SESAME
SECURE AND SAFE MULTI-ROBOT SYSTEMS

Project Number 101017258

D3.4 Executable Scenario Workbench (Final Version)

Version 1.0
30 June 2023
Final

Public Distribution

Bonn-Rhein-Sieg University

Project Partners: **Aero41, ATB, AVL, Bonn-Rhein-Sieg University, Cyprus Civil Defence, Domaine Kox, FORTH, Fraunhofer IESE, KIOS, KUKA Assembly & Test, Locomotec, Luxsense, The Open Group, Technology Transfer Systems, University of Hull, University of Luxembourg, University of York**

Project Partner Contact Information

Aero41 Frédéric Hemmeler Chemin de Mornex 3 1003 Lausanne Switzerland E-mail: frederic.hemmeler@aero41.ch	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany E-mail: scholze@atb-bremen.de
AVL Martin Weinzerl Hans-List-Platz 1 8020 Graz Austria E-mail: martin.weinzerl@avl.com	**Bonn-Rhein-Sieg University** Nico Hochgeschwender Grantham-Allee 20 53757 Sankt Augustin Germany E-mail: nico.hochgeschwender@h-brs.de
Cyprus Civil Defence Eftychia Stokkou Cyprus Ministry of Interior 1453 Lefkosia Cyprus E-mail: estokkou@cd.moi.gov.cy	**Domaine Kox** Corinne Kox 6 Rue des Prés 5561 Remich Luxembourg E-mail: corinne@domainekox.lu
FORTH Sotiris Ioannidis N Plastira Str 100 70013 Heraklion Greece E-mail: sotiris@ics.forth.gr	**Fraunhofer IESE** Daniel Schneider Fraunhofer-Platz 1 67663 Kaiserslautern Germany E-mail: daniel.schneider@iese.fraunhofer.de
KIOS Maria Michael 1 Panepistimiou Avenue 2109 Aglatzia, Nicosia Cyprus E-mail: mmichael@ucy.ac.cy	**KUKA Assembly & Test** Michael Laackmann Uhthoffstrasse 1 28757 Bremen Germany E-mail: michael.laackmann@kuka.com
Locomotec Sebastian Blumenthal Bergiusstrasse 15 86199 Augsburg Germany E-mail: blumenthal@locomotec.com	**Luxsense** Gilles Rock 85-87 Parc d'Activités 8303 Luxembourg Luxembourg E-mail: gilles.rock@luxsense.lu
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium E-mail: s.hansen@opengroup.org	**Technology Transfer Systems** Paolo Pedrazzoli Via Francesco d'Ovidio, 3 20131 Milano Italy E-mail: pedrazzoli@ttsnetwork.com
University of Hull Yiannis Papadopoulos Cottingham Road Hull HU6 7TQ United Kingdom E-mail: y.i.papadopoulos@hull.ac.uk	**University of Luxembourg** Miguel Olivares Mendez 2 Avenue de l'Universite 4365 Esch-sur-Alzette Luxembourg E-mail: miguel.olivaresmendez@uni.lu
University of York Simos Gerasimou & Nicholas Matragkas Deramore Lane York YO10 5GH United Kingdom E-mail: simos.gerasimou@york.ac.uk nicholas.matragkas@york.ac.uk	

Document Control

Version	Status	Date
0.1	Version ready for internal review	26 June 2023
0.2	Address feedback by HULL	29 June 2023
0.3	Address feedback by ATB	29 June 2023
1.0	Final Version	30 June 2023

Table of Contents

List of Figures

Executive Summary

In this deliverable, we describe both the underlying concepts and the technical realization of our final version of the Executable Scenario Workbench (Deliverable D3.4). In Deliverable D3.1, we formulated a methodology of scenario-based development for multi-robot applications and described the design dimensions involved in constructing these scenarios. Here, we report on three concrete tools conforming to our proposed methodology and each addressing a particular design dimension, namely, environment, robot specification, and acceptance criteria.

In the following introductory section, we will give an overview of our methodology and the three developed tools, as well as how they conform to the aforementioned methodology. We will also discuss in this section the relevance of these tools to the requirements listed in Task 3.2 of the SESAME proposal. Afterwards, we include descriptions of the developed tools in the form of tutorials that are automatically generated from their respective documentation.

1 Introduction

The Executable Scenario (ExSce) Workbench is the collection of tools to handle ExSce models which conform to the ExSce metamodels. This Deliverable presents the final version of the ExSce Workbench and hence is an evolution – i.e. it contains the results – of the initial version from D3.2. The deliverable contains direct results of the associated Task T3.3 and indirect contributions from Task T3.2 (the ExSce metamodels). Additionally, the scenario management tools and metamodels are part of the ExSce Workbench, but are detailed in their own Deliverable D3.3. The majority of this document represents an automatically-generated snapshot of the living documentation in the model and tooling repositories that will be linked in the respective sections.

To recapitulate from our previous deliverable D3.1, the ExSce's ingredients are as follows:

Executable Scenario Concept Executable Scenarios (ExSce) are model-based narrative descriptions of robotic missions guiding the engineering of MRS applications. An ExSce supports the definition of scenarios, composed of mission-relevant and mission-plausible information. On the one hand, by mission-relevant information we refer to, among others, the environment and its dynamics, time and events, objects (e.g., inspected building) and subjects (e.g., human operators) and their potential behaviour. On the other hand, the mission-plausible information describes acceptance criteria that enable the verification and validation of MRS requirements. Both, mission-plausible and mission-relevant information are computer-interpretable models and hence enable the transformation into artefacts which can be executed in different contexts such as in simulators or on real robots.

Executable Scenario Methodology To put the ExSce concept into action, the ExSce methodology enables the structured development of MRS from stakeholders' scenarios. Hence, it comprises both a tailorable process model and associated tools that guide and support the stakeholders in (i) specifying scenarios; (ii) transforming them to executable artefacts; (iii) executing those artefacts; (iv) assuring the quality of the MRS in those scenarios; and (v) generalizing those scenarios.

Executable Scenario Workbench The ExSce Workbench is a collection of tools that supports stakeholders in carrying out one or more activities of the ExSce Methodology.

D3.1 has also outlined and analysed various design dimensions that underlie the ExSce and are required for constructing such scenarios. For this document, the functional design dimensions of *robot* and *environment* as well as the non-functional design dimension of *quality assurance*, together with the *methodology* and *modelling* dimension will be relevant. The former, functional dimensions, while self-explaining, are extremely complex. Introducing the sometimes subjective interpretation of when an MRS has successfully executed a customer's task adds even more complexity ontop. The methodology refers to the different phases in the ExSce workflow, of which we will focus mostly on the specification and transformation phases. As for the latter dimension of *modelling* — that is the whole model-driven engineering perspective — the levels of realization (the "real" entities), models (some abstraction of a real entity) and metamodels (the concepts available for building models) are discussed here.

We build upon the results of Deliverable D3.2 where we had described two concrete (meta)models and associated tools that conform to the ExSce methodology and address particular design dimensions. This included an environment specification as part of the scenario building (the FloorPlan DSL), and a kinematic chain specification (`comp-rob2b`) that is required for any MRS to physically interact with their environments. Following the methodology outlined in our first deliverable, these metamodels are decoupled from specific systems and missions, which allows their reuse in specifying MRS missions with similar features.

FloorPlan DSL[1] is a domain-specific language (DSL) for specifying floor plans of indoor environments as required by several of SESAME's use cases. The DSL conforms to the geometry metamodel because it is required to describe spatial relations and geometric shapes of objects like rooms or doors. Additionally, this DSL

[1] https://github.com/sesame-project/FloorPlan-DSL

allows a user to describe variability of the environment, e.g. the thickness of walls or the width of doors. Such a variability specification is provided via probability attachments which forms another cross-cutting metamodel on its own. We have substantiated the design of this language with developer interviews with domain experts in robotic navigation. From concrete FloorPlan DSL models we generate 3D boundary representations in formats that are understood by a wide range of simulators. Those results have already been used by YORK and LOMTC for simulating environments. The DSL and its grammar is only targeted at human users. Thus, we additionally generate composable models from such models that are instead meant as an interchange format between various tools.

`comp-rob2b`[2][3] is a methodology for creating *composable models*. In software engineering it has been known for several decades that composition should be preferred over inheritance. Still, inheritance remains *the* means to introduce explicit extension points in most model-based approaches. Here, we specifically demonstrate our methodology and its benefits in specifications of robotic kinematic chains. Just like environment specifications, kinematic chains conform to the geometry metamodels, but compose it with motion constraints like joints or kinematic chains in addition to dynamics that introduce concepts like inertia. While we introduce those concepts for describing kinematic chains, they are by no means restricted to that application and must remain reusable in other applications. Another core contribution of `comp-rob2b` is the *reification* of behaviour in terms of algorithms. Such a first-class representation of computations enables, for example, their manipulation also at runtime or code generation towards various backend libraries. Finally, we have implemented tools that work with the previous composable models for visualization and code generation.

1.1 Updates for the final version of the ExSce Workbench

We have extended both of the above metamodels and tools to be part of the final ExSce Workbench. The FloorPlan DSL features a new tool to support model-to-model transformation from FloorPlan models into the JSON-LD[4] format for better conformance to the composable models methodology. Additionally, we have developed a new companion tool for the FloorPlan DSL which consumes these JSON-LD models and enables the modelling and placement of stateful, dynamic objects in existing indoor environments that the robots can interact with (see [5] for more details).

With `kindyngen`[6] we have implemented a new tool to synthesize and generate correct-by-construction kinematics and dynamics solvers for kinematic chains. Our composable design enables vendors to hook their custom computations, for example controllers or estimators, into the conventionally atomic computations that make up such solvers. Those "extensions" have access to and can reuse the solver's overall state and can consequently avoid duplicating computations. Structurally, the tool consists of two parts. First, a synthesizer that transforms a kinematic chain model and a solver specification into an algorithm model. Second, a code generator that transform the algorithm model into C code.

Additionally, we have realized `bdd-dsl`[7], a tool and metamodels to capture acceptance criteria for MRS using Behaviour-Driven Development (BDD) [1]. BDD complements the ExSce by introducing the scenario view on the robotic system using a Given-When-Then cascade. The "Given" clause captures the pre-conditions including the robotic system and its state, the "When" clause describes an event that the system should react to, while the "Then" clause specifies the desired or expected behaviour. However, applying traditional BDD approaches to MRS applications can be challenging due to the complex interplay of multiple variability dimensions and domains involved in describing robotic scenarios. To address these challenges more effectively, `bdd-dsl` introduces a metamodel for defining composable BDD scenario templates that can be customized to

[2] https://github.com/comp-rob2b/modelling-tutorial
[3] https://github.com/comp-rob2b/modelling-tools
[4] https://json-ld.org/
[5] https://github.com/hbrs-sesame/floorplan-object-modelling-and-placement
[6] https://github.com/hbrs-sesame/kindyngen
[7] https://hbrs-sesame.github.io/bdd-dsl/

specific robotic use cases. The metamodel design takes inspiration from a similar approach used for managing data structures [2] and is further supported by an analysis of competition rulebooks to identify the variability dimensions that may arise when evaluating robotic scenarios.

While the tools and metamodels of the scenario management are also part of the ExSce Workbench we refrain from including them into this Deliverable and instead refer the interested reader to the dedicated Deliverable D3.3.

The ExSce Workbench has contributed to and supported several scientific publications [3]–[6].

1.2 Relations to Other Deliverables

The metamodels, models, and tools developed for the ExSce Workbench targets specific uses cases and stakeholders activities in the ExSce Methodology. As such, they serve as concrete suggestions and examples of how to *represent* and *compose* models for different aspects of a Multi-Robot Systems (MRS) scenario. The realization of concrete models, however, requires contribution from developers to support their specific use cases and scenarios. In the Multi-Robot Collaborative Rescue Mission (D2.5), for example, specification of the robot models, kinematic and dynamics constraints, and missions can be created from our current metamodels. Specifying temporal dependencies between tasks and different robot behaviours requires development additional metamodels, the process of which can benefit from applying our ExSce Methodology. In another example, acceptance criteria produced as the result of safety (D4.6) & security (D5.6) analysis performed in the respective deliverables can be modelled using `bdd-dsl` as BDD scenarios.

1.3 Status of SESAME tasks and requirements

To conclude the introduction, we review the work progress along the SESAME tasks and requirements. We also show how the developed (meta)models and tools directly address several requirements listed in Task 3.2 of the SESAME project proposal.

We had already completed the following tasks for the initial version of the ExSce Workbench (cf. D3.2):

- FloorPlan DSL enables us to successfully model environments. Such specifications also capture variability in such environments (T3.2).
- From the above environment models, we can generate 3D shape representations that are supported by a wide range of simulators (T3.2).
- We have introduced a methodology for creating composable models which facilitates the separation of concerns and, hence, allows us to reuse models and metamodels.
- `comp-rob2b` has demonstrated that methodology for creating robot models of kinematic chains and their algorithms. The concepts are designed in such a way that they are reusable in domains beyond kinematic chains (T3.2).

With the final version we have completed the remaining, previously open and work-in-progress tasks:

- We carried out the evaluation of the models and tools of the FloorPlan DSL and published the results in [3].
- FloorPlan-DSL is extended to allow modelling dynamic elements and composing with existing indoor environments, as demonstrated in the door modelling tutorial[8].
- `kindyngen`[9] extends `comp-rob2b` with tools to synthesize computation models from solver specifications for rigid-body dynamics, as well as to generate executable implementations from the synthesized models.

[8] https://github.com/hbrs-sesame/floorplan-object-modelling-and-placement
[9] https://github.com/hbrs-sesame/kindyngen

- `bdd-dsl` includes metamodels and tooling to specify acceptance criteria (T3.3) as BDD scenarios, as well as, tools to transform such specifications into the Gherkin format for integrating with existing testing frameworks. Initial versions of instrumentation for verifying simulated executions is also available and will be further improved in the future.
- ExSce workflow and process management (T3.4) and provenance data acquisition for scenario generalization (T3.5) is reported in the ExSce Management deliverable (D3.3).
- Our approach to introduce collaborative intelligence (T2.5) to MRS for utilizing past "experiences" to support latter executions is reported in D2.6.

The following sections will describe the (meta)models and tools in further detail, explain the underlying modelling concepts and provide tutorials of how to use them.

2 Composable robot models - A modelling tutorial for kinematic chains and their behaviour

2.1 Introduction

The objective of this tutorial is to demonstrate how to create composable models for robots. Here, we focus on a simple kinematic chain with one degree of freedom (DoF) because (i) kinematic chains are the smallest viable world models that robots employ and hence go beyond toy examples; (ii) they provide many building blocks that are also relevant for other domains; (iii) they demonstrate the challenges in modelling and how to address them with composable models; and (iv) kinematic chains with more than one DoF are structurally very similar to the 1-DoF chain.

The following figure (Figure 1) depicts the kinematic chain example that we will study below. It consists of the two links `link1` and `link2` as well as the revolute joint `joint1`.

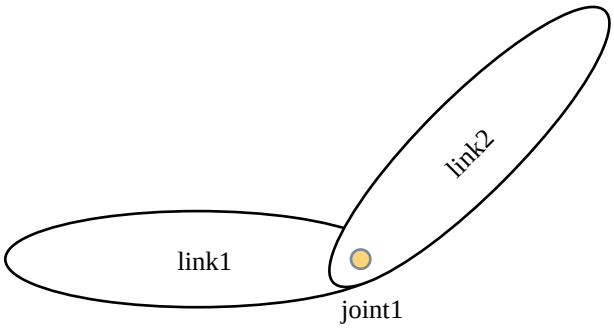

Figure 1: Kinematic chain example

2.1.1 JSON-LD

Here we choose to represent our models and metamodels with JSON-LD[10]. JSON-LD is a W3C standard to encode linked data in JSON and hence, it bridges between the widely-used JSON format and the Semantic Web[11] with all its established tools.

JSON-LD introduces several keywords that recur in the models so that we briefly explain them here. First, each model can be identified by an identifier, an Internationalized Resource Identifier (IRI), with the `@id` property. Second, one or more types (also IRIs) can be associated with a model by using the `@type` property. Third, `@context` indicates a context which describes the mapping between the JSON world and the Semantic Web world. In other words, the context defines the metamodels that a model conforms to. Common contexts or elements can be hoisted out of a model to a composition model (e.g. the "document root"). Finally, a graph can be reified using the `@graph` keyword.

2.2 Skeleton

The first step is to create the "skeleton[12]" or "stick figure" for the bodies in the kinematic chain. By skeleton, we mean the bare minimum structure to which we attach further models. Here, the skeleton consists of points,

[10] https://www.w3.org/TR/json-ld/
[11] https://www.w3.org/standards/semanticweb/
[12] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/skeleton.json

vectors and frames as shown in the following diagram (Figure 2). The arrows indicate that the origins of both frames coincide, but for better readability both frames are spatially separated in the figure.

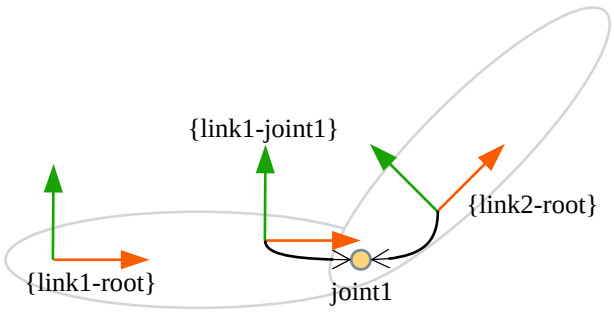

Figure 2: Skeleton of the kinematic chain example

Note that the links' spatial extent or shape is not part of the skeleton. Instead, it is one of the possible attachments to entities in the skeleton. All attachments, some of them discussed below, are independent of each other and only refer to the common elements defined in the "skeleton".

