implied elements is that URDF can only represent serial chains and kinematic trees, but not parallel kinematic chains: this would require locating two or more joint frames on the same link.

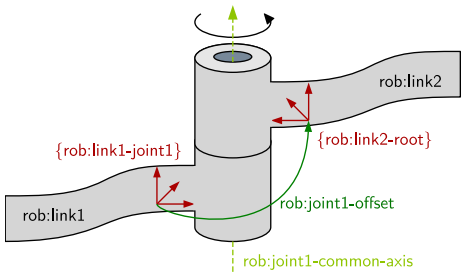


Fig. 1: Graphical model of revolute joint rob:joint1.

Most of those anti-patterns also prevail in COLLADA [10]. While it promotes better use of identifiers and symbolic pointers, it still limits composability to specifically designed extension points. Yet, two noteworthy features are a document-wide context to explicitly define units of measurement and its support for parallel chains. The Simulation Description Format [11] takes inspiration from URDF and hence inherits most of the above anti-patterns. However, three proposals aim at selectively increasing composability: *(i)* the addition of further coordinate representations [12]; *(ii)* support for custom schema extensions [13]; and *(iii)* the reification of frames [14] (albeit while still treating relations between frames as attributes of those frames). In general, we also observe those anti-patterns in the Universal Scene Description (USD) [15]. Still, USD supports multi-conformance via vendor-definable metadata which is e.g., used to tag rigid bodies, add impedances or collision information to objects. USD models mostly have nested hierarchies, yet support for symbolic pointers means that it can represent parallel chains.

The remainder of this paper is structured as follows. In Section II we present examples of our composable models. Section III critically discusses our approach while Section IV concludes the paper and reviews the future work.

A. Contributions

The major results and contributions of our paper comprise:

- The previous analysis and description of anti-patterns.
- We present composable models and metamodels of kinematic chain structures but also algorithms that do not suffer from those anti-patterns.
- We have developed tools that interact with those models for constraint checking and systematic code generation.
- We reuse mature standards and tools as much as possible to avoid the "not-invented-here syndrome".

More details freely available via:

https://github.com/comp-rob2b/modelling-tutorial
https://github.com/comp-rob2b/modelling-tools

II. Composable Models for Kinematic Chains

In this section we will employ the revolute joint depicted in Fig. 1 as a running example. Its main entities are the two links with attached frames whose relative motion the joint constrains. A complete specification additionally requires

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

Listing 2: Position relation in JSON-LD.

```
1 {
2   "@context": [
3     "https://comp-rob2b.github.io/metamodels/geometry/
          spatial-relations.json",
4     "https://comp-rob2b.github.io/metamodels/qudt.json"
5   ],
6   "@id": "rob:joint1-offset",
7   "@type": "Position",
8   "of": "rob:link2-root-origin",
9   "with-respect-to": "rob:link1-joint1-origin",
10  "quantity-kind": "Length"
11 }
```

Listing 3: Position coordinate representation

```
1 {
2   "@context": {
3     "coord": "https://.../coordinates#",
4     "PositionReference": "coord:PositionReference",
5     "of": { "@id": "coord:of-position",
6             "@type": "@id" }, ...
7   },
8   "@id": "rob:joint1-offset-coord",
9   "@type": [ "3D", "Euclidean", "PositionReference",
10             "PositionCoordinate", "VectorXYZ" ],
11  "of": "rob:joint1-offset",
12  "as-seen-by": "rob:link1-joint1",
13  "unit": "M",
14  "x": 0.0, "y": 0.0, "z": 0.42
15 }
```

(i) the offset between the two frames' origins; *(ii)* the axis of rotation, that is a directed line; *(iii)* an assignment of the handedness; and *(iv)* a description of the zero configuration.