2.2.1 Point

The point is the most primitive entity in any space. In the context of robotics the purposes of points are multifold. They are the building block of further geometric entities e.g. line segments, vectors or frames as we will see below. Points also play various roles in encoding spatial relations or dynamic quantities. For instance, they are required to define positions, or they represent the point of observation/measurement for twists and wrenches.

The following example shows a point — that is meant to represent the origin of the "link1-joint1" frame — that lives in 3D Euclidean space as indicated by its type. As (Euclidean) space consists of an infinite amount of points, the interpretation of this model is that it designates one particular instance.

```
{
    "@id": "rob:link1-joint1-origin",
    "@type": [ "3D", "Euclidean", "Point" ]
}
```

The types `3D`, `Euclidean` and `Point` are examples of what we call semantic constraints in that they implicitly encode constraints (that can however be looked up in the definition of the type). This contrasts with structural constraints which define a specific representation of a model and allow checking for well-formedness in terms of the presence or absence of certain properties in that model.

Additionally, the three types in the above model demonstrate multi-conformance. That means that a model can conform to more than one metamodel or type. This is a feature that lacks in many established model-driven engineering approaches. Also note that in the step from JSON to JSON-LD all strings in the type array get converted to IRIs so that one can choose to look up their meaning.

2.2.2 Vector

Euclidean space distinguishes two types of vectors:

- Free vectors that possess only a direction and a magnitude.

- Bound vectors that are composition relations between two Euclidean points (the start and end point). Bound vectors are also vectors and hence possess a direction and magnitude.

The following model exemplifies a bound vector that will represent the x-direction of the "link1-joint1" frame. Hence, we also impose the constraint that it should be of unit length (this constraint is valid only in metric spaces such as Euclidean space). Also note that we omit the end point because here it is not relevant to the application.

```
{
    "@id": "rob:link1-joint1-x",
    "@type": [ "3D", "Euclidean", "Vector",
               "BoundVector", "UnitLength" ],
    "start": "rob:link1-joint1-origin"
}
```

Similar to the previous example, neither the `Vector` type nor the `UnitLength` type impose a structural constraint. We notice some mismatch in the terminology: one would usually not refer to "unit length" as a "type". It is actually a constraint that this model must conform to. Especially in the setting of defining a frame and later a standard basis of a vector space (see below), "unit length" must at some point be axiomatically grounded.

In contrast to the previous two types, the `BoundVector` does have an impact on the structure of the model in that it does require the `start` property to be present. The metamodel defines the `start` property to be a symbolic pointer (an IRI) and the model shown here lets it refer to the point defined in the previous example.

2.2.3 Frame

A frame (of reference) in Euclidean space is an attachment point for orientation or pose specifications. In n-dimensional space it can be represented by $n+1$ non-coplanar points (a simplex) or by n vectors that all share a common start point. All points are constrained to remain at a fixed distance which means that a frame is the most simple rigid body. Moreover, we will work with orthonormal frames which means that all vectors are of unit length and form right angles with respect to each other. In three-dimensional space a frame also indicates orientation or handedness, i.e. a frame can either be left-handed or right-handed.

The model below exemplifies the frame "link1-joint1" which represents the attachment point of "joint1" on the first link. Additionally, in this model we chose to structurally represent the frame with the `OriginVectorsXYZ` type which requires the `origin` property that points to the common start point of the vectors and the three properties that represent the vectors themselves with the names `vector-x`, `vector-y` and `vector-z` as defined by the type.

```
{
    "@id": "rob:link1-joint1",
    "@type": [ "3D", "Euclidean", "Frame", "RigidBody",
               "Orthonormal", "RightHanded",
               "OriginVectorsXYZ" ],
    "origin": "rob:link1-joint1-origin",
    "vector-x": "rob:link1-joint1-x",
    "vector-y": "rob:link1-joint1-y",
    "vector-z": "rob:link1-joint1-z"
}
```

We notice that the `OriginVectorsXYZ` forces the names of the vectors to be x, y and z. But this is just a choice, albeit a common one. Other conventions to denominate the vectors are x_0, x_1 and x_2 or just represent

them as an ordered list. To support such further representations a different type (potentially originating from another metamodel) would be required.

Another objective of frames is to map geometric entities such as points or vectors to their coordinate representation i.e. they define coordinate systems as we will discuss below.

2.2.4 Simplicial complex

With the above geometric entities instantiated, we can now model the first part of a link: it is simply a collection of all those entities that are part of the link. In accordance with mathematics we call this composition structure a simplicial complex as illustrated in the following diagram (Figure 3).

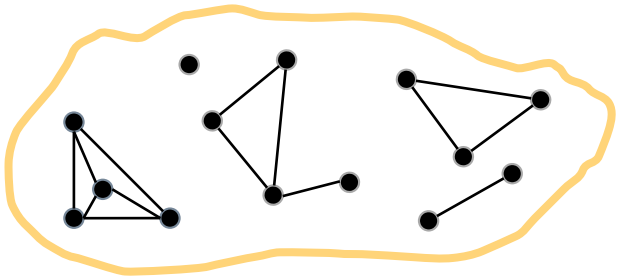

Figure 3: Graphical illustration of a simplicial complex as a collection of simplices

In three dimensions it can consist of points, line segments, triangles ("2D frames") and tetrahedra ("3D frames") which are the 0-, 1-, 2- and 3-dimensional simplices, respectively. Since the simplicial complex is a transitive relation all constituents of the simplices will be part of the simplicial complex, too. The following model shows the simplicial complex associated with "link1".

```
{
    "@id": "rob:link1",
    "@type": [ "SimplicialComplex", "RigidBody" ],
    "simplices": [
        "rob:link1-root",
        "rob:link1-joint1",
        "rob:link1-joint1-origin",
        "rob:link1-joint1-x",
        "rob:link1-joint1-y",
        "rob:link1-joint1-z"
    ]
}
```

This example also imposes a rigid-body constraint on the simplicial complex to model that no relative motion occurs between any of the simplices, i.e. the material exhibits infinite stiffness. While not yet implemented here, our modelling approach is also open to support soft bodies where the relative motion between some or all simplices is described by constitutive equations that characterize mechanical material properties. The rigid-body is indeed one of the most simple constitutive equations.

The simplicial complex with a rigid-body constraint is commonly used to simplify numeric computations that solve the equations of motion for complex mechanisms because it enables a simplified representation of inertia. Such a lumped parameter representation will be attached to a link in the discussion of dynamics below.

2.3 Spatial relations

The first type of "declarative" spatial relations defined in the spatial relations model[13] represents the collinearity of two lines or rather vectors. Namely, the z-vectors of the "link1-joint" and "link2-root" frames. The reason for this constraint relation is that a revolute joint requires a common axis that is fixed to both rigid bodies and around which the rotation occurs. The following model exemplifies this constraint for "joint1".

```
{
    "@id": "rob:joint1-common-axis",
    "@type": "LineCollinearity",
    "lines": [
        "rob:link1-joint1-z",
        "rob:link2-root-z"
    ]
}
```

The second type of "imperative" spatial relations comprises the position, in its linear, angular and composite version as well as its first- and second-order time derivatives of velocity and acceleration, respectively. The following diagram (Figure 4) depicts the three pose relations (i.e. the composition of positions and orientations represented as an arrow with a solid head) that are of interest in the kinematic chain example.

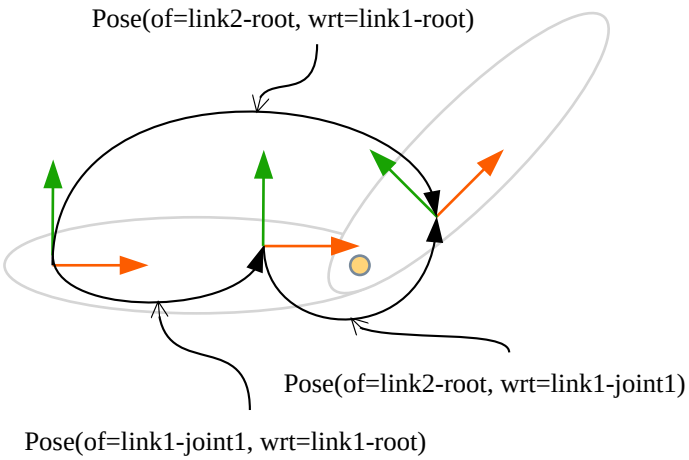

Figure 4: Pose relations attached to the skeleton

The lower-left pose crosses the link, the lower-right pose crosses the joint and the upper pose is the composition of the two previous poses. The following model exemplifies the textual representation of the former pose. A pose relates two frames which we call the `of` frame and the `with-respect-to` frame. An additional property of every position-based spatial relation is the quantity kind as used in dimensional analysis. Here, we reuse the QUDT[14] ontology to represent quantity kinds. For any pose there will always be two quantity kinds, namely the `Angle` to represent the orientation part of the pose and the `Length` to represent the position part.

```
{
    "@id": "rob:pose-link1-joint1-wrt-link1-root",
    "@type": "Pose",
    "of": "rob:link1-joint1",
```

[13] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/spatial-relations.json
[14] https://www.qudt.org/

```
    "with-respect-to": "rob:link1-root",
    "quantity-kind": [ "Angle", "Length" ]
}
```

A pose is sufficient to describe the relative position of any two points that belong to two different rigid bodies. Hence, even if the pose only relates two frames it follows that it also characterizes the pose between the rigid bodies (or simplicial complices) that those frames are a part of.

By separating the pose from the frames which it relates, we enable a modeller to associate multiple poses with a frame. This is a more faithful representation of poses than treating them as properties of frames as is often found in state-of-the-art modelling approaches. However, it also allows creating cycles of poses. Whenever such a situation occurs, the poses accumulated along each path in the cycle must be consistent. While we have chosen the pose as an example, all arguments in this paragraph hold for any spatial relation.

2.3.1 Coordinates

To perform computations a numeric representation of the spatial relations is required as shown in the model[15] for Cartesian coordinates.

The following example shows the coordinates for the pose across the "link1". Since that pose remains static, its coordinates can be provided during the robot's design, for example, by a robot manufacturer. For poses across motion constraints such as joints the concrete coordinate values will most likely only be available during the robot's runtime and will be acquired by some sensor.

```
{
    "@id": "rob:pose-link1-joint1-wrt-link1-root-coord",
    "@type": [ "3D", "Euclidean", "PoseReference",
               "PoseCoordinate", "DirectionCosineXYZ",
               "VectorXYZ" ],
    "of-pose": "rob:pose-link1-joint1-wrt-link1-root",
    "as-seen-by": "rob:link1-root",
    "unit": [ "UNITLESS", "M" ],
    "direction-cosine-x": [ 0.0, 0.0, 1.0 ],
    "direction-cosine-y": [ 0.0, 1.0, 0.0 ],
    "direction-cosine-z": [ 1.0, 0.0, 0.0 ],
    "x": 1.0,
    "y": 2.0,
    "z": 3.0
}
```

The model represents Cartesian coordinates, hence the `Euclidean` type. Moreover, it references the previously defined pose relation via the `of-pose` property as required by the `PoseReference` structural constraint. A different option (not shown in the example) is to directly embed the coordinate representation into the pose relation. This, however, is in general not advisable because the same physical pose can be represented by an infinite amount of coordinate representations, i.e. spatial relations are properties of coordinate representations but not vice versa. The next property required by any coordinate representation is the coordinate system that the spatial relation is measured or expressed in as represented by the `as-seen-by` symbolic pointer. Additionally, coordinates must be accompanied by a `unit` of measurement which must conform to and complements the quantity kind in the spatial relation. Finally, in the model above we choose to represent

[15] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/cartesian-coordinates.json

the pose by three direction cosines, the columns of the rotation matrix, and a vector with the entries named as x, y and z.

A wide range of coordinate representations of positions, orientations and poses exist such as cylindrical, spherical or projective/homogeneous coordinates, all of which require their own model and metamodel representations. Similarly, velocity and acceleration representations demand for further metamodels. Discussing those metamodels, however, is out of scope for this document.

2.4 Dynamics

The equations of motion for mechanical systems are of second-order, i.e. the Newton-Euler equations for a rigid body relate acceleration and force via the body's inertia. Hence, for solving those equations a specification of the inertia is required as shown in the dynamics model[16].

The following diagram (Figure 5) graphically represents three types of inertia attachments: a point mass m, a rotational inertia modelled by the principal moments of inertia I_{xx} and I_{yy} as well as a density over a volume $\int_V \rho(\vec{r})dV$.

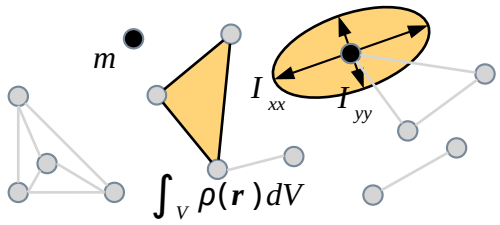

Figure 5: Different inertia attachments

The textual model below gives an example of an inertia attachment to the second link in the kinematic chain. The so-called rigid-body inertia composes linear inertia (mass) and rotational inertia into the same model. While this inertia specification is attached to a rigid body via the `of-body` property we chose to merge the coordinate-free representation with the coordinates. The coordinate-free version of rotational inertia requires a `reference-point` property and an invariant `quantity-kind` of `MomentOfInertia`. Here, the coordinates are then expressed in units of $[kg \cdot m^2]$ with respect to the `as-seen-by` frame that references the second link's root frame and plays the role of a coordinate system. Three properties, namely `xx`, `yy` and `zz` represent the concrete values. A similar representation is used for mass, it is just independent of a reference point or coordinate frame.

```
{
    "@id": "rob:link2-inertia",
    "@type": [ "RigidBodyInertia", "Mass",
               "RotationalInertia",
               "PrincipalMomentsOfInertiaXYZ" ],
    "of-body": "rob:link2",
    "reference-point": "rob:link2-root-origin",
    "as-seen-by": "rob:link2-root",
    "quantity-kind": [ "MomentOfInertia", "Mass" ],
    "unit": [ "KiloGM-M2", "KiloGM" ],
    "xx": 0.1,
```

[16] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/dynamics.json

```
    "yy": 0.1,
    "zz": 0.1,
    "mass": 1.0
}
```

2.5 Kinematic chain

The kinematic chain model[17] shows two types of motion constraints. The joint is the simplest motion constraint between attachments such as frames on two bodies. In the following example we recognize this by the `between-attachments` property. More specifically the joint is a revolute joint as already mentioned before. Hence, two more properties are required: the `common-axis` around which the relative motion occurs and the `origin-offset` that defines the distance or position of the two frames' origins along the common axis.

```
{
    "@id": "rob:joint1",
    "@type": [ "Joint", "RevoluteJoint" ],
    "between-attachments": [
        "rob:link1-joint1",
        "rob:link2-root"
    ],
    "common-axis": "rob:joint1-common-axis",
    "origin-offset": "rob:joint1-offset"
}
```

The kinematic chain is a composite motion constraint with a set of joints as its property as seen below. For the 1-DoF example obviously only one joint is required. The kinematic chain imposes a semantic constraint that all referenced joints must be part of the same graph.

```
{
    "@id": "rob:kin-chain1",
    "@type": "KinematicChain",
    "joints": [ "rob:joint1" ]
}
```

2.5.1 Coordinates

For a real robot the joint configuration can often, but not always, be directly acquired from sensors such as encoders associated with a joint. The model of such a sensor would be a higher-order relation that refers to the same attachments that are also used by the joint. In this tutorial however, we rely on a fixed value to represent the joint position as realized by the joint coordinates model[18].

2.6 Operators and algorithms

Most modelling approaches for kinematic chains stop with a structural model similar to what we have discussed up to now. Such a model would then be interpreted by software to configure a solver for kinematics or dynamics queries. Hence, the source code of that software describes the kinematic chain's behaviour.

[17] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/chain.json
[18] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/joint-coordinates.json

In contrast, here we also establish an explicit model of such behaviour. This model[19] exemplifies a behavioural model of a simple forward position kinematics algorithm. To this end it employs two operators. The `ForwardPositionKinematics` for a single joint, as shown in the following textual representation, requires two input properties, namely a model of the `joint` and the `joint-space-motion` as discussed in the context of the kinematic chain. Given those two inputs, the operator computes the `cartesian-space-motion` as output which here is a symbolic pointer to a pose relation.

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

The second operator, `ComposePose`, composes the pose over the first link and the pose over the joint, the latter of which is computed by the forward position kinematics operator above. This results in the relative pose of the most distal frame "link1-root" with respect to the most proximal frame "link0-root".

Both of the previous operators describe *what* to compute. However, the order of those computations is still missing: clearly, the forward position kinematics must be evaluated before the composition of the poses as there exists a data dependency between both of them. We call the model of such an ordered computation a `Schedule`. A schedule has an ordered list of symbolic pointers to the individual operators in the `trigger-chain` property. The model below shows the schedule for the full forward kinematics algorithm.

```
{
    "@id": "rob:schedule1",
    "@type": "Schedule",
    "trigger-chain": [ "rob:fpk1", "rob:compose-pose1" ]
}
```

2.7 Discussion about composable models

In this tutorial we have discussed how to build composable models for robots. Specifically, the tutorial has focused on the description of a kinematic chain's structure, including geometry, dynamics and motion constraints, as well as its behaviour in terms of a kinematics algorithm. While the example is a simple 1-DoF kinematic chain the concepts transfer to more complex chains with tree structures or parallel structures and more complex algorithms such as solvers for problems involving velocities, accelerations or dynamics.

While we have discussed how to construct the models, we have not described why they are composable. We consider the following points as core contributors towards composability:

- The first class comprises the *structure* of models and metamodels:
  - Every model has an *identifier*, the `@id`, and hence can be referenced from other models. In other words *every* model is reified. The reification of relations enables higher-order relations and indeed the majority of models are actually higher-order composition structures.
  - The identifiers in the form of *symbolic pointers* also contribute to the loose coupling of models: symbolic pointers do not impose type constraints and additionally they separate the specification of the symbolic pointer from it resolution.