A. Structural model representation: JSON-LD & SHACL

Graphs are the workhorse of knowledge representation [16]. Yet, only few standardized graph interchange formats exist, with the older XLink [17] and the relatively recent JSON-LD [18] being the most popular ones. Here, we decide for the latter due to its design for composability — it can be retrofitted to existing JSON data on demand — and its native support for multi-conformance. JSON-LD's objective is to bring together the JSON [19] world from web development with the Linked Data [20] principles from the Semantic Web [21] to foster interoperability between loosely-coupled providers of semi-structured data. To this end, JSON-LD introduces a vocabulary, i.e. a collection of keywords with well-defined semantics, to transform a tree-structured JSON document into a graph. Loose coupling of Linked Data refers to documents scattered throughout the Web. Consequently, the models below may physically be included in a single file or distributed among different files.

Listing 2 shows an excerpt of a JSON-LD model. It is easily recognizable as a JSON document that separates into JSON-LD metadata (lines 2–7) and data (lines 8–10). The JSON-LD metadata is three-fold and indicated by `@`-prefixed keywords (highlighted in purple colour): the `@context`, the `@id` and the `@type`. The `@context` defines all the metadata — or more correctly, the metamodels — that is required to correctly interpret the data. It can take different shapes. Here, it is just a list of Internationalized Resource Identifiers (IRIs) that must be dereferenced and imported by a JSON-LD interpreter. Further, more concrete examples follow below. Next, the `@id` keyword assigns an identifier to this model. Any IRI is a valid JSON-LD identifier, but only absolute IRIs must be unique. To this end, the context offers some configuration options that control the mapping from relative to absolute IRIs. As a best practice JSON-LD suggests assigning identifiers to all entities so that they become addressable from other models. This design fosters forward-compatibility even if developers do not foresee such composability in the current application context. When a JSON-LD interpreter encounters an anonymous model, it will generate a temporary, internal identifier. Finally, `@type` symbolically points to the model's metamodel. In pure JSON-LD the type signals the developer's intention that the model conforms to the type, yet it does not enforce any structural constraints. For example, in Listing 2 pink colour indicates concepts related to the `Position` type, but JSON-LD interpreters would not complain about a missing property. Such constraints remain separated, as we exemplify below.

Listing 3 depicts a more complicated model that embeds the `@context` with three entries in-place instead of importing a remote one. Line 3 defines `coord` as a namespace or shorthand for the longer IRI. The following line 4 remaps the model-local short term `PositionReference` to its absolute IRI within the previous namespace. The interpretation is as follows: any mention of the term within this model really refers to the absolute and hence unique IRI. Such remappings are necessary to disambiguate model-local terms: when comparing the data of the previous model (Listing 2, line 8) with the current model (Listing 3, line 11) we notice that both use the same term (`of`), yet should have different meaning. To this end, line 5 declares that the local term `of` is really identified by the IRI `coord:of-position`. The next line defines the term `of` to be a symbolic pointer; its `@type` is to be interpreted as an identifier (`@id`). Such a symbolic pointer *(i)* can refer to external models that may not even exist yet; *(ii)* does not impose a specific representation of the referenced entity; and *(iii)* does not enforce how to resolve that pointer. Hence, symbolic pointers facilitate loose coupling between models and enable us to fully exploit identifiers to describe graph-structured data (cf. **AP1**). From a structural perspective, JSON-LD straight-forwardly supports multi-conformance with a uniform syntax as shown in line 9. Simply assigning a list of types to a model tackles **AP2**. Only few, advanced JSON-LD features directly rely on the `@type` and their behaviour is not influenced by multi-conformance. Instead, it is the *semantics* and surrounding models that benefit from this mechanism.

For example, some types may impose structural constraints on a model that we represent here using the Shapes Constraint Language (SHACL) [22]. Listing 4 shows an example of a SHACL constraint that is serialized in JSON-LD. This model issues a (higher-order) statement about the `PositionReference` class: each concrete instance of that class must have exactly one term `of-position` that is either directly embedded (as a so-called blank node) or a symbolic pointer to another model (`sh:BlankNodeOrIRI`) which, in itself, must conform to

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

Listing 4: SHACL constraint.

```
{
  ...
  "@id": "coord:PositionReference",
  "@type": [ "Class", "NodeShape" ],
  "property": [{
    "@id": "coord:PositionReference-of-position",
    "@type": "PropertyShape",
    "path": "coord:of-position",
    "minCount": 1, "maxCount": 1,
    "nodeKind": "sh:BlankNodeOrIRI",
    "node": "geom:Position"
  }]
}
```