[19] https://github.com/comp-rob2b/modelling-tutorial/blob/main/models/fpk-algorithm.json

  - In contrast to the more common, human-facing domain-specific languages (DSL), composable models appear in some sense *inverted*. DSLs frequently feature a single top-level concept (with vagues or sometimes even ambiguous names such as "package" or "document"). Then any further concept must be placed somewhere hierarchically below that top-level concept. As a consequence any change to the lower concepts propagates up in the hierarchy, meaning that the whole metamodel must be developed in lock-step. Once all changes are incorporated a new version of *the* language can be released. In contrast, composable models are not restricted towards the "top". Metamodels can originate from diverse sources that can remain loosely coupled and developed independently.

- A further contributor is the conscious *design* of models and metamodels, often following established best practices:
  - Good metamodels should be *normalized*, as in database normalization, which means that they separate domain concepts as much as possible and leave no implicit assumptions both in the structure and in the meaning of the involved terminology. As a result, a model should specify exactly what is required, neither more nor less.
  - Closely related to the normalization is the clear separation of (intrinsic) properties and (extrinsic) attributes. The former should be part of models whereas the latter are usually better represented as (reified) relations.
  - In most models above we have already seen the utility of *multi-conformance*, i.e. a model can conform to more than one constraint as imposed by a single type. Most state-of-the-art modelling approaches support only single conformance of models to metamodels. That means a model has a single type and all the model's properties are defined in a single metamodel. To emulate behaviour similar to multi-conformance, those approaches rely on subsumption (`is-a`) relations on the metamodel level.

- Finally, a key feature of composable models is the *open-world assumption*:
  - For composable models there does not exist *the* global schema to check for well-formedness. Instead, it is up to the tools to decide which structural constraints they must enforce or check. For example, a simple viewer may treat joints as primitives in its context while a code generator most likely requires knowledge about the types of joints. Implementation-wise the SHACL[20] W3C standard nicely complements this effort by supporting composable well-formedness constraints.
  - Recall that multiple entities like coordinate, shape or inertia representations can be attached to a kinematic chain. But it is *not* the decision of the kinematic chain model to "close the world", i.e. to decide which of those representations to use in a specific context. Instead, this decision is left to some higher-order or application-level model.
  - When multiple JSON objects have the same `@id` they all refer to the same entity. Hence, a model can be logically separated. Note that we have not made use of this feature here.

[20] https://www.w3.org/TR/shacl/

3 FloorPlan DSL

3.1 How to specify an indoor environment

The ExSce-FloorPlan DSL is a domain-specific language and tooling for specifying and generating indoor environments. In this tutorial we are going to go over the most important concepts to model a concrete floor plan.

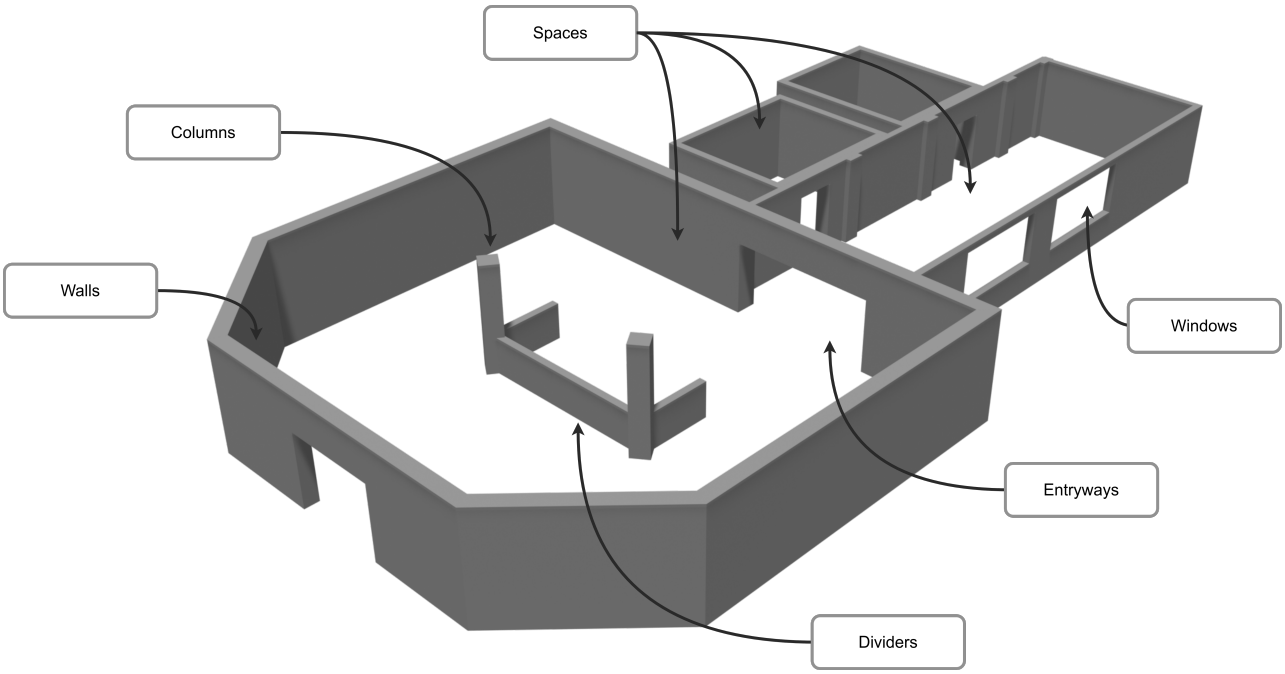

Figure 6: Environment generated with the tooling with concepts annotated

The goal of this tutorial is to create the environment above. We will go over concepts such as Spaces, Entryways, and other features in order to specify a specific environment.

3.1.1 Concepts available

Let's do a review of the most important concepts when modelling an indoor environment:

- Spaces: these are the central concepts when modelling. This concept allows you to model any space in a floor plan: a room, a hallway, an intersection, a reception, and any other space that is surrounded by walls.
- Walls: walls surround the spaces, and these are not modelled directly. When modelling a space, you select the shape of the space, and the walls are created automatically when interpreting the model.
- Entryways and Windows: These are openings in the walls that allow you to connect two spaces.
- Column and Dividers: These are features of the environments that are usually located freely in space or next to walls. Both cases can be modelled using the language.

3.1.2 Modelling

Modelling an indoor environment consists of declaring the Spaces, Entryways, Windows, Columns, and Dividers; and specifying their location in the environment. Each of these concepts is modelled by specifying its

shape and other attributes such as thickness or height. Specifying the location is simple, but requires some background.

The location of any space or feature is specified by a translation and rotation with regards to a frame of reference. There are multiple frames of references that can be chosen for this. Apart from the world frame, each space has N + 1 frames of references that can be selected, where N is the number of walls.

For each wall in the space, there is a frame located in the middle of the wall, with the x axis going along the wall and the y axis perpendicular to the wall. From the perspective of being inside the room looking into one of the walls: Positive values in the x axis are located from the centre to the right, and negative values in the opposite direction. Whereas the positive direction from the y axis moves away from you and the negative direction moves closer. The frame is located at floor level, meaning that for the z axis only positive values are above the floor.

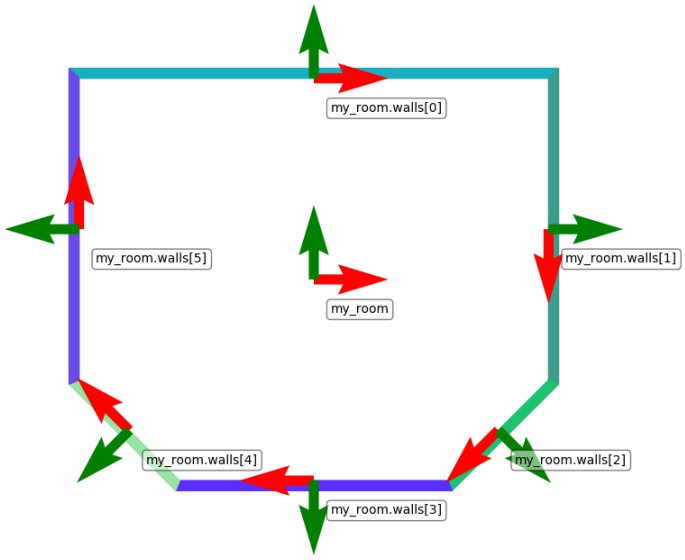

Figure 7: Frames available when modelling

The image above illustrates a room with all of its frames. Each wall has an index, so you can select the frame of reference by specifying the index of the desired wall: `<name of space>.walls[<index>]`. You can also select the center frame of the space by just referring to the name: `<name of space>`. You may also select the world frame with the `world` keyword.

A space requires two frames in order to specify a location: A reference frame where the translation and rotation are specified from, and the frame that will get translated and rotated.

Using the world frame You can select the world frame as your frame of reference by using the keyword `world`. You may only use this frame for locating spaces. Any other feature or entryway must be specified by either the center frame or one of the walls. Should be noted that you can use the `this` keyword to reference a frame when you are inside the scope of the space that frame belongs to.

```
location:
    from: world
    to: this
    pose:
        translation: x:3.0 m, y:4.0 m
        rotation: 45.0 deg
```

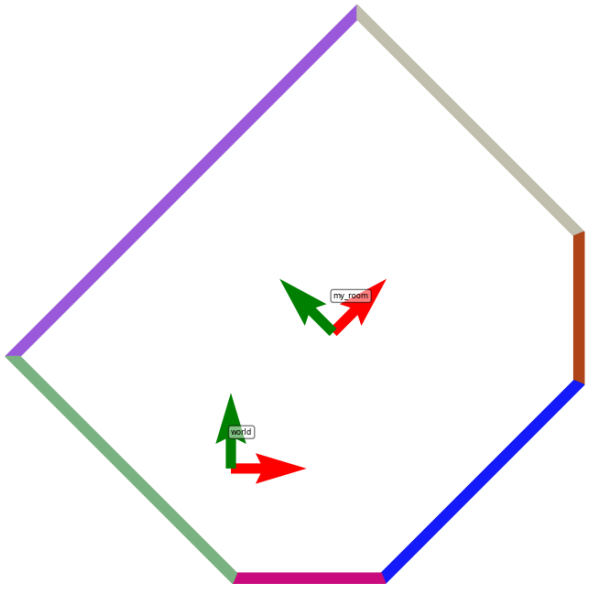

Figure 8: Pose of a space with regard to the world frame

Using two wall frames You can also use two wall frames to define locations. When you model a location using two wall frames, the default behaviour is to do an extra 180 degree rotation of the space you are locating so that the two spaces are not overlapping. Depending on the two walls that are chosen, the results can be different, as illustrated in the two next examples

```
location:
    from: my_room.walls[1]
    to: second_room.walls[0]
    pose:
        translation: x:0.0 m, y:0.0 m
        rotation: 0.0 deg
    spaced
```

```
location:
    from: my_room.walls[1]
    to: second_room.walls[1]
    pose:
        translation: x:0.0 m, y:0.0 m
        rotation: 0.0 deg
    spaced
```

The flag `spaced` is used to tell the interpreter to calculate the combined thickness of the two walls, and space the two rooms accordingly. When not present, the two rooms are not spaced correctly, as seen in the next figure.

Similarly, the default alignment behaviour can be disabled by using the `not aligned` flag, so that the two rooms overlap.

```
location:
    from: my_room.walls[1]
```

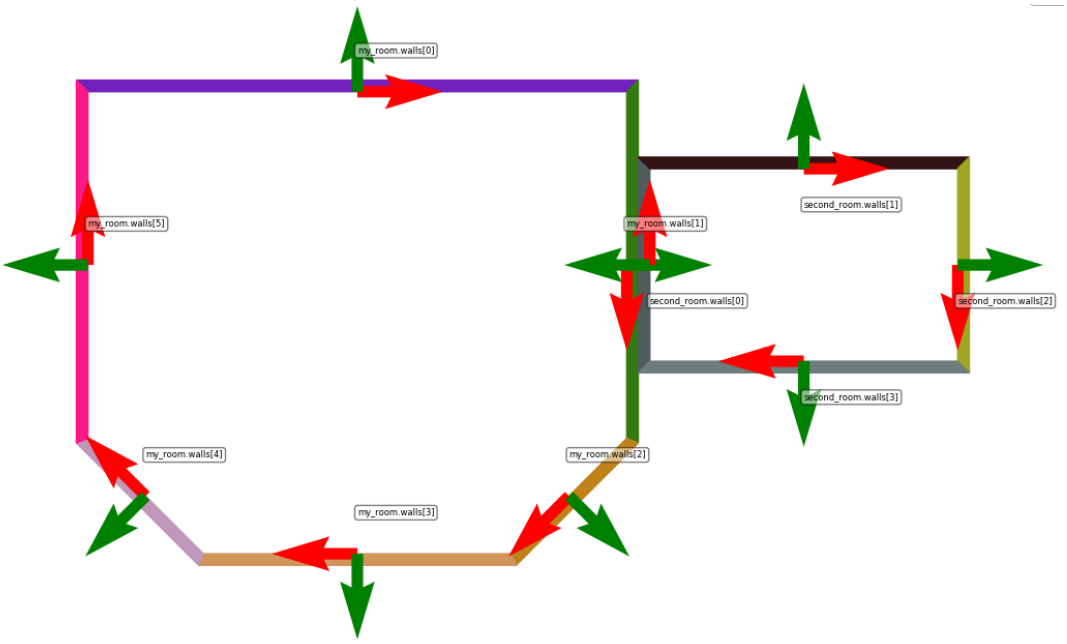

Figure 9: Pose of two spaces when walls are used as reference frames

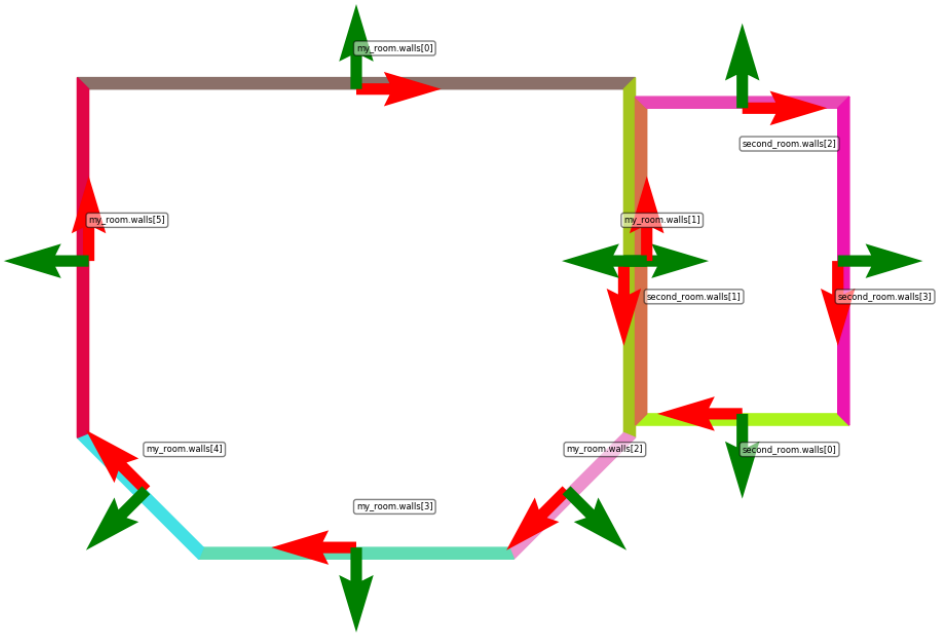

Figure 10: Pose of two spaces when another wall is used as a reference frame

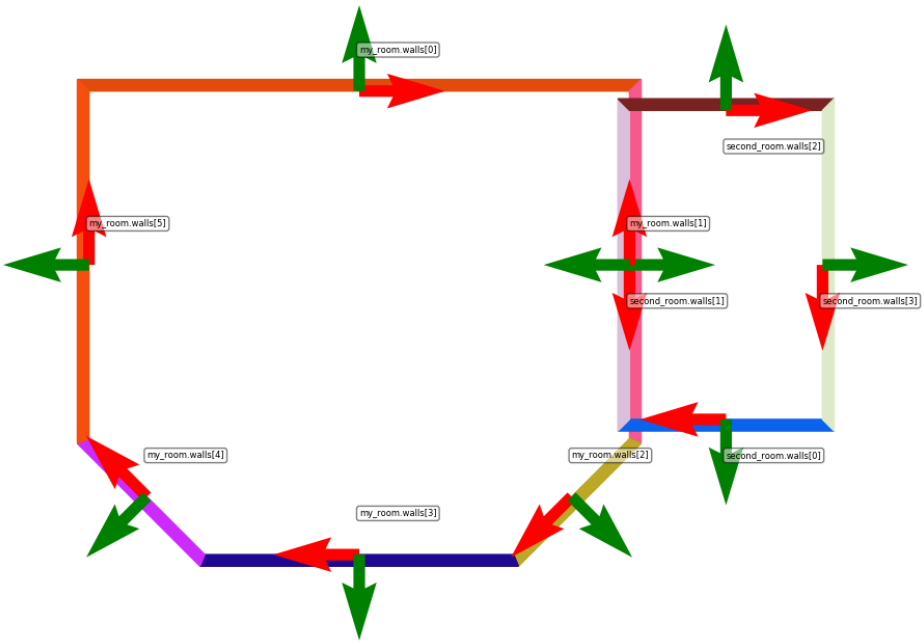

Figure 11: Two spaces not spaced correctly, as the `spaced` flag was not included

```
    to: second_room.walls[1]
    pose:
        translation: x:0.0 m, y:0.0 m
        rotation: 0.0 deg
    not aligned
```

3.1.3 Features

The language enables the modelling of doorways, windows, columns, and dividers. Features such as columns or dividers are always defined within a space scope, so you can use the “this” keyword to refer to the walls inside the space.

```
Column wall_column:
    shape: Rectangle width=0.5 m, length=0.3 m
    height: 2.5 m
    from: this.walls[3]
    pose:
        translation: x:7.0 m, y:0.0 m, z: 0.0 m
        rotation: 0.0 deg
```

Entryways and windows are specified outside of the scope of the space, after all the spaces in the floorplan have been specified. These features create the openings in the walls required to connect two spaces or one space with the “outside”.

```
Entryway doorway:
    in: my_room.walls[0] and second_room.walls[1]
    shape: Rectangle width=1.0 m, height=1.8 m
    pose:
        translation: x: -1.0 m, y: 0.0 m, z: 0.0 m
        rotation: 0.0 deg
```

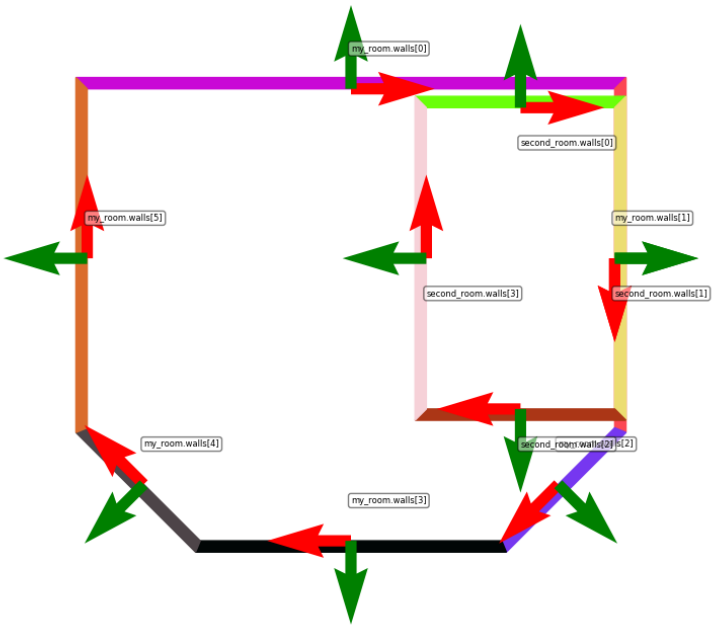

Figure 12: Two spaces not aligned as the `not aligned` flag was used

Whenever an entryway or window is located in a wall shared by two spaces, you must specify the two walls that will be opened by the entryway or window (`my_room.walls[0] and second_room.walls[1]`). However, The location is specified with regards to the first frame specified, in the example above it would be `my_room.walls[0]`.