Listing 5: Textual model of joint `rob:joint1`. Colours that match with Figure 1 indicate identical entities.

```
{
  "@context": [ ... ],
  "@id": "rob:joint1",
  "@type": [ "Joint", "RevoluteJoint",
             "RevoluteJointWithPolarity" ],
  "between-attachments": [ "rob:link1-joint1",
                           "rob:link2-root" ],
  "common-axis": "rob:joint1-common-axis",
  "origin-offset": "rob:joint1-offset",
  "positive-direction": "rob:link2-root-z",
  "handedness": "RightHanded"
}
```

the structural constraints of the `geom:Position` class. The highlighting establishes the link to the model in Listing 3 which conforms to this constraint (orange colour) and the model in Listing 2 (pink colour). By default, SHACL does not "close" models, i.e. additional properties may co-exist with the constrained ones. SHACL defines various built-in constraints but also allows formulating custom constraints using the SPARQL Protocol and RDF Query Language [23]. For readers familiar with JSON Schema [24], which validates JSON trees, we note that SHACL can also follow graph links.

This brief overview of JSON-LD and SHACL shows that the lessons pertaining to a model's structure are not new insights. Instead, they have already found their ways into mature formats and even tools that we prefer to reuse as much as possible. However, to unfold their full potential those formats should be employed in conjunction with the correct domain models, as we demonstrate in the following.

B. Spatial relations between bodies

In this subsection we revisit the previous examples and complement the discussion of their structural properties with their domain-specific semantics. For the formalization of spatial relations, we rely on the prior work in [9].

Listing 2 shows a model of a position. Semantically, a position is a relation with three properties: `of`, `with-respect-to` and `quantity-kind`. The former two are the symbolic pointers to models of *some* points in a space, whereas the latter references *the* model of the length quantity as defined by the QUDT [25] standard. By design, coordinates are *not* part of the position relation! Instead, we compose them on top of the position relation as shown in Listing 3. The `of` property points to the previous, coordinate-free relation. Additionally, coordinates must be measured in a coordinate frame as referenced by the `as-seen-by` property and require a `unit` of measurement complementary to the quantity. Finally, in this example we represent the position's coordinate vector along three named coordinate axes: `x`, `y` and `z`. Here, the coordinate values remain fixed as they are design parameters of the kinematic chain, yet the same representation also captures time-dependent spatial relations such as the orientation over the joint. We have already discussed types that impose structural constraints on a model (highlighted in the Listing). In contrast, the two types `3D` and `Euclidean` are *semantic tags* in that they only convey meaning to other models; here, to indicate that the position is represented in 3D Euclidean space.

The model in Listing 2 represents one measurement of a position relation which could originate, for example, from a Computer-aided design (CAD) model as provided by the robot's manufacturer. However, another measurement can co-exist with that relation, for instance, as the result of a kinematics identification procedure. To any measurement an arbitrary amount of coordinate representations can be attached with potentially differing choices of coordinate frames or units of measurement. Since the coordinate-free spatial relations then are meant to represent *different* measurements they will most likely also feature different coordinates. In contrast, when multiple coordinate representations refer to the same position relation they are constrained to be consistent, i.e. they must have the same coordinate values when converted to the same frame and the same units.

Now a robotics vendor may prefer a syntactically different, yet semantically equivalent, representation of the coordinate values such as a number-indexed array of coordinate values. They would introduce a new metamodel with an associated `@context` and SHACL constraints that includes a type such as `vendor:VectorList`. They can reuse most of our metamodels but in the coordinates model would exchange the `VectorXYZ` type and replace the associated properties by a new property, for example, `"vendor:values": [0.0, 0.0, 0.42]`. Obviously, tools must be able to interpret this vendor extension.

C. Engineered motion constraints: joints & kinematic chains

Listing 5 shows the model of a joint, another relation between attachments, that is points or frames, on bodies. More specifically it is a *constraint* relation in that the joint physically constrains the relative motion of the involved bodies. Thus, each attachment must be part of a different body. The `Joint` type is the most abstract part of a joint model with the single property `between-attachments` to represent the unordered set of points or frames that the joint constrains. A `RevoluteJoint` narrows the valid types of the attachments to a frame. It also requires the `common-axis` property which specifies about which line the joint allows rotation. This property is expressed as a collinearity constraint between two lines, each being attached to one of the joint-constrained bodies. The two frames must remain at a constant distance with respect to each other as prescribed by the `origin-offset` property; the position

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

Listing 6: Forward position kinematics operator of a joint.

```
{
  "@context": [ ... ],
  "@id": "rob:fpk1",
  "@type": "ForwardPositionKinematics",
  "joint": "rob:joint1",
  "joint-space-motion": "rob:q1",
  "cartesian-space-motion":
      "rob:pose-link2-root-wrt-link1-joint1"
}
```

relation between the two frames' origins from Listing 2. A revolute joint also has a polarity that describes the positive or negative direction of motion. The polarity decomposes into two properties: the `positive-direction` to identify a directed line and the `handedness` around that line.

A joint's zero configuration can be described geometrically without reference to coordinates, for example, in the real world when two markers on both bodies line up. Physically, this can be expressed as a collinearity relation between two lines. A joint can have multiple zero configurations so that we compose them onto the joint by external models.

The structural composition relation of one or more joints is the kinematic chain. It is represented via the `KinematicChain` type with a single property `joints`, a set of all joints that belong to that chain. A valid chain must form a so-called connected graph, i.e. at least one path must exist between any bodies that are reachable via the joints. Since each joint in a kinematic chain simply refers to existing attachments, instead of containing them, the resulting topology can be an arbitrary graph as required for parallel mechanisms and not just a serial or tree structure.

D. Operators & algorithms

Kinematics and dynamics libraries like RBDL [26], SimBody [27] or MuJoCo [28] have accepted domain-specific languages such as URDF as useful interfaces and interchange formats to let users configure the *structure* of kinematic chains. Yet, up to now, no common format exists to exchange *behaviour*, that is algorithms, among those libraries. While all those libraries implement a set of common operators their composition into algorithms remains hidden behind an API.

For instance, the forward position kinematics algorithm computes the relative pose between two frames in a kinematic chain. It consists of two types of operators: the mapping of joint-space positions to Cartesian poses and the composition of those poses. Listing 6 demonstrate a concrete model of the former type, the forward position kinematics for a single joint, with its three properties the `joint` that we have already discussed in the previous subsection, the `joint-space-motion` that refers to a measurement compatible with the joint type as input to the operator and the `cartesian-space-motion` as output of the operator. A complete algorithm is a composition relation of data and one or more operator instances that must be evaluated on that data in the correct order to compute the desired output.

E. Anti-patterns revisited

For the first two anti-patterns we have already pinpointed specific technical JSON-LD features that address them. The vendor-extension for coordinate representations exemplifies how to add new metamodels and their concrete models which allows us to address the artificial structural constraints (**AP3**) that are prevalent in many modelling languages. This extension is only possible because of JSON-LD's graph model that facilitates loosely-coupled models by not enforcing a strict composition hierarchy (**AP4**). However, the *correct* graph structure originates from the best practices on how to decompose domain concepts. Here, the art lies in distinguishing an entity's intrinsic properties from externally-imposed attributes. Properties represent either primitive data types (numbers, strings, ...) or symbolic pointers. This aligns with the properties in the Web Ontology Language (OWL) [29], but differs from properties as used in the (labelled-)property graph model [30], [31] where they only represent primitive data types. To foster composability, metamodel designers should represent attributes as explicit relations.

A further result of the domain analysis are *complete* metamodels (**AP5**) that we have represented here using JSON-LD and SHACL. They also lay the foundation to alleviate implicit assumptions (**AP6**). By definition, properties describe only the *intrinsics* of an entity, that is a necessary condition for minimal metamodels. Complementing this with the previous requirement of completeness means that correct metamodels will define exactly the properties that are required to faithfully represent an entity, nothing more and nothing less. Hence, no implicit assumptions can exist in conforming models. The knowledge for designing such metamodels can only come from domain experts.