3.2 Modelling the example

Now that we have reviewed all of the important concepts, we can put them together in a model. The finished model for this tutorial is available here[21]. In this section we will go over the model section by section with some explanations when needed.

```
Floor plan: hospital

    Space reception:
        shape: Polygon points:[
            (-7.0 m, 6.0 m),
            (7.0 m, 6.0 m),
            (7.0 m, -3.0 m),
            (4.0 m, -6.0 m),
            (-4.0 m, -6.0 m),
            (-7.0 m, -3.0 m)
        ]
        location:
            from: world
            to: this
            pose:
                translation: x:0.0 m, y:-5.0 m
                rotation: 45.0 deg
```

[21] ../models/examples/hospital.floorplan

Each floor plan has a name, which gets used to identify all the artefacts that get generated. The `reception` space has a custom polygon as shape, so we specify all the points to bound it. Every pair of points is a wall, with the last point and the first point being the final wall to close the polygon (i.e. no need to repeat the first point at the very end). From the world frame, this space is translated -5 metres in the y axis and rotated 45 degrees w.r.t. the z axis.

```
        ...
        wall thickness: 0.40 m
        wall height: 3.0 m
        features:
            Column central_left:
                shape: Rectangle width=0.5 m, length=0.5 m
                height: 3.0 m
                from: this
                pose:
                    translation: x:-2.5 m, y:0.0 m
                    rotation: -35.0 deg
        ...
```

The `reception` space will have a wall thickness of 0.4 metres and a wall height of 3 metres. These values don't have to be specified for every space, as default values will be set later for all spaces. Nested in the feature concept, columns and dividers can be specified. The reference frame for these is always part of the space (either the space frame or a wall frame of the `reception` space)

```
    Space hallway:
        shape: Rectangle width=5.0 m, length=14.0 m
        location:
            from: reception.walls[0]
            to: this.walls[2]
            pose:
                translation: x:2.0 m, y:0.0 m
                rotation: 0.0 deg
            spaced
        ...

    Space room_A:
        ...

    Space room_B:
        ...
```

The hallway is located using two wall frames, and the flag `spaced` is present so that the interpreter calculates spacing necessary to avoid overlapping in between the spaces. Two other spaces also get defined in a similar way.

```
    Entryway reception_main:
        in: reception.walls[3]
        shape: Rectangle width=2.5 m, height=2.0 m
        pose:
            translation: x:0.0 m, y: 0.0 m, z: 0.0 m
            rotation: 0.0 deg

    Entryway reception_hallway:
```

```
        in: reception.walls[0] and hallway.walls[2]
        shape: Rectangle width=4.0 m, height=2.0 m
        pose:
            translation: x: 2.0 m, y: 0.0 m, z: 0.0 m
            rotation: 0.0 deg

    ...
    Window hallway_window_1:
        in: hallway.walls[1]
        shape: Rectangle width=3.0 m, height=1.5 m
        pose:
            translation: x: 3.0 m, y: 0.0 m, z: 0.8 m
            rotation: 0.0 deg
```

Two entryways and one window are modelled, each with a unique name. In the case of the second entryway, since it connects two spaces we must specify the two walls where the entryway is located. Notice that for windows we have to specify a translation in the z axis.

```
    Default values:
        Wall thickness: 0.23 m
        Wall height: 2.5 m
```

At the very end of the model the default values for all the spaces must be specified.

3.2.1 How to generate 3D files and occupancy grid maps

Once all requirements are installed, as specified here[22], you can get artefacts generated. To interpret the model and get artefacts, you only need to run one command:

```
blender --python src/exsce_floorplan/exsce_floorplan.py --background
--python-use-system-env -- <path to model>
```

The `--` after the variable paths are important to distinguish the blender parameters and the parameters for the tooling. You will obtain an occupancy grid map and a `.stl` file with the 3D mesh of the environment.

3.3 Concepts of the FloorPlan DSL

3.3.1 Space

Space concepts are the main concepts in a floor plan. They can be used to describe any room or hallway as long as it is bounded by walls. The concept is agnostic to the functionality of the space itself. i.e. you can model a reception, a hallway, a storage room, a sleeping room, a waiting area, and any other space with the same concept.

Attributes:

- name: name for the space, should be unique.

[22] https://github.com/sesame-project/FloorPlan-DSL

- shape: A shape that describes the boundaries of the space. The boundaries of the space will become the walls.
- location: A location description:
  - from: The frame of reference for the location, can be `world` to refer to the world frame or `<space>` to refer to the space frame of another space (i.e. the centre of a rectangle or circle), or `<space>.walls[<index>]` to refer to a wall of a space.
  - to: The frame of the space that you are locating. All walls and features of the space will keep their pose with regards to this frame. The value can be `this` to refer to the space frame of this space (i.e. the space you are modelling), or `this.walls[<index>]` to refer to one of the walls
  - pose: A pose description. It contains a translation in the `x` and `y` axis, and a rotation w.r.t. the `z` axis.
  - spaced (optional, recommended): A flag to tell the interpreter that it must calculate the correct space between the two spaces to ensure no overlap.
  - not aligned (optional): A flag to tell the interpreter to not perform the default behaviour of aligning two spaces when two wall frames are used.
- wall thickness (optional): The wall thickness for the walls of the space, when the desired value is different than the default.
- wall height (optional): The wall height for the walls of the space, when the desired value is different than the default.
- features: A set of features.

```
Space <name>:
        shape: <shape>
        location:
            from: <frame>
            to: <frame>
            pose:
                translation: x: 0.0 m, y:0.0 m
                rotation: 0.0 deg
            {spaced}
            {not aligned}
        {wall thickness: 0.0 m}
        {wall height: 0.0 m}
        {features:
            <feature>
        }
```

3.3.2 Shapes

Rectangle (for Space or Feature)

Attributes:

- width: Float, in metres
- length: Float, in metres

```
Rectangle width=0.0 m, length=0.0 m
```

Rectangle (for Entryway or Window)

Attributes:

- width: Float, in metres
- height: Float, in metres

```
Rectangle width=0.0 m, height=0.0 m
```

Polygon (for Space or Feature)

Attributes:

- points: Set of Points, the points are specified w.r.t to the space frame of the polygon, which is aligned with the world frame (no rotation on any axis).

```
Polygon points:[
            <points>,
        ]
```

Point

Attributes:

- x: Float, in metres
- y: Float, in metres

```
(0.0 m, 0.0 m)
```

3.3.3 Entryway

The entryway concept is used to model the space for doorways and other openings between rooms and the outside.

Attributes:

- Name: name for the entryway, should be unique.
- in: wall reference (`<space>.walls[<index>]`). The frame associated with this wall will be used as the reference frame for the location of the entryway. When an entryway is between two spaces, both walls have to be specified. The first frame remains as a reference frame.
- shape: Shape for the entryway.
- pose: A pose description. It contains a translation in the `x` and `z` axis, and a rotation w.r.t. the `y` axis. Translations in the y axis should be avoided for appropriate results.

```
Entryway <name>:
    in: <wall reference 1> {and <wall reference 2>}
    shape: <shape>
    pose:
        translation: x: 0.0 m, y: 0.0 m, z: 0.0 m
        rotation: 0.0 deg
```

3.3.4 Window

Attributes:

- Name: name for the window, should be unique.
- in: wall reference (`<space>.walls[<index>]`). The frame associated with this wall will be used as the reference frame for the location of the window. When a window is between two spaces, both walls have to be specified. The first frame remains as a reference frame.
- shape: Shape for the window.
- pose: A pose description. It contains a translation in the `x` and `z` axis, and a rotation w.r.t. the `y` axis. Translations in the y axis should be avoided for appropriate results.

```
Window <name>:
    in: <wall reference 1> {and <wall reference 2>}
    shape: <shape>
    pose:
        translation: x: 0.0 m, y: 0.0 m, z: 0.0 m
        rotation: 0.0 deg
```

3.3.5 Floor Features

Features are common in the floor of a floor plan.

Column

Attributes:

- Name: name for the column, should be unique.
- shape: Shape for the column.
- height: Height of the column, in metres.
- from: Frame of reference, it can be: space frame of the space (`this`) or wall reference (`this.walls[<index>]`)
- pose: A pose description. It contains a translation in the `x` and `y` axis, and a rotation w.r.t. the `z` axis.

```
Column <name>:
    shape: <shape>
    height: 0.0 m
    from: <frame>
    pose:
        translation: x:0.0 m, y:0.0 m
        rotation: 0.0 deg
```

Divider

Attributes:

- Name: name for the divider, should be unique.
- shape: Shape for the divider.
- height: Height of the divider, in metres.
- from: Frame of reference, it can be: space frame of the space (`this`) or wall reference (`this.walls[<index>]`)
- pose: A pose description. It contains a translation in the `x` and `y` axis, and a rotation w.r.t. the `z` axis.

```
Divider <name>:
    shape: <shape>
    height: 0.0 m
    from: <frame>
    pose:
        translation: x:0.0 m, y:0.0 m
        rotation: 0.0 deg
```

3.4 How to introduce variations into the environment

The objective of this tutorial is to demonstrate how to specify variations of FloorPlan DSL models and generate new concrete environments.

The variations of an environment are specified with the Variation DSL in a separate model. We create a new file and give it a name with the appropiate format: `<file_name>.variation`. The first line of the model imports the concrete environment where we will introduce the variations. The variation model is available here[23].

```
import "hospital.floorplan"
```

We can then declare the spacial attributes we wish to associate to one of the three probability distributions: normal, discrete, and uniform:

```
<attribute>: normal(mean=<mean value, in meters>, std=<standard deviation, in
 ↪  meters>)
<attribute>: discrete([
        (<weight 1>, <value, in meters>),
        (<weight 2>, <value, in meters>),
        ...
        (<weight n>, <value, in meters>)
    ]) // Important: all the weights must sum to 1
<attribute>: uniform([<value 1, in meters>, <value 2, in meters>, ... , <value
 ↪  m, in meters>])
```

We can refer to attributes through refences to the corresponding spaces or features. We need to provide the FQN of the attribute inside of the scope of the refence.

```
hallway: {
    location.pose.translation.x : normal(mean=0.0, std=5.0)
    wall_thickness : discrete([
        (0.2, 0.14),
        (0.4, 0.35),
        (0.4, 0.51)
    ])
}
```

For features such as columns and dividers, we need to provide the FQN that include the space they belong to:

[23] ../models/examples/hbrs.variation

```
reception.divider_central: {
    height : normal(mean=1.0, std=0.2)
    location.pose.rotation : discrete([
        (0.8, 0.0),
        (0.2, 0.20)
    ])
}
```

After we have specified all the probability distributions, we can generate as many variations as we desired:

```
textx generate <input model> --target exsce-floorplan-dsl --variations <number
↪  of variations> --output <output folder>
```

Each resulting concrete environment will follow the format `<name of model>_<seed number>.floorplan` and can be found at the specified output folder. These models are ready to be transformed into 3D models and other artefacts as shown in the previous tutorial. At the moment the generator does not check for the soundness of the resulting floor plan, nor for uniqueness.

3.5 Tutorial: modelling objects with movement constraints and placing them in indoor environments

This chapter contains 3 tutorials with the objective of building an understanding of the composable modelling approach, and how it is used to extend the use case of the FloorPlan DSL. The first tutorial covers the concepts behind modelling a common object with a motion constraint: a door. Then, in the second tutorial we place an instance of the door in the floor plan generated by the FloorPlan DSL. The final tutorial covers the modelling of states, and how we can assign an initial state to a door instance. These tutorials build upon ideas presented in the modelling kinematic chains tutorial[24]; as well as ideas presented in the modelling an indoor environment tutorial[25].

3.5.1 Background

Composable modelling enables the creation of semantic rich models that are easily extendible and reusable. We use JSON-LD[26] to represent our models and metamodels, as the format allows composition by linking models through unique identifiers. Composable models expressed in JSON-LD consist of a `@graph` with elements that conform to one or multiple metamodels, which are specified in a `@context`. Each element is a JSON object with a unique identifier (`@id`), a list of metamodel concepts it conforms to (`@type`), and a set of properties conforming to the metamodel concepts. For example, a composable model of a vector can be modelled by (i) giving the model a unique id, (ii) selecting the list of relevant concepts for a bound vector in 3D space, and (iii) referring to a 3D Point model for the `start` property. This model of a vector can be referred to by its identifier by another model in the graph.

```
{
    "@id": "vector-joint-door-hinge-joint-x",
    "@type": [ "3D", "Euclidean", "Vector", "BoundVector", "UnitLength" ],
    "start": "point-door-hinge-origin"
}
```

[24] https://github.com/comp-rob2b/modelling-tutorial
[25] https://github.com/sesame-project/FloorPlan-DSL/blob/main/docs/Tutorial.md
[26] https://www.w3.org/TR/json-ld/

The FloorPlan DSL can transform the TextX[27] models into composable models. This has the ability to extend the possible applications for the models. For instance, the tool used in this tutorial is a companion tool that allows to model objects with movement constraints and place them in the indoor environments described in the FloorPlan DSL. This companion tool was developed using the RDFLib[28], which is used to navigate the graph and interpret the models. While there is no tutorial on creating such a tool, the companion tool presented in this repository is well documented and can be used as a guide for developing new tools.

The companion tool presented not only consumes the composable models from the FloorPlan DSL, but also reuses other metamodels from the ExSce workbench in order to model dynamic objects. These are the kinematic chain metamodel and the finite state machine metamodel. The kinematic chain metamodel is used to model the motion constraints of the dynamic objects that will be placed in the indoor environments. Whereas the finite state machine metamodel is used to model the states of these dynamic objects and their transitions. The states are used to specify the starting state of an object in the simulation. While the robot can interact with these dynamic objects, there is no dynamic change of state based on events or timers.

3.5.2 How to: Model a Door

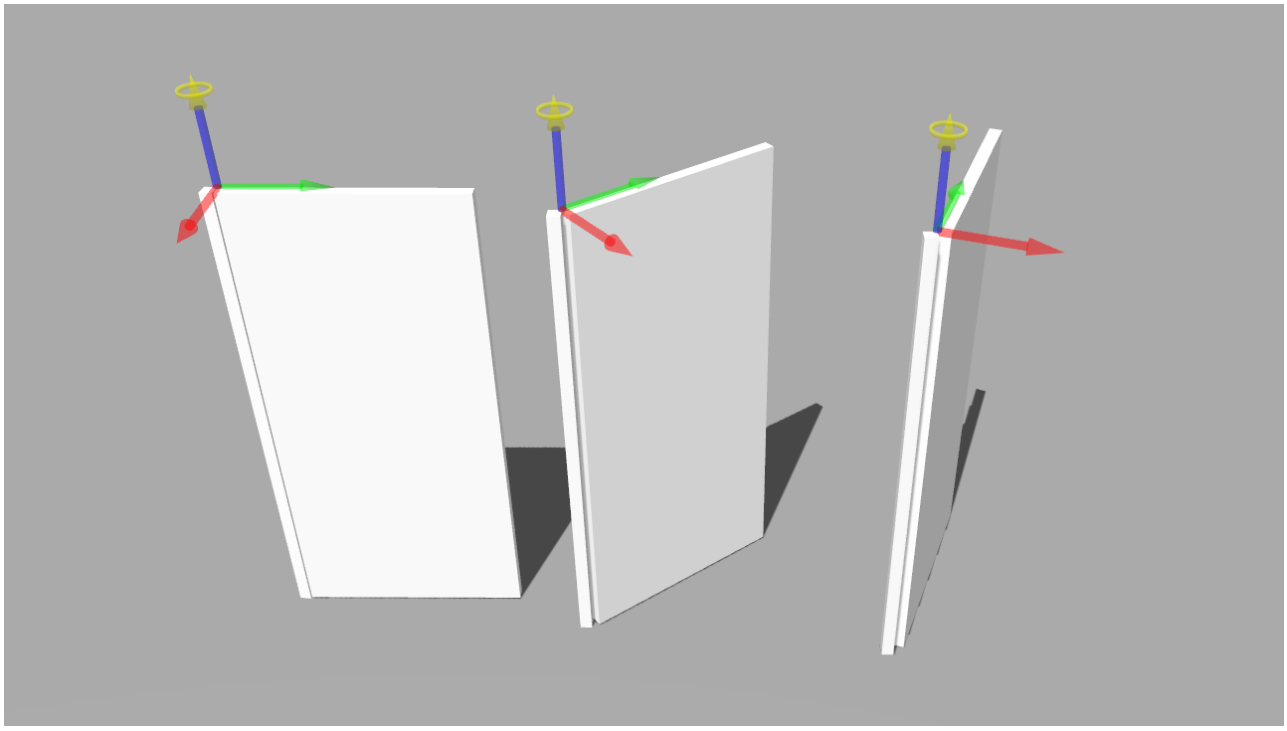

Figure 13: A door model

A door is an object with a motion constraint. It has a hinge that attaches the door to the doorway, and which constraints its movement to a swing action that allows for opening and closing. In this tutorial we will review the modelling process to create a door with our tool.

A door can be modelled as a kinematic chain, with two links representing the doorway and the door itself. A revolute joint represents the hinge, and it joins the two links and constrains the movement of the door with regards to the doorway. All the following model snippets, as well as the complete json-ld model of the door is available here[29].

[27] https://textx.github.io/textX/3.1/
[28] https://rdflib.readthedocs.io/en/stable/
[29] ../input/object-door.json

The first elements to model are the geometric skeleton of the door. Those are points, vectors, and frames. For our door we need to model 5 frames: the `door-object` frame is the root of the object. This frame is later used to specify a pose in the world with an object instance. The door is made up of two links and a joint. Each link can have multiple frames associated with it. In this case, one at the centre of each link (`door-holder-frame` and `door-body-frame`), and two frames of coincident origins for the joint (`door-holder-hinge` and `door-body-hinge`). For these last two, we model the frame not only as an entity with an origin point, but also its three direction vectors. This allows specifying the axis of the joint where the motion is constrained to.

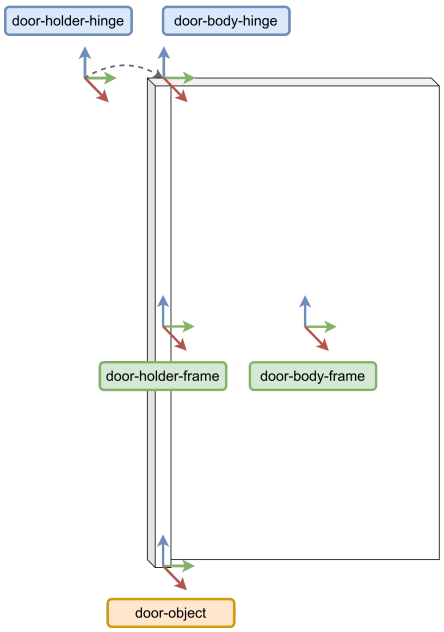

Figure 14: All the frames required to model the door

```
{
    "@id": "point-joint-door-hinge-origin",
    "@type": [ "3D", "Euclidean", "Point" ]
},
{
    "@id": "vector-joint-door-hinge-joint-x",
    "@type": [ "3D", "Euclidean", "Vector", "BoundVector", "UnitLength" ],
    "start": "point-door-hinge-origin"
},

{
    "@id": "frame-joint-door-hinge",
    "@type": [
        "3D",
        "Euclidean",
        "Frame",
        "Orthonormal",
        "RigidBody",
        "RightHanded",
        "OriginVectorsXYZ"
        ],
    "origin": "point-joint-door-hinge-origin",
    "vector-x": "vector-joint-door-hinge-joint-x",
```

```
    "vector-y": "vector-joint-door-hinge-joint-y",
    "vector-z": "vector-joint-door-hinge-joint-z"
}
```

These elements are the building blocks for the spatial relations necessary to position the links and joints in space. The pose is specified for each frame with regards to another frame. This representation is coordinate free, as coordinates are associated with another entity that refers to the pose, as shown in this snippet:

```
{
    "@id": "pose-frame-joint-door-hinge-wrt-frame-door-holder-origin",
    "@type": "Pose",
    "of": "frame-joint-door-hinge",
    "with-respect-to": "frame-door-holder-origin",
    "quantity-kind": [ "Angle", "Length" ]
},
{
    "@id": "coord-pose-frame-joint-door-hinge-wrt-frame-door-holder-origin",
    "@type": [
        "PoseReference",
        "PoseCoordinate",
        "VectorXYZ"
    ],
    "unit": ["M", "RAD"],
    "of-pose": "pose-frame-joint-door-hinge-wrt-frame-door-holder-origin",
    "as-seen-by": "frame-door-holder-origin",
    "x": -0.025,
    "y": 0.025,
    "z": 0,
    "theta": 0
}
```

To model the pose of each link and joint, only 4 pose descriptions are necessary, and they are illustrated here:

For each link in the kinematic chain there is also inertia, visual geometry, and collision geometry information that is required. The joint of the door also requires some information about the zero configuration, its maximum and lowest values, and optionally the states that it can be.

For the link, we must model the rigid body inertia. This is straightforward as it only requires calculating the moment of inertia values, for which there are calculator tools available. In the simulator, in this case Gazebo[30], each link is represented visually and physically with a polytope. The polytope can be modelled in various ways. For our example we model the polytope using the `"GazeboCuboid"` type, which describes a cuboid by its length in the x, z, and y directions. We then link this cuboid model to the visual and physics representation of the door body using the `"LinkVisualRepresentation"` and `"LinkPhysicsRepresentation"`.

```
{
    "@id": "inertia-door-body",
    "@type":  [
        "RigidBodyInertia",
        "Mass",
        "RotationalInertia",
        "PrincipalMomentsOfInertiaXYZ"
        ],
```

[30] https://gazebosim.org/

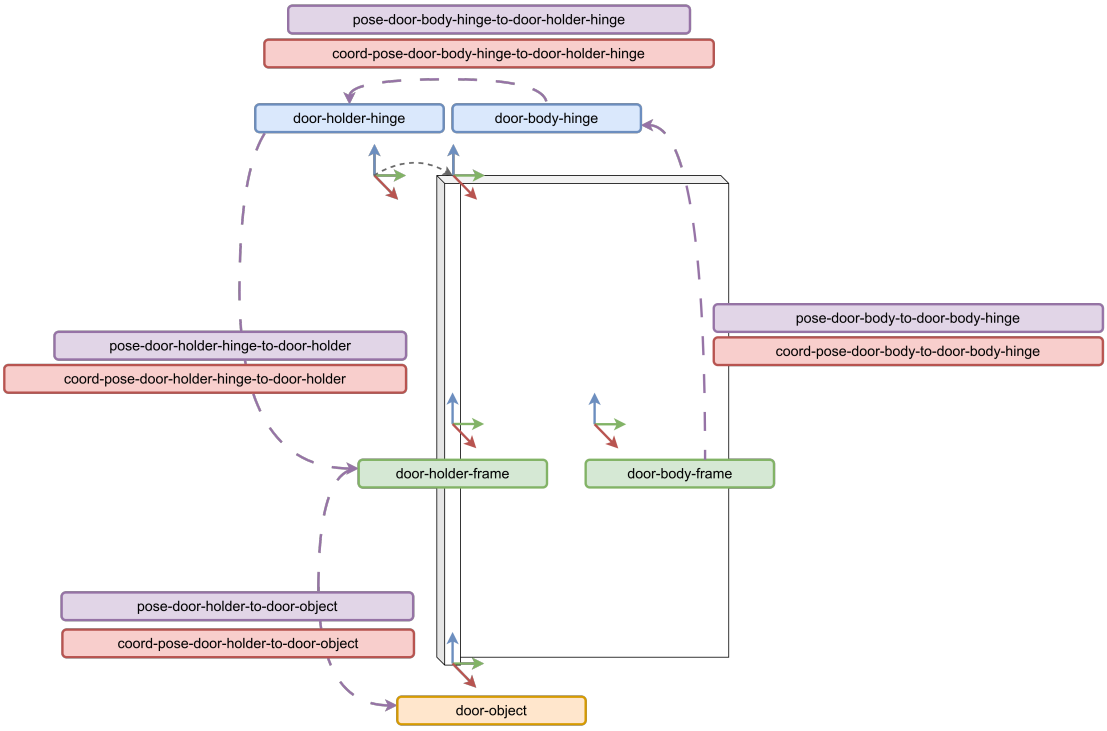

Figure 15: All the pose relations and coordinate references required for the door model

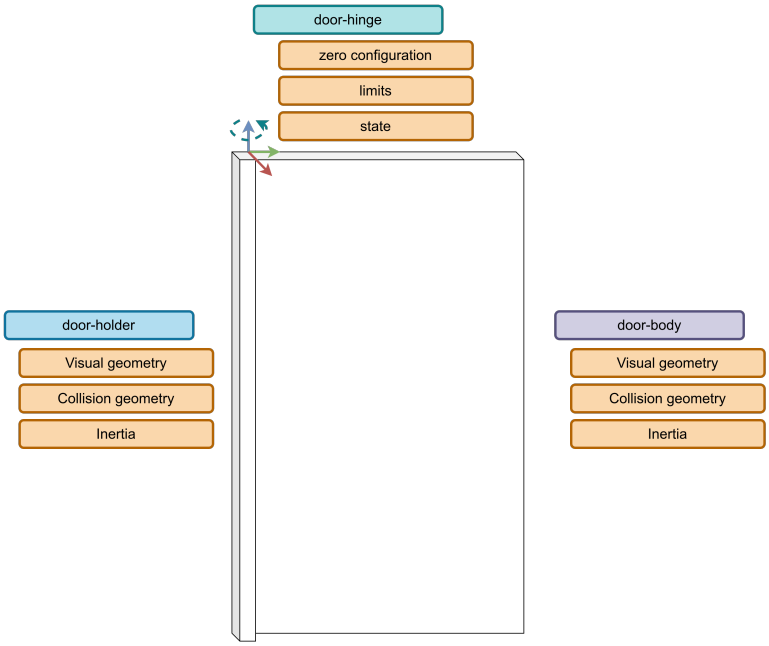

Figure 16: Models required for each link and joint in the kinematic chain

```
    "of-body": "door-body",
    "reference-point": "point-door-body-origin",
    "as-seen-by": "frame-door-body-hinge",
    "quantity-kind": [ "MomentOfInertia", "Mass" ],
    "unit": [ "KiloGM-M2", "KiloGM" ],
    "xx": 0.4069,
    "yy": 0.3322,
    "zz": 0.0751,
    "mass": 1.0
},
{
    "@id": "polytope-door-body",
    "GazeboCuboid""@type": ["Polytope", "3DPolytope", "GazeboCuboid"],
    "unit": "M",
    "x-size": 0.05,
    "y-size": 0.93,
    "z-size": 1.98
},
{
    "@id": "link-visual-door-body",
    "@type": ["LinkWithPolytope", "LinkVisualRepresentation"],
    "link": "door-body",
    "polytope": "polytope-door-body"
},
{
    "@id": "link-physics-door-body",
    "@type": ["LinkWithPolytope", "LinkPhysicsRepresentation"],
    "link": "door-body",
    "polytope": "polytope-door-body"
}
```

The modelling of the joint and the rest of the kinematic chain is explained in more detail in this tutorial[31].

3.5.3 How to: Place an object in the FloorPlan

Placing an object instance in the simulation world requires modelling a new frame of reference. We model a frame with its origin, where the frame is co-located with the object frame of the object instance. This way, by specifying a pose to this frame of reference we can give a pose to the object instance in the world. This relation between the frame of reference of the instance and the object frame of the object is implicit and is enforced by the script that interprets the models. For now we only need to model the frame of reference, and later refer to it when modelling the object instance. The instance placement model is available here[32], and is the source for the following snippets.

```
{
    "@id": "geom:point-location-door-1",
    "@type": "Point"
},
{
    "@id": "geom:frame-location-door-1",
    "@type": "Frame",
    "origin": "geom:point-location-door-1"
},
```

[31] https://github.com/comp-rob2b/modelling-tutorial#kinematic-chain
[32] ../input/object-door-instance-1.json

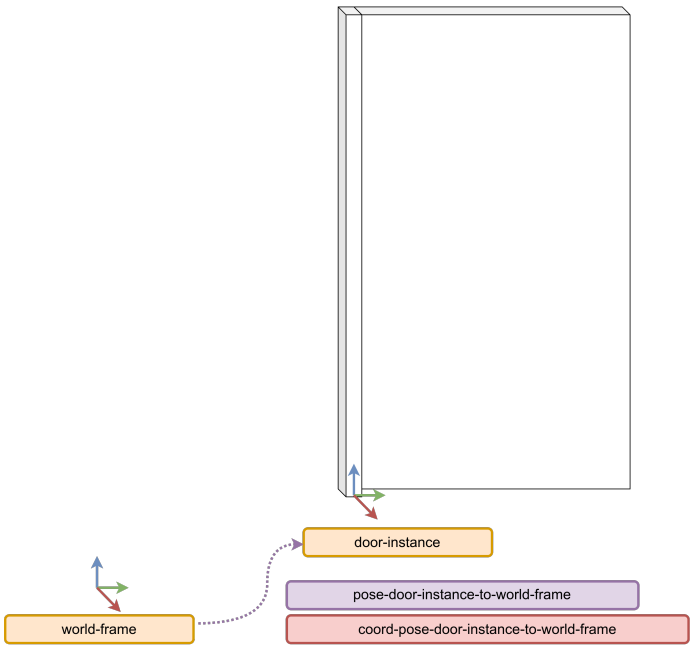

Figure 17: Frames and pose modelled to place an object instance in the world

Since we are using the composable models, we can use any frame from the FloorPlan DSL model to specify the pose relation. For instance, the frame of id `frame-left_long_corridor-wall-1` comes from the floor plan model. We can obtain the composable models from the FloorPlan DSL by using the TextX generator:

```
textx generate <model_path> --target json-ld
```

We can then model the pose relation using one of the frames of the floor plan model. Modelling a pose works in the same way as in modelling the object: we model a coordinate free pose, and then link to it to specify coordinates.

```
{
    "@id": "pose-frame-location-door-1",
    "@type": "Pose",
    "of": "frame-location-door-1",
    "with-respect-to": "frame-left_long_corridor-wall-1"
}
{
    "@id": "door:coord-pose-frame-location-door-1",
    "@type": [
        "PoseReference",
        "PoseCoordinate",
        "VectorXYZ"
    ],
    "of-pose": "door:pose-frame-location-door-1",
    "as-seen-by": "geom:frame-left_long_corridor-wall-1",
    "unit": [
        "M",
        "degrees"
```

```
    ],
    "theta": -90.0,
    "x": 20.50,
    "y": 0.1,
    "z": 0.0
},
```

In our "ModelInstance" entity we can link together the instance frame, the object to be instantiated, and in which world it is instantiated. Optionally, we can also specify an initial state for the objects.

```
{
    "@id": "door-instance-1",
    "@type": "ObjectInstance",
    "frame": "frame-location-door-1",
    "of-object": "door",
    "world": "brsu_building_c_with_doorways",
}
```

After running our tool, we obtain the SDF model file for the object, and a world file (also specified in SDF) for Gazebo. The input parameter for our tool is the folder that contains all the json-ld models. For this example, all the json-ld models are available here[33].

```
python3 main.py <input folder>
```

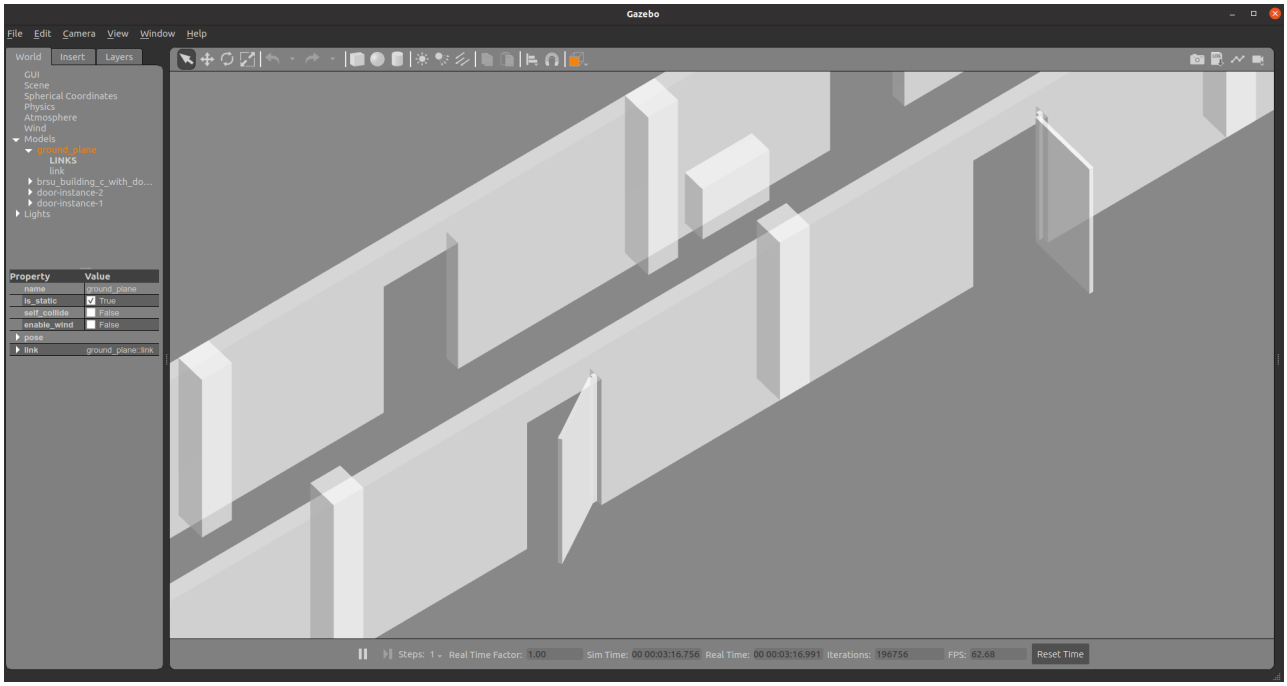

Figure 18: Screenshot of Gazebo showing the doors positioned inside the entryways

3.5.4 How to: Model a state machine and specify intial states

It might be of interest for a test that a door is at a specific state. A finite state machine, such as the one picture above, can be used to model all the states the door can be in and the transitions between them. With kinematic

[33] ../input/

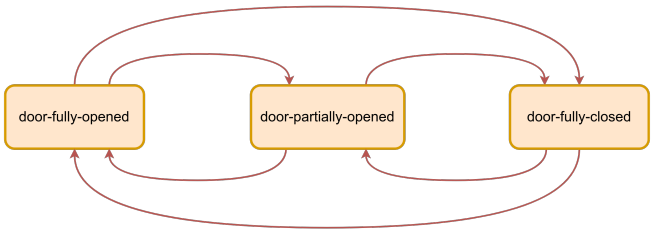

Figure 19: Finite state machine for a door

chains, such as the door, the states can refer to joint positions. A fully closed door is a door where the joint position is set to 0 radians, whereas a fully opened door has the joint position set at 1.7 radians. We compose the joints positions with the states model to specify the pose of the door at a specific state. The object state model is available here[34].