F. Case study: code generation

As an initial validation we have implemented a code generator that consumes composable models and generates a KDL-based solver implementation. We rely on Python and reuse established libraries including RDFLib [32], pySHACL [33], PyLD [34] and the Jinja [35] template engine.

As an interchange format the model of an algorithm as discussed above is agnostic about a programming language and software library. Hence, we gradually lower it to code. As a first step after loading the models with RDFLib, we employ pySHACL to validate that they conform not only to the constraints above, but also to the additional, library-specific constraints. For example, a pure kinematics library would not be able to handle algorithms that require dynamics operators. Next, we rewrite the graph to the representation required by the library. This includes, for instance, selecting the desired coordinate representation, converting units of measurement or annotating the joint type as many libraries include optimized routines when a joint axis aligns with a coordinate axis. To this end, we rely on RDFLib's SPARQL Update [36] interface or native Python implementations. The result is an abstract syntax graph that we serialize into an abstract syntax tree (AST) using JSON-LD's framing algorithm [37] as implemented in pyLD. As a Query by Example [38] language, JSON-LD framing allows us to provide a prototypical layout of the desired tree that will be filled with concrete data from the graph during processing.

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

The final step uses Jinja to render the AST into the concrete C++ syntax with the appropriate variables and function calls.

III. Discussion

In this paper we have only presented an excerpt of the full models to exemplify our approach and the resulting models. More complete models and tools in the accompanying repositories demonstrate how our approach scales for more complete applications. Even if the models are still in an early development stage, the examples above suffice to explain how composable models allow us to address the anti-patterns that commonly exist in DSLs such as URDF.

a) Added value: The composable models for kinematic chains extend on existing languages such as URDF along two dimensions. First, they are complete with respect to physics and mechanics and hence more detailed. Second, they go beyond such languages by enabling simple compositions on top of existing models, as we have shown with the representation of solver algorithms. More examples that we have omitted in this paper due to space limitations include *(i)* at a lower level, the geometry domain with the systematic composition of entities like points, frames or (rigid) bodies; and *(ii)* at a higher level, mechanical dynamics with concepts like inertia or forces, but also motion specifications.

b) Compatibility with existing DSLs: Many models exist in DSLs like URDF. Therefore, the question arises how those can live alongside or be composed with our models? A very pragmatic answer is that those DSLs potentially suffice in some scenarios. An intermediate way is to annotate existing models where they offer identifiers to reach into the composition hierarchy (similar to SRDF). Finally, we are working on automated model-to-model transformations that allow us to transition between those models where the metamodels support the same features.

c) Model structure: From a structural perspective, adding identifiers to every entity is the strongest principle: even if the other anti-patterns were still present in a model, one could still annotate a model with an identifier. Those annotations then resemble provenance. As a consequence the resulting composite models appear "inverted" compared to DSLs. The latter start at a top-level element (e.g. the `robot` in URDF) from where the composition hierarchy fans out towards the primitive entities. In contrast, the former starts at the primitives and gradually composes them in (higher-order) relations without a single top-level element. Additionally, multiple relations can independently co-exist with each other.

d) Development process: Loose coupling of metamodels avoids the common problem in DSLs that all sub-metamodels must be developed and updated in lockstep and then released as *the* common metamodel. With composable models all metamodels can instead be developed concurrently and released independently of each other. Additionally, vendors can easily provide custom metamodels with specific extensions: from a technical standpoint there exists no difference between such extensions and standardized metamodels. Furthermore, we don't have to strive for completeness with respect to all possible model representations, for example, to support the wide variety of orientation representions. Hence, composable models play along nicely with iterative and incremental development processes where new representations can be added in later cycles without interfering with already established models.

e) Size of composable models: Already the examples in this paper show that the composable models are more extensive (more lines in the textual representation) than their DSL counterparts like URDF: the models are *complicated*, but not complex [39]. In other words the models are *complete* and *minimal*, i.e. they include everything that is required to be physically correct but nothing more, and they are composed via the simplest rules or composition structures. Hence, any smaller models are necessarily incomplete, and the complicatedness is hidden away behind implicit choices or assumptions. This can lead to difficult-to-trace software bugs, or worse, robots communicating with each other in incompatible dialects without being aware of the differences.

f) Existing standards: One objective was to maximally reuse existing standards with their conforming tools and implementations. Here, an important lesson was that most of the structural anti-patterns have already been addressed. In view of the more complicated, graph-structured models it is advantageous that many of those standards specify *declarative* languages that enable to offload more effort on the tooling. Graph traversals, graph matching and graph rewriting are more difficult to implement than their tree-based counterparts. Yet, languages like SPARQL alleviate us from implementing those algorithms in imperative languages.

IV. Conclusions and future work

Our analysis of robot description DSLs has revealed several shortcomings, in particular tightly-coupled *structure* of unrelated domains and implicit *semantics* with assumptions hidden in human-targeted documentation. Those shortcomings hinder composability and computer-understandability of those languages. We tackle those anti-patterns by the design of composable models and complete metamodels that unambiguously define model semantics. Our composable models rely on established standards and enable use to extend structural kinematic chain models with behavioural models of algorithms. We systematically transform those algorithms to correct-by-construction code.

Our future work includes extending the (meta)models to the mechanics domain, extending the code generation to further solver libraries and investigating how the models can support robots at runtime.

Acknowledgement

Sven Schneider and Nico Hochgeschwender gratefully acknowledge the ongoing support of the Bonn-Aachen International Center for Information Technology. This work was supported by the European Union's Horizon 2020 project *SESAME* (H2020-101017258, *Secure and Safe Multi-Robot Systems*). HB gratefully acknowledges the support of *RobMoSys* (H2020-732410, *Composable Models and Software for Robotics Systems-of-Systems*), *Esrocos* (H2020-730080, *European Space Robot Control Operating System*).

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] I. Sucan and J. Kay, "Semantic Robot Description Format (SRDF)," 2009, last accessed: 11. Sep. 2022. [Online]. Available: http://wiki.ros.org/srdf

[2] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[3] D. Hershberger, D. Gossow, J. Faust, and W. Woodall, "rviz - ROS Wiki," last accessed: 11. Sep. 2022. [Online]. Available: http://wiki.ros.org/rviz

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

[5] R. Smits, "KDL: Kinematics and Dynamics Library," last accessed: 11. Sep. 2022. [Online]. Available: http://www.orocos.org/kdl

[6] S. Chitta, K. Hsiao, G. Jones, I. Sucan, and J. Hsu, "Unified Robot Description Format (URDF)," 2011, last accessed: 11. Sep. 2022. [Online]. Available: http://wiki.ros.org/urdf

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan, *Extensible Markup Language (XML) 1.1 (Second Edition)*, World Wide Web Consortium (W3C) Std., 2006, https://www.w3.org/TR/2006/REC-xml-names11-20060816/.

[8] J. Hsu, "urdfdom/urdf.xsd," 2012, last accessed: 11. Sep. 2022. [Online]. Available: https://github.com/ros/urdfdom/blob/6da0620ab9d0c7f970ccc701b9befee349ac311d/xsd/urdf.xsd

[9] T. De Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies: Semantics for standardization," *IEEE Robotics & Automation Magazine*, vol. 20, no. 1, pp. 84–93, 2012.

[10] *COLLADA – Digital Asset Schema Release 1.5.0*, The Khronos Group Inc., Sony Computer Entertainment Inc. Std., 2008.

[11] *SDFormat Version 1.9*, Open Source Robotics Foundation Std., http://sdformat.org/.

[12] E. Cousineau, "Specifying pose: Proposal for a better pose," last accessed: 26. Feb. 2023. [Online]. Available: http://sdformat.org/tutorials?tut=better_pose_proposal

[13] A. Taddese, "Custom elements and attributes," last accessed: 26. Feb. 2023. [Online]. Available: http://sdformat.org/tutorials?tut=custom_elements_attributes_proposal

[14] S. Peters, A. Taddese, and E. Cousineau, "Pose frame semantics proposal," last accessed: 26. Feb. 2023. [Online]. Available: http://sdformat.org/tutorials?tut=pose_frame_semantics_proposal

[15] "Universal Scene Description," Pixar Animation Studios, last accessed: 25. Feb. 2023. [Online]. Available: https://graphics.pixar.com/usd/release/index.html

[16] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutiérrez, S. Kirrane, J. E. Labra Gayo, R. Navigli, S. Neumaier, A.-C. Ngonga Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, and A. Zimmermann, *Knowledge Graphs*. Morgan & Claypool, 2021.