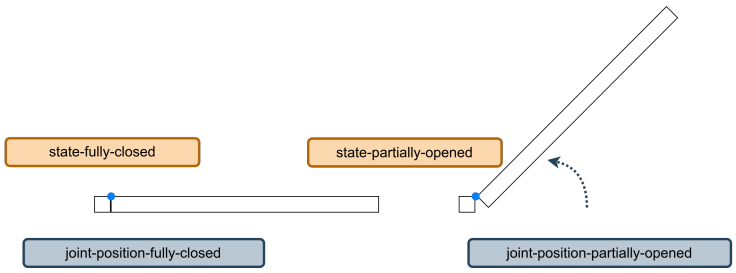

Figure 20: Door states with their joint positions

```
{
    "@id": "door-fully-opened",
    "@type": "State"
},
{
    "@id": "joint-pose-door-fully-opened",
    "@type": [
        "JointReference",
        "JointPosition",
        "RevoluteJointPosition",
        "RevoluteJointPositionCoordinate",
        "JointLowerLimit"
        ],
    "of-joint": "joint-door-hinge",
    "quantity-kind": "Angle",
    "unit": "RAD",
    "value": 1.6
},
{
```

[34] ../input/object-door-states.json

```
    "@id": "joint-state-door-fully-opened",
    "@type": "JointState",
    "joint": "joint-door-hinge",
    "pose": "joint-pose-door-fully-opened",
    "state": "door-fully-opened"
},
```

With the state definitions, we can now add an initial state to the object instance. This will add a Gazebo plugin[35] to the SDF model so that the door is set up to the correct state at start-up.

```
{
    "@id": "door-instance-2",
    "@type": "ObjectInstance",

    "start-state": "door-fully-opened"
}
```

[35] https://github.com/hbrs-sesame/floorplan-gazebo-initial-state-plugin

4 Solver synthesis and generation: kindyngen

4.1 Introduction

`kindyngen` is a toolchain to transform *composable models*[36] of kinematic chains and queries thereupon into *correct-by-construction* solver code. Example solvers are forward position/velocity/acceleration kinematics or forward/inverse/hybrid dynamics. Additionally, the `kindyngen` toolchain enables to *interleave* those familiar algorithms with additional computations, including control or estimation that reuses the kinematic chain state as the most simple *world model* available to a robot.

Structurally, the `kindyngen` toolchain consists of two parts. First, the `kindynsyn` synthesizer to compute a kinematics or dynamics algorithm given a kinematic chain model. Second, the code generator that transforms the `kindynsyn` output to code. For now the target language for the code generator is C (supported by various software libraries).

4.1.1 Composability

Both, the synthesizer and the code generator are composable. On the one hand, they require the composable building blocks which we call *steps* (for the synthesizer) and *fragments* (for the code generator). On the other hand, a top-level context represents the overall composition. This composition includes the solver *configuration* (for the synthesizer) and an application template (for the code generator) that contains, for example, the `main` function.

4.1.2 Tutorials

In following tutorials we will provide a developer with the basic understanding of the `kindyngen` toolchain. This includes an overview of the terminology as well as the architecture. Afterwards we start with configuring solvers using the built-in functionality of `kindyngen`. Finally, the objective is to demonstrate how a developer can extend the toolchain with custom functionality. For this tutorial we focus on the specification of a Cartesian level controller and how it maps to the robot's joint-level hardware interface.

4.2 Installation

4.2.1 Synthesizer

The following dependencies are required and can either be installed with the operating system's package manager or pip (see below):

- Python[37]
- rdflib[38]
- pySHACL[39]
- numpy[40]

[36] https://github.com/comp-rob2b/modelling-tutorial
[37] https://www.python.org/
[38] https://github.com/RDFLib/rdflib
[39] https://github.com/RDFLib/pySHACL
[40] https://numpy.org/

First, clone the repository and change into the cloned folder:

```
git clone https://github.com/comp-rob2b/kindynsyn.git
cd kindynsyn
```

(Optional) To avoid cluttering the operating system's Python installation it is advisable to use a virtual environment:

```
python3 -m venv .venv
source ./.venv/bin/activate
```

Now, install the synthesizer via pip (the `-e` flag indicates an editable installation to avoid repeated installation steps after changes to the code). This will also install the required dependencies:

```
pip3 install -e .
```

Note: the virtual environment can be deactivated again, at the very end, via:

```
deactivate
```

4.2.2 Code generator

The following dependencies are required for the code generator:

- Java[41]
- Apache Ant[42]
- StringTemplate[43]
- STSTv4[44] (from Git!)
- GNU Make[45]

For convenience, we provide a step-by-step installation guide for the latter two dependencies. Note, that STSTv4 comes with a pre-bundled version of StringTemplate. Hence, the steps for StringTemplate can be considered optional and are only relevant if one plans to use a more recent StringTemplate version.

StringTemplate can either be installed[46] from the pre-compiled version[47] or it can be built from source:

1. Download the latest version and extract it to a directory `<st>`
2. Change to the directory `cd <st>`
3. For version 4.3.3 execute `sed "s/1.6/1.8/g" -i build.xml`
4. Compile using `ant`
5. This creates a JAR file `<jar>` (e.g. `ST-4.3.4.jar`) in the `<st>/dist` folder

[41] https://openjdk.org/
[42] https://ant.apache.org/
[43] https://www.stringtemplate.org/
[44] https://github.com/jsnyders/STSTv4
[45] https://www.gnu.org/software/make/
[46] https://github.com/antlr/stringtemplate4/blob/master/doc/java.md#installation
[47] https://www.stringtemplate.org/download.html

STSTv4 must be built[48] from the Git version (to support nested JSON arrays[49]):

1. Clone the source code: `git clone https://github.com/jsnyders/STSTv4.git`
2. Change into the repository: `cd STSTv4`
3. Build with `ant`
4. Copy the launch script template: `cp stst.sh.init stst.sh`
5. Adapt the `STST_HOME` variable in the launch script
6. (Optional) To use StringTemplate from above adapt the `CP` variable: `sed "s#\$STST_HOME/lib/ST-4.0.8.jar#<st>/dist/<jar>#g" -i stst.sh`
7. Fix the launch script: `sed "s#lib/stst.jar#build/jar/stst.jar#g" -i stst.sh`
8. Make the launch script executable: `chmod +x stst.sh`

4.2.3 Generated code

The following dependencies are required to build and execute the generated code:

- GCC[50]
- CMake[51]
- dyn2b[52]
- robif2b[53]

Make sure that the latter dependencies are accessible to the generated CMake script. That can be achieved via a local or system-wide installation, but also using CMake's package registry (to avoid the installation). To this end, both projects can be configured with a `-DENABLE_PACKAGE_REGISTRY=On` option.

4.3 Background

The kinematics and dynamics of manipulators are among the oldest problems in robotics. Their objective is to answer queries such as "where is a robot's end-effect with respect to the base" or "how much torque to apply to the joints to achieve a desired end-effector motion". Efficient solvers to such queries dispatch computations along the kinematic chain to propagate physical quantities such as positions, velocities, acceleration, forces or inertia in so-called *sweeps* or traversals.

We have recently identified and described common problems in the design of such solvers and their integration in overall robotic software architectures [7]. The figure above shows a typical control loop for a manipulator. It features

- the position (K_p) and velocity (K_v) controllers via the green elements
- a forward position kinematics (FPK), forward velocity kinematics (FVK) and an acceleration-constrained hybrid dynamics (ACHD) solver via the red boxes
- the environment as the robot and its external disturbances via the blue elements

Due to the cyclic nature of the control loop there exists a mutual dependency between the controller and the solver. Therefore, solvers repeat computations: the FVK solver internally solves the FPK problem while the ACHD contains the computations of the FVK and hence also the FPK. Thus, in the above paper we suggest refactoring the solvers so to

[48] https://github.com/jsnyders/STSTv4#install-instructions
[49] https://github.com/jsnyders/STSTv4/commit/6f72c8cc19b773bab015ef9cf58cabd2cb2984c8
[50] https://gcc.gnu.org/
[51] https://cmake.org/
[52] https://github.com/comp-rob2b/dyn2b
[53] https://github.com/rosym-project/robif2b

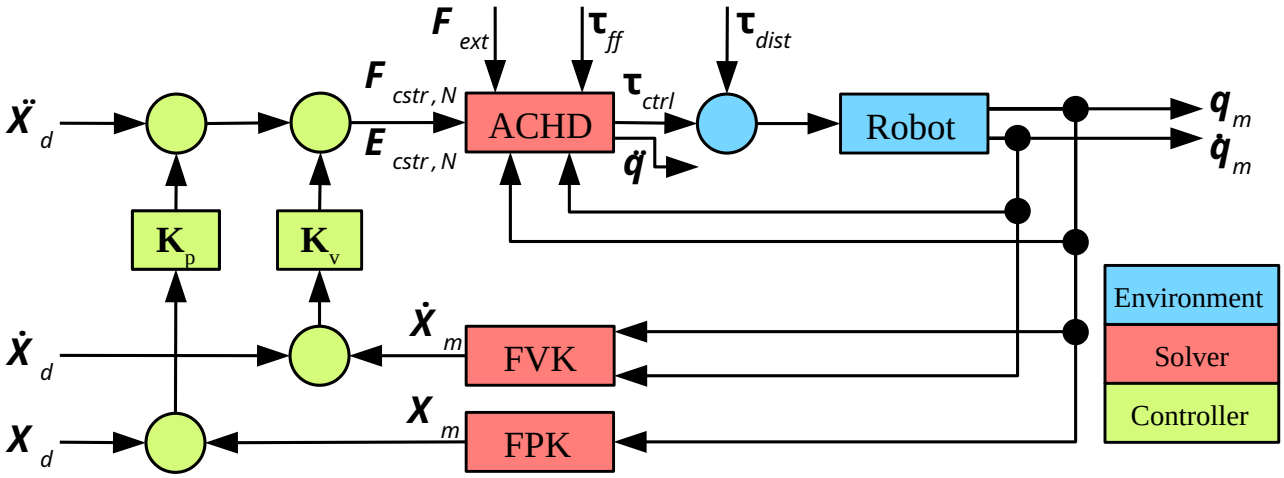

Figure 21: Solver and control loop [7]

- Interleave further computations, such as control or estimation, with the solvers
- Reuse existing state by treating the kinematic chain and its state as the robot's simplest world model
- Propagate the motion drivers (acceleration constraints, external forces and feedforward joint forces) separately and then compose them flexibly during the last outward, *solver* sweep

The figure below demonstrates how the refactored control loop from the previous figure could look like.

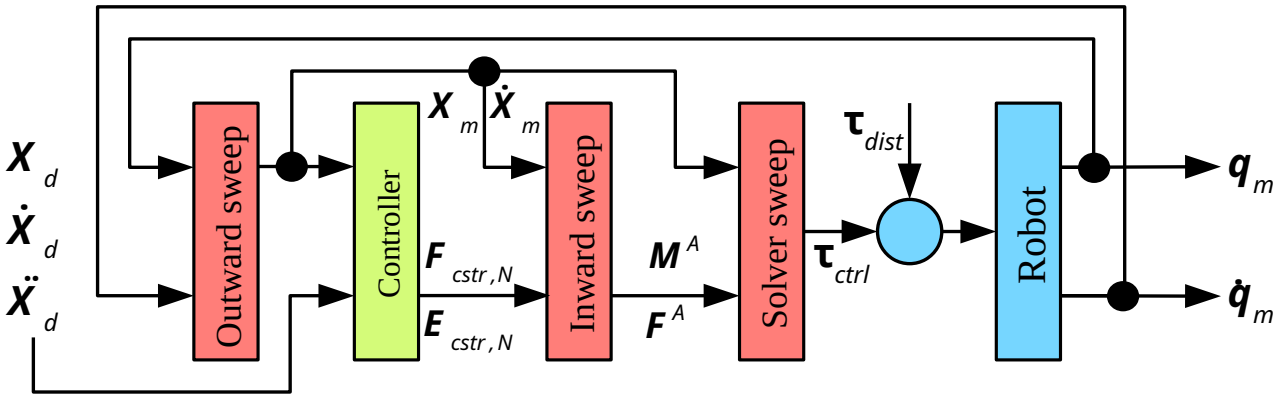

Figure 22: Refactored solver and control loop [7]

For our previous work we had developed an ad-hoc proof-of-concept implementation. Hence, with `kindyngen` we provide a toolchain that allows to generate correct-by-construction applications from models. Here, we focus on the Recursive Newton-Euler Algorithm [8] as the dynamics solver where we separate the propagation of the inertial force from the external force.

4.4 kindyngen's architecture

4.4.1 Synthesizer: kindynsyn

`kindynsyn` is a Python library that relies on RDFlib[54] to represent composable kinematic chain models in memory and perform queries on those graphs. Especially, for non-trivial queries the SPARQL[55] interface provides a powerful and standardized way to interact effectively with the graph. Hence, `kindynsyn` is a graph

[54] https://rdflib.dev/
[55] https://www.w3.org/TR/sparql11-query/

rewriting[56] software, in that it consumes one graph model (of a kinematic chain and a query thereupon) to produce another graph model (the solver algorithm). By exploiting domain-specific knowledge about kinematic chains and their solvers, it can efficiently synthesize the solver algorithms. As output, the synthesizer produces a JSON-based intermediate representation[57] (IR) of the synthesized algorithm.

The core of `kindynsyn` is a graph traversal which can be controlled by extension modules. Inspired by terminology of the Gremlin[58] graph query language, we call these extensions *steps* because they represent the individual "instructions" of the overall graph traversal program. In `kindynsyn` a step declares an expansion query (for example, to define how to find the "root" frame on the next segment in a kinematic chain), or *expander*, to control which nodes and edges the graph traversal should visit next. Additionally, the step associates domain-specific computations (for instance, "map a joint position to a Cartesian pose") pertaining to the nodes and edges with the expander. Note, that steps only *declare* but don't execute the expansion queries. This design originates from our observation that the query execution is on of the slower computations. Hence, `kindynsyn` groups all steps with the same expander to reduce the number of overall query executions. This approach resembles modern web development where React components[59] declare their required data via GraphQL queries[60] with the overall objective of minimizing the slow client-server interaction.

4.4.2 Code generator

A template-based code generator backend consumes `kindynsyn`'s IR. In `kindyngen` we opt for StringTemplate[61] as the template engine and the StringTemplate Standalone Tool[62] as its frontend. Its author has reported on StringTemplate's benefits[63] over the more popular template engines such as Cheetha[64], Django's[65] templating system, Genshi[66] or Jinja[67]. A more hands-on paper draft is also available[68].

To summarize, the main reasons for choosing StringTemplate are:

- It enforces the separation of "business logic" from "display" by only introducing a minimal amount of control flow (if-then-else conditional, template application/map[69] operator, template inclusion) in the templates. This contrasts with the afore-mentioned, popular template engines that often embed Turing-complete languages.
- A template is composed of rules (akin to, or dual to, rules in formal grammars[70]) where each rule has a unique identifier. This aligns with our modelling approach for composable models[71].

However, the benefit of the IR is that it decouples the synthesizer from the code generator. Hence, one could swap over the code generator to a more popular template engine.

[56] https://en.wikipedia.org/wiki/Graph_rewriting
[57] https://en.wikipedia.org/wiki/Intermediate_representation
[58] https://tinkerpop.apache.org/docs/current/reference/#the-graph-process
[59] https://react.dev/reference/react/components
[60] https://graphql.org/
[61] https://www.stringtemplate.org/
[62] https://github.com/jsnyders/STSTv4
[63] Terence John Parr, "Enforcing strict model-view separation in template engines", in Proc. of the International Conference On World Wide Web (WWW), 2004.
[64] https://cheetahtemplate.org/
[65] https://www.djangoproject.com/
[66] https://genshi.edgewall.org/
[67] https://palletsprojects.com/p/jinja/
[68] Terence John Parr, "A Functional Language For Generating Structured Text", available https://www.cs.usfca.edu/~parrt/papers/ST.pdf, 2006.
[69] https://en.wikipedia.org/wiki/Map_(higher-order_function)
[70] https://en.wikipedia.org/wiki/Formal_grammar
[71] https://github.com/comp-rob2b/modelling-tutorial

4.5 Tutorial: Configuring solvers

4.5.1 Forward position kinematics

In the previous section[72] we have shown the common parts of the tutorial to synthesize a solver. Here, we will investigate the variation points (solver configuration and translator) for one of the most simple solvers – the forward position kinematics (FPK) – as implemented in `fpk.py`[73]. It relies on two steps that are already part of the `kindynsyn` library, the `PositionPropagationStep` and the `PositionAccumulationStep`. The former emits all models of data and functions to the graph that are required to compute the pose between the root frames of two adjacent segments. In the example figure below, we see the three relevant pose relations:

- `Pose(of=link1-joint1, wrt=link1-root)`: this pose originates from the kinematic chain model and determines the joint's location on the link.
- `Pose(of=link2-root, wrt=link1-joint)`: this pose is the joint's active contribution and must be computed at runtime given the current joint position as well as the joint model. In other words, it represents the solution to the joint's forward position kinematics. This forward position kinematics operator is the first function emitted by the `PositionPropagationStep`.
- `Pose(of=link2-root, wrt=link1-root)`: this pose is the composition of the previous two poses. Hence, the associated composition operator is the second function emitted by the `PositionPropagationStep`.