[17] S. DeRose, E. Maler, D. Orchard, and N. Walsh, *XML Linking Language (XLink) Version 1.1*, World Wide Web Consortium (W3C) Std., 2010, https://www.w3.org/TR/xlink11/.

[18] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.-A. Champin, and N. Lindström, *JSON-LD 1.1. A JSON-based Serialization for Linked Data*, World Wide Web Consortium (W3C) Std., 2020, https://www.w3.org/TR/json-ld/.

[19] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, Internet Engineering Task Force (IETF) Std., 2017, https://datatracker.ietf.org/doc/html/rfc8259.

[20] T. Berners-Lee, "Linked data," Design Issues. World Wide Web Consortium (W3C), July 2006. [Online]. Available: https://www.w3.org/DesignIssues/LinkedData.html

[21] ——, "Semantic web road map," Design Issues. World Wide Web Consortium (W3C), November 1998. [Online]. Available: http://www.w3.org/DesignIssues/Semantic.html

[22] H. Knublauch and D. Kontokostas, *Shapes Constraint Language (SHACL)*, World Wide Web Consortium (W3C) Std., 2017, https://www.w3.org/TR/shacl/.

[23] S. Harris and A. Seaborne, *SPARQL 1.1 Query Language*, World Wide Web Consortium (W3C) Std., 2013, https://www.w3.org/TR/sparql11-query/.

[24] A. Wright, H. Andrews, B. Hutton, and G. Dennis, *JSON Schema: A Media Type for Describing JSON Documents*, Internet Engineering Task Force Std., 2020, https://json-schema.org/draft/2020-12/json-schema-core.html.

[25] *Quantities, Units, Dimensions and Types (QUDT)*, QUDT.org Std., 2011, https://qudt.org/.

[26] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, pp. 1–17, 2016.

[27] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia IUTAM – IUTAM Symposium on Human Body Dynamics*, vol. 2, pp. 241–261, 2011.

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

[29] C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, and M. Smith, *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)*, World Wide Web Consortium (W3C) Std., 2012, https://www.w3.org/TR/owl2-syntax/.

[30] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys*, vol. 50, no. 5, 2018.

[31] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," 2019. [Online]. Available: https://arxiv.org/abs/1910.09017

[32] D. Krech, "RDFLib," 2002, last accessed: 13. Sep. 2022. [Online]. Available: https://github.com/RDFLib/rdflib

[33] A. Sommer, "pySHACL," 2018, last accessed: 13. Sep. 2022. [Online]. Available: https://github.com/RDFLib/pySHACL

[34] D. Longley, "PyLD," 2011, last accessed: 13. Sep. 2022. [Online]. Available: https://github.com/digitalbazaar/pyld

[35] A. Ronacher, "Jinja — The Pallets Projects," 2007, last accessed: 13. Sep. 2022. [Online]. Available: https://palletsprojects.com/p/jinja/

[36] P. Gearon, A. Passant, and A. Polleres, *SPARQL 1.1 Update*, World Wide Web Consortium (W3C) Std., 2013, https://www.w3.org/TR/sparql11-update/.

[37] D. Longley, M. Sporny, G. Kellogg, M. Lanthaler, and N. Lindström, *JSON-LD 1.1 Framing. An Extension to the Application Programming Interface for the JSON-LD Syntax*, World Wide Web Consortium (W3C) Std., 2020, https://www.w3.org/TR/json-ld-framing/.

[38] M. M. Zloof, "Query by example," *National Computer Conference*, vol. 44, 1975.

[39] D. Snowden, "Liberating knowledge," in *Liberating Knowledge. CBI Business Guide*. Caspian Publishing, 1999.

Authorized licensed use limited to: KU Leuven Libraries. Downloaded on October 23,2023 at 13:06:11 UTC from IEEE Xplore. Restrictions apply.