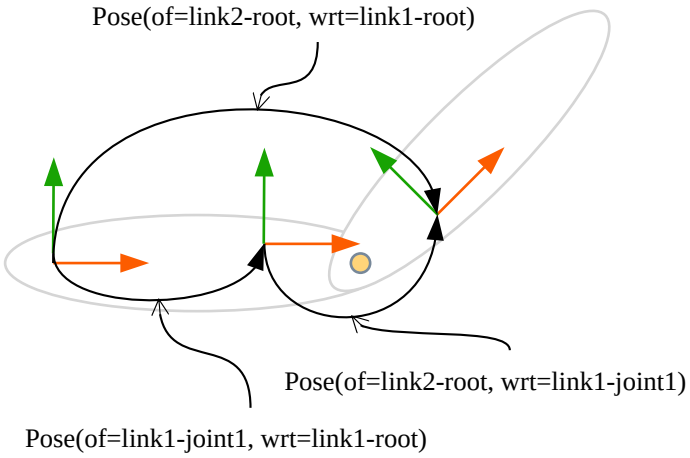

Figure 23: Pose relations associated with two segments

The `PositionAccumulationStep` emits one more composition operator to compute the current segment's pose with respect to the robot's base. For example, related to the figure above and assuming that the robot's base is chosen as the frame `link0-root`, the step would ...

- ... compute `Pose(of=link2-root, wrt=link0-root)` ...
- ... given the `Pose(of=link1-root, wrt=link0-root)`, and ...
- ... given the output from the previous `PositionPropagationStep`

In code this is implemented as follows:

[72] `getting_started.md`
[73] `https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/fpk.py`

```
...
def solver_configurator(g, cache, ROB, slv_algo):
    frm_world = ROB["world-frame"]

    # Instantiate factories
    geom_rel = SpatialRelations(g)
    geom_coord = SpatialRelationsCoordinates(g)
    geom = SpatialRelationsWithCoordinates(geom_rel, geom_coord)
    kc = KinematicChainOperators(g)

    # Instantiate "standard" solver steps
    index = ChainIndexStep(g, cache, frm_world)
    j_mot = JointStep(g, cache, slv_algo)
    pos_prop = PositionPropagationStep(g, cache, slv_algo, geom, geom_coord, kc)
    pos_acc = PositionAccumulationStep(g, cache, slv_algo, geom, geom_coord)

    # Configure solver
    out_1 = SweepConfig(
        direction=SweepDirection.OUTWARD,
        steps=[index, j_mot, pos_prop, pos_acc])

    return SolverConfig(sweeps=[out_1])
```

Apart from the already mentioned `PositionPropagationStep` and `PositionAccumulationStep`, we see two auxiliary steps:

- `ChainIndexStep`: this represents an index or cache of frequently-used entities from the kinematic chain model such as the IDs of the segment's frames or the segment's joint. Later steps can refer to this cache instead of directly interacting with the graph.
- `JointStep`: this represents another cache, but this time for the joint state of the kinematic chain, i.e. it retains the joints' motion (q, $\dot{q}$, $\ddot{q}$) and force variables (τ).

All steps are parameterized with the rdflib graph `g` and the SPARQL `cache`. Additionally, they receive as arguments various factories that contain adaptors to emit data into the rdflib graph. That data can represent spatial relations, their coordinate representation but also operators thereupon.

At the end we compose all steps into a `SweepConfig`, the declarative representation of a "sweep template". We call it a "sweep template", because steps could only be conditionally instantiated at very specific nodes in the graph (this will be discussed in a following tutorial). Here, we only need a single outward sweep with the steps ordered as shown. The order is relevant because of the (data) dependency relations between the steps. For example, the indices or caches must be setup before being accessed by further steps and, similarly, the relative segment-to-segment pose must be known before it can be accumulated with the global root-to-segment pose. The overall solver can consist of multiple sweeps that are composed in the `SolverConfig`.

Since we have only used concepts that are built-in to `kindynsyn`, no additional translators will be required for `IRGen`. Thus, we can simply return an empty list of translators:

```
def translator_configurator():
    return []
```

Now we can execute the synthesizer and code generator as explained before:

```
cd <kindyngen>
python kindynsyn_tutorial/runner.py fpk
cd <kindyngen>/code_generator
make tutorial-dyn2b
cd <kindyngen>/gen
cmake .
make
./main
```

However, this program does neither produce any robotic behaviour nor any visible output because the solver is not connected to any consumers or sinks. We will address this in the next tutorial.

4.5.2 Recursive Newton-Euler inverse dynamics

The tutorial module also features a more complicated, RNE inverse dynamics solver in `rne.py`[74]. We notice how the RNE extends the FPK by adding ...

- ... more steps to first, outward sweep. Those steps effectively compute the forward velocity kinematics and the forward acceleration kinematics.
- ... a second, inward sweep. This sweep propagates inertial forces inwards, from the leafs to the root.

```
def solver_configurator(g, cache, ROB, slv_algo):
    ...
    vel_prop = VelocityPropagationStep(g, cache, slv_algo, geom, geom_coord, kc)
    acc_prop = AccelerationPropagationStep(g, cache, slv_algo, geom, geom_coord,
↪   kc)
    rbi = RigidBodyInertiaStep(g, cache, slv_algo, dyn)
    f_nrt = InertialForceStep(slv_algo, dyn_coord, dyn)
    nrt_prop = QuasiStaticInertialForcePropagationStep(slv_algo, dyn_coord, dyn,
↪   kc, kc_stat)

    out_1 = SweepConfig(
        direction=SweepDirection.OUTWARD,
        steps=[index, j_mot, j_dyn, pos_prop, pos_acc, vel_prop, acc_prop])
    in_1 = SweepConfig(
        direction=SweepDirection.INWARD,
        steps=[rbi, f_nrt, nrt_prop])

    return SolverConfig(sweeps=[out_1, in_1])
```

To build the associated artefacts execute:

```
cd <kindyngen>
python kindynsyn_tutorial/runner.py rne
cd <kindyngen>/code_generator
make tutorial-dyn2b
cd <kindyngen>/gen
cmake .
make
./main
```

Again, the program does not produce any visible output.

[74] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/rne.py

4.6 Tutorial: solver sweep & robot interface

The previous tutorial configured various solvers using only built-in functionality. In this tutorial we compose new functionality onto the RNE solver: a solver sweep that accumulate joint forces from multiple sources (see `my_solver`[75]) and an interface to a robot (see `my_robot_interface`[76]).

4.6.1 Solver sweep

Synthesizer

Vocabulary First, any package should define the relevant graph concepts – the vocabulary – using an rdflib `DefinedNamespace` (see `namespace.py`[77]). The `DefinedNamespace` raises a warning when an "undefined" term is used. Alternatively, rdflib's `Namespace` allows defining concepts on-the-fly, without raising such a warning.

```
class MY_SLV(DefinedNamespace):
    AccumulateJointForce: URIRef

    sources: URIRef
    destination: URIRef

    _NS = Namespace("https://example.org/my-solver#")
```

Here, we require the type `AccumulateJointForce` with imposes two properties `sources` – to represent the list of symbolic pointers to the input joint forces – and `destination` – to represent the symbolic pointer to the accumulated joint force. The vocabulary is identified via the IRI `https://example.org/my-solver#`.

Step & traverser The following code snippet shows an excerpt from `my_solver_steps.py`[78] to define a custom step class.

```
class MySolverStep:
    def __init__(self, g, algo, source_classes):
        self.g = g
        self.algo = algo
        self.source_classes = source_classes

    def traverse(self):
        disp = Dispatcher(condition=None, configure=None,
            compute=self.compute_edge)

        return Traverser(expander=q_expand, edge=[disp])
    ...
```

The class has a constructor (`__init__` function) with the following parameters:

[75] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_solver
[76] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_robot_interface
[77] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_solver/namespace.py
[78] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_solver/my_solver_steps.py

1. `g`: the rdflib graph that contains the kinematic chain model to which the steps can compose custom data blocks or function blocks.
2. `algo`: a dictionary with the two entries `data` and `func`, both of which are lists to pass any instantiate data block or function block back to the synthesizer (cf. Synthesizing a solver algorithm[79]).
3. `source_classes`: a list of all classes from which the solver sweep should get and accumulate the joint forces.

The `traverse` function turns this class into a `kindynsyn` step via duck typing[80]. It first instantiates a `Dispatcher` with takes three arguments. First, a condition which must be a SPARQL query or `None` and must evaluate to `true` for the dispatcher to activate. In kinematics and dynamics solvers, one often must perform special computations at the kinematic chains root or its leaves. This allows, for example, initializing the root's acceleration to gravitational acceleration or initializing the to-be-propagated forces at leave segments. Then the dispatcher declares which functions to call, i.e. where to *dispatch* to. We foresee two types of functions: `configure` and `compute`. The former is meant to declare or allocate any required data blocks, whereas the latter should emit function blocks to perform actual computations at runtime.

Once one or more dispatchers are configured, we can connect them to (i) a graph expansion query; and (ii) either the nodes of this expansion or the edges (only the edge version is shown above) using a `Traverser` instance. The node vs. edge difference is relevant for kinematics and dynamics solvers because some computations pertain to frame entities (the nodes) whereas other pertain to the relation between frames (the edges). An example of the former is the computation of inertial forces from known accelerations. An example of the latter is the propagation of physical quantities such as positions, velocities or accelerations from one frame to another.

In the above example we reuse the `q_expand` query to expand the graph. It is defined as follows:

```
PREFIX geom-ent:
 ↪  <https://comp-rob2b.github.io/metamodels/geometry/structural-entities#>
PREFIX kc-ent:
 ↪  <https://comp-rob2b.github.io/metamodels/kinematic-chain/structural-entities#>

SELECT ?child ?parent WHERE {
    ?node ^geom-ent:simplices / geom-ent:simplices ?joint_prox .
    FILTER (?node != ?joint_prox)
    ?joint_prox ^kc-ent:between-attachments / kc-ent:between-attachments ?child
 ↪  .
    FILTER (?joint_prox != ?child)
    BIND(?node as ?parent)
}
```

First, the query defines two namespaces to allow for shorter notation (the `PREFIX` lines) and then starts the query definition via the `SELECT ... WHERE` statement that declares two return arguments named `?child` and `?parent`. We explain how the core query works along the following figure. The query is applied to a `?node` whose ID `kyndynsyn` passes to the query (in the example figure this would be the frame `l1-fr1`). Then the query follows any edge `geom-ent:simplices` in the opposite direction (indicated by the caret character `^`) to end up at a simplicial complex – the most simple version of a rigid body or link – which would be `l1` in the figure. From there it follows the `geom-ent:simplices` edge to end up at `l1-fr1`, `l1-fr2` *and* `l1-fr3`. The first `FILTER` statement removes the start node from this set, so that the next line tries to follow any sequence of `kc-ent:between-attachments`, first in the inverse direction then in the forward direction, from the nodes `l1-fr2` and `l1-fr3`. This query is only applicable to `l1-fr3` but leads

[79] https://github.com/comp-rob2b/docs/getting_started.md#synthesizing-a-solver-algorithm

[80] https://en.wikipedia.org/wiki/Duck_typing

to the nodes `l1-fr3` and `l2-fr1`. Hence, the second `FILTER` statement removes the previously-visited node `l1-fr3` from the set so that only `l2-fr1` remains as the `?child`. Finally, the query assigns (`BIND`) the `?node` parameter to the `?parent` parameter which is an expected output.

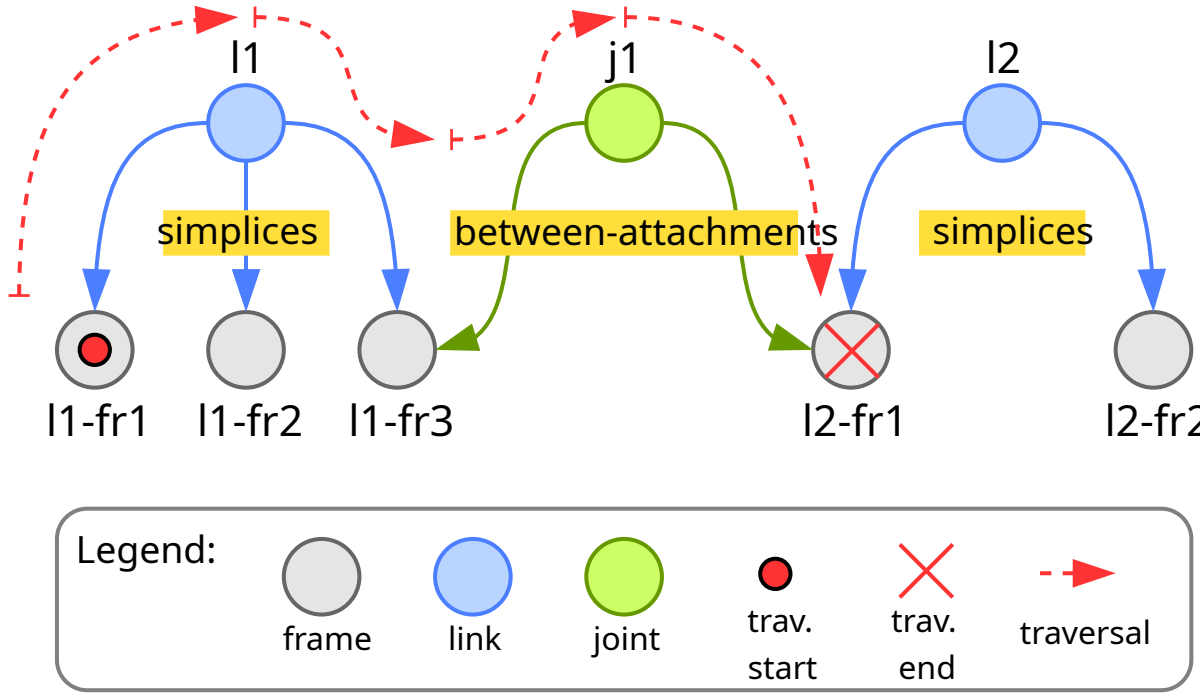

Figure 24: Expansion query

Whenever this expansion query produces any non-empty output, i.e. it finds a node to expand to, `kindynsyn` will trigger the dispatchers as defined in the associated `Traverser`.

Computation: emitting closures Now we look into the `compute_edge` function for the joint force accumulator:

```
    def compute_edge(self, state, parent, child):
        sources = []
        for source_class in self.source_classes:
            sources.append(state[child][source_class].tau)

        jnt = state[child][JointState]
        acc = self.accumulate_joint_force(sources, jnt.tau)

        self.algo["func"].extend([acc])
```

The first parameter is a state dictionary that, for each node, can contain structural state of the kinematic chain or computational state associated with the synthesized algorithm. The state parameter has a two-fold role. On the one hand, within a step, it can transfer data ("variable definitions") from the configuration to the computation. On the other hand, it can transfer data between different steps, as is the case in this example.

In the expansion query above we saw that we only visit frames, but here we want to perform a computation on a joint which lies *between* two frames. This is the reason for dispatching to an edge handler and also why there are a `parent` and `child` parameter, both of which contain the graph ID (`URIRef`) of the respective node. Note, that depending on the direction of the sweep that the steps should be invoked in, the number of children

differs. For an *outward* sweep each child is visited separately and hence the function takes a single parent as well as a single child (cf. example above). Instead, for an *inward* sweep the parent is visited and hence accepts a single parent as well as a list of children.

In the beginning, the function iterates over all defined states' `source_classes`. It collects the joint torque (`tau`) for each of those classes from the state associated with the child. Next, it obtains a handle to the `JointState` for the child, which will be used as the destination to emit the accumulated joint forces to as realized by the following `accumulate_joint_force` function (see below). Finally, the accumulation operator is appended to the list of function that form the overall synthesized algorithm.

`accumulate_joint_force` is an auxiliary function to emit data (here, a model of the accumulation operator) into the graph:

```
    def accumulate_joint_force(self, sources, destination):
        lst = BNode()
        collection.Collection(self.g, lst, sources)

        id_ = uuid_ref()
        self.g.add((id_, RDF["type"], MY_SLV["AccumulateJointForce"]))
        self.g.add((id_, MY_SLV["sources"], lst))
        self.g.add((id_, MY_SLV["destination"], destination))
        return id_
```

To this end, the function transforms the list of sources (from where joint forces should be accumulated) to the rdflib list representation. Then it creates a new identifier, a Universally Unique Identifier[81] (UUID), for the operator and associates it with its `type`, the `sources` and `destination` as defined in the vocabulary.

IR generator The next objective is to implement a custom translator from the graph to the IR. The code is available in the `my_solver_gen.py`[82] module:

```
class AccumulateJointForceTranslator:
    @staticmethod
    def is_applicable(g, node):
        return set([MY_SLV["AccumulateJointForce"]]) <= set(g[node :
        ↪  RDF["type"]])

    @staticmethod
    def translate(g, node):
        l = list(collection.Collection(g, g.value(node, MY_SLV["sources"])))

        sources = list(map(lambda v: escape(qname(g, v)), l))
        destintation = escape(qname(g, g.value(node, MY_SLV["destination"])))

        return {
            "represents": str(node),
            "name": escape(qname(g, node)),
            "operator": "ex-accumulate-joint-force",
            "sources": sources,
            "destination": destintation
        }
```

[81] https://en.wikipedia.org/wiki/Universally_unique_identifier

[82] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_solver/my_solver_gen.py

Each translator must provide two methods (either as true method or static method[83]): `is_applicable` and `translate`. Both methods take two arguments, the overall rdflib graph `g` from which to obtain data and the current (candidate) `node` to be translated. The former method returns a boolean result to indicate whether the translator can handle this node (e.g. by inspecting its type or other properties). The latter method performs the actual translation. Here, the translation entails the creation of a Python dictionary that can straightforwardly be mapped to JSON. Hence, we transform the list representation and shorten (via the `qname` function) as well as escape identifiers (e.g. `http://` would not be a valid variable name in C or Python).

Code generator The code generator fragment (`my_solver.stg`[84]) for the joint force accumulator is extremely simple and consists of the following three lines:

```
ex-accumulate-joint-force(args) ::= <<
<args.destination>[0] = <args.sources:{s | <s>[0]}; separator=" + ">
>>
```

The first line defines the rule with a single (composite) argument `args`. Note that rule's name must match the `operator` name in the IR. << and >> are the rule delimiters. The second line determines the actual code that should be emitted. < and > represent instructions specific to StringTemplate. In particular does `<args.destination>` access the `destination` field in the `args` object and emit the associated value. The `sources` field is actually a list. Hence, we want to apply a sub-template to each entry of that list which StringTemplate realizes with the map operator `:`. In the following braces follows an anonymous template rule with a single argument `s` which just emits that argument followd by the string `[0]`, i.e. an array index operation in C. Finally, we want to add all entries in the array which we achieve by separating them with the string +.

As a concrete example, the following JSON object ...

```
{
    "operator": "ex-accumulate-joint-force",
    "sources": [ "a", "b" ],
    "destination": "c"
}
```

... would result in `c[0] = a[0] + b[0]`.

4.6.2 Robot interface

The robot interface[85] does not introduce new concepts beyond those from the above solver sweep. To this end, we keep the description in this tutorial brief. The module introduces two new steps, one for reading positions or velocities from the robot and the other to write control torques to the robot. Additionally, it provides the associated vocabulary and the IR generator backend.

However, we introduce two code generators. The first one (cf. `robif_print.stg`[86]) sets static values for the joint position and velocity measurements (not shown here), but prints the control torques to the terminal via the following rule:

[83] https://docs.python.org/3/library/functions.html#staticmethod

[84] https://github.com/comp-rob2b/kindyngen/models/templates/fragments/my_solver.stg

[85] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_robot_interface

[86] https://github.com/comp-rob2b/kindyngen/models/templates/fragments/robif_print.stg

```
...
joint-force-from-solver(args) ::= <<
<print-variable(args.source, variables)>
>>
```

The second code generator (cf. robif2b.stg[87]), on the contrary, implements a proper interface to a robot using the robif2b library. Here, the control torque interface defines the following two rules that establish the data exchange between the solver and the library:

```
robif2b-variables() ::= <<
double eff_cmd[] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 };
...
struct robif2b_kinova_gen3_nbx rob = {
    ...
    .jnt_trq_msr = eff_msr,
    ...
};
>>
...

joint-force-from-solver(args) ::= <<
eff_cmd[<args.destination-index>] = <args.source>[0]
>>
```

> ***NOTE:***
> The robif2b code generator fragment hardcodes the connection details and user credentials for now and may require adaptations for the real robot at hand. A more advanced version should provide those parameters via explicit models.

4.6.3 Application template

The application context, i.e. the top-level code generator rule, for printing the resulting torques to the terminal is located in tutorial_dyn2b_slv_print.stg[88] and defines the main function as follows:

```
import "../models/templates/fragments/algorithm_c.stg"
import "../models/templates/fragments/dyn2b.stg"
import "../models/templates/fragments/my_solver.stg"
import "../models/templates/fragments/robif_print.stg"
import "../models/templates/fragments/print.stg"


application(data-types, variables, input, output, local, closures, schedule) ::=
 ↪  <<
<dyn2b-include()>
<robif-print-include()>
<print-include()>

int main()
```

[87] https://github.com/comp-rob2b/kindyngen/models/templates/fragments/robif2b.stg

[88] https://github.com/comp-rob2b/kindyngen/models/templates/applications/tutorial_dyn2b_slv_print.stg

```
{
    <local:{id | <define-variable(id, variables.(id))>}:stmt(); separator="\n">

    <schedule:schedule(closures):stmt(); separator="\n">

    return 0;
}
>>
```

We see StringTemplate `import` instructions to load the fragments (e.g. the `my_solver.stg` and the `robif_print.stg` as described in this tutorial). `application` is the name of the generator rule with the listed arguments. Next, some fragments require include directives for C's pre-processor. Finally, the key lines in the C main function are a definition of all required variables that are listed in the rule's `local` argument and the emission of the schedule, i.e. all instructions that represent the overall solver, by applying the `schedule` rule to the `schedule` argument.

4.6.4 Build and execute

To build the associated artefacts execute:

```
cd <kindyngen>
python kindynsyn_tutorial/runner.py rne_slv_robif
cd <kindyngen>/code_generator
```

Now there are two code generator targets available. The first one prints the resulting torques to the terminal . . .

```
make tutorial-dyn2b-slv-print
```

. . . whereas the second one interfaces with a real robot using the robif2b library:

```
make tutorial-dyn2b-slv-robif2b
```

Upon either choice compile the code and execute the resulting software as before:

```
cd <kindyngen>/gen
cmake .
make
./main
```

The program should either print some values to the terminal or execute a plain gravity compensation behaviour on a connected, real robot.

4.7 Cartesian control

Most users of a robot want to describe tasks in Cartesian space. Hence, in this tutorial we will implement a simple, linear impedance controller[89] to regulate the artificial force to be exerted on the robot's end-effector. Here, we connect this controller to the robot's end-effector via the RNE's external force interface. The objective is to apply a torque on the end-effector so that it should rotate when not constrained.

[89] An impedance controller maps from the motion domain to the force domain. The impedance relations are known as *stiffness* (position to force), *damping* (velocity to force) and *inertia* (acceleration to force).

4.7.1 Controller design

The controller should realize a desired (maximum) end-effector velocity while simultaneously limiting the force. We achieve this by a constant damping controller where the damping D is calculated from a maximum force (f_{max}) and a maximum velocity (v_{max}). The commanded force (f_{cmd}) is computed given the measured velocity (v_{msr}) via the following formula:

$$f_{cmd} = f_{max} + D \cdot v_{msr} = f_{max} - \frac{f_{max}}{v_{max}} v_{msr}$$

The resulting behaviour is best understood by first considering the two "extreme" cases and then the general case:

- $v_{msr} = 0$: the controller commands the maximum permissible force. If the system is not in a contact situation it will accelerate, otherwise it will only exert the maximum force onto the environment.
- $v_{msr} = v_{max}$: the controller commands no force and, hence, does not accelerate the system. Most likely external effects such as friction will brake the system.
- Otherwise: the controller commands a force that is proportional to the difference from the maximum velocity, offset by the maximum permissible force. The damping represents the proportionality factor.

The following figure depicts this behaviour.

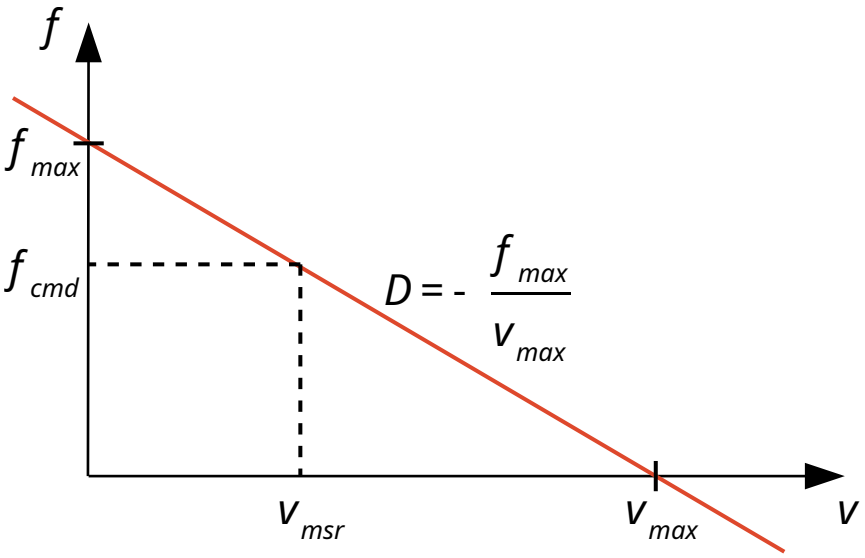

Figure 25: Impedance controller

4.7.2 Traverser: mounting the controller in the graph

The synthesizer steps live in the `my_contoller_steps.py`[90] module. Here, the traverser is similar to the ones from the previous tutorial, but it (i) relies on a custom expansion query; and (ii) employs an additional configuration method:

[90] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/my_contoller/my_controller_steps.py

```
ctrl_expand = """
PREFIX ex-ctrl: <https://example.org/ctrl#>

SELECT ?child ?parent WHERE {
    ?node ^ex-ctrl:attached-to ?child .
    BIND(?node as ?parent)
}
"""

...

def traverse(self):
    return Traverser(
        expander=ctrl_expand,
        edge=[Dispatcher(None, self.configure_ede, self.compute_edge)]
    )
```

A controller is assumed to point to a frame via the `ex-ctrl:attached-to` property. Hence, the expansion query tries to find such a controller by following that property in the opposite direction (`^`).

4.7.3 Configuration: declaring and caching data

Previously we have already encountered the `state` parameter of the configuration and compute functions that relied on built-in state representations. For the controller we want to introduce a custom state to capture the controller's output. We achieve this by adding a dataclass[91] with the single field `wrench` (of type `URIRef` or `None`) that has a default value of `None`:

```
@dataclass
class MyCartesianControllerState:
    wrench: URIRef | None = field(default=None)
```

We connect the controller's output (f_{cmd}) to the external force exerted on the end-effector. To this end, during the configuration, we first declare the wrench data. Then we tag it as an external force specification, so that the RNE's external force propagation can find it:

```
def configure_edge(self, state, parent, child):
    par = state[parent][ChainIndexState]

    wrench = self.dyn.wrench(acts_on=par.bdy, as_seen_by=par.frm_prox,
        number_of_wrenches=1)

    ext = uuid_ref()
    self.g.add((ext, RDF["type"], SPEC["ExternalForce"]))
    self.g.add((ext, SPEC["force"], wrench))
    ...
```

Note, that (i) the wrench is expressed locally to the link (`as_seen_by=par.frm_prox`); and (ii) only consists of a single number of wrench instances (`number_of_wrenches=1`).

Next, we cache that data so that it is efficiently accessible during the computation:

[91] https://docs.python.org/3/library/dataclasses.html

```
    ...
    s = MyCartesianControllerState()
    s.wrench = wrench

    state[child][MyCartesianControllerState] = s
    ...
```

Finally, don't forget to register the wrench data with the algorithm so that the according data blocks can be generated in the algorithm representation:

```
    ...
    self.algo["data"].extend([wrench])
```

4.7.4 Computation: emitting closures

The computation introduces no new concepts beyond the previous tutorial and is only listed for completeness:

```
def compute_edge(self, state, parent, child):
    vel = state[parent][VelocityPropagationState]
    ctrl = state[child][MyCartesianControllerState]

    damp = self.my_damper(
        velocity_twist=vel.xd_tot,
        wrench=ctrl.wrench)

    self.algo["func"].extend([damp])

def my_damper(self, velocity_twist, wrench):
    id_ = uuid_ref()
    self.g.add((id_, RDF["type"], EX_CTRL["Damping"]))
    self.g.add((id_, EX_CTRL["max-velocity"], Literal(0.1)))
    self.g.add((id_, EX_CTRL["max-force"], Literal(2.0)))
    self.g.add((id_, EX_CTRL["velocity-twist"], velocity_twist))
    self.g.add((id_, EX_CTRL["wrench"], wrench))
    return id_
```

NOTE:
In the interest of keeping the tutorial more concise and focused on the software, we have opted for a (too) simplistic controller model and synthesizer implementation. In particular, the `max-velocity` and `max-force` properties are just configured numbers and hence lack ...

- ... a value per degree of freedom
- ... a coordinate-free representation
- ... physical units and a reference to a coordinate frame
- ... semantic checks (because of the previous two points)
- ... reification as a runtime-accessible data block.

The modules in the `kindynsyn` package contain more complete modules that try to avoid such issues.

4.7.5 Solver and translator configurator

The synthesizer that includes the controller (see `rne_slv_robif_ctrl.py`[92]) is an extension with respect to the previous tutorial. First, we require an additional propagation step of the external forces and need to include those external forces in the joint force accumulation:

```
...
ext_prop = QuasiStaticExternalForcePropagationStep(g, cache, slv_algo,
    dyn_coord, dyn, kc, kc_stat)

my_slv = MySolverStep(g, slv_algo, [
    QuasiStaticInertialForcePropagationState,
    QuasiStaticExternalForcePropagationState
])
...
```

Notice that this accumulation was the main reason for introducing this custom solver step in the previous tutorial.

Next, we can attach a controller to the end-effector using the `attached-to` property and instantiate the controller step:

```
...
g.add((ROB["my-ctrl"], EX_CTRL["attached-to"], ROB["link7-root"]))
my_ctrl = MyCartesianControllerStep(g, slv_algo, dyn)
```

Then, accomodate for the two new steps in the sweep configuration by appending … 1. … the controller to the first outward sweep 2. … the external force propagation to the inward sweep

```
out_1 = SweepConfig(
    direction=SweepDirection.OUTWARD,
    steps=[..., my_ctrl])
in_1 = SweepConfig(
    direction=SweepDirection.INWARD,
    steps=[..., ext_prop])
...
```

Finally, don't forget to append a translator instance to the return value of the `translator_configurator` function:

```
def translator_configurator():
    return [
        ...
        MyCartesianControllerTranslator()
    ]
```

[92] https://github.com/comp-rob2b/kindyngen/kindynsyn_tutorial/rne_slv_robif_ctrl.py

4.7.6 Code generator

The code generator template for the `ex-damping` type controller lives in the `my_controller.stg`[93] fragment and simply emits a function call:

```
...
ex-damping(args) ::= <<
my_controller(<args.max-velocity>, <args.max-force>, <args.velocity-twist>,
 ↪  <args.wrench>)
>>
```

Additionally, the fragment contributes the implementation of the controller as discussed before but only applies that controller to the "angular-z" degree of freedom:

```
...
controller-definition() ::= <<
void my_controller(
        double max_velocity,
        double max_force,
        const double *restrict velocity,
        double *restrict force)
{
    const double DAMPING = -max_force / max_velocity;

    for (int i = 0; i \< 3; i++) {
        force[DYN2B_WRENCH3_LIN_OFFSET + i] = 0.0;
        force[DYN2B_WRENCH3_ANG_OFFSET + i] = 0.0;
    }

    double v = velocity[DYN2B_WRENCH3_ANG_OFFSET + DYN2B_Z_OFFSET];
    force[DYN2B_WRENCH3_ANG_OFFSET + DYN2B_Z_OFFSET] = max_force + DAMPING * v;
}
>>
```

4.7.7 Build and execute

To build the associated artefacts execute:

```
cd <kindyngen>
python kindynsyn_tutorial/runner.py rne_slv_robif_ctrl
cd <kindyngen>/code_generator
```

Now there are two code generator targets available. The first one prints the resulting torques to the terminal . . .

```
make tutorial-dyn2b-slv-print-ctrl
```

. . . whereas the second one interfaces with a real robot using the robif2b library:

[93] https://github.com/comp-rob2b/kindyngen/models/fragments/my_controller.stg

```
make tutorial-dyn2b-slv-robif2b-ctrl
```

Upon either choice compile the code and execute the resulting software as before:

```
cd <kindyngen>/gen
cmake .
make
./main
```

5 Model-Based BDD for Robotics

`bdd-dsl`[94] is one of the tools within the ExSce Workbench ecosystem developed to support the specification and execution of acceptance tests for robotic applications using the Behaviour-Driven Development (BDD)[1] methodology.

To this end, we employ the JSON-LD schema[95] to define the BDD concepts as described in the original article[1], concepts necessary for specifying robotic scenarios, and the relations needed to compose them into concrete test scenarios. These concepts and relations are based on our analysis of rulebooks from several robotic benchmarks and competitions, the results of which is reported in [6].

5.1 Metamodels for Specifying BDD Scenarios for Robotic Applications

5.1.1 Metamodel Design

Quick Review of Behaviour-Driven Development (BDD) Requirements in Test-Driven Development (TDD) are often represented using *User Stories*, typically in the following format:

```
As a [Stakeholder role]
I want [Feature]
So that [Benefit]
```

In his original blog post [1] introducing BDD, North proposed to represent acceptance criteria for each user story as a list of scenarios capturing the expected behaviours of the system, each using the following formulation:

```
1 Given [Precondition]
2 When [Event/Action]
3 Then [Expected Outcome]
```

This formulation extends user stories with *narratives*, each consist of initial condition(s), events or actions that signal the start of the behaviour, and the criteria that characterizes what constitute a successful behaviour. We consider the `Given-When-Then` formulation a good metamodel because of its simple concepts, which are both easy to understand and flexible to different interpretations and extensions. Furthermore, BDD approaches, e.g. with the popular Gherkin syntax[96], has wide support in the software engineering community for test automation in many program languages and frameworks. This can reduce the effort in generating executable implementations for verifying BDD acceptance criteria in the future.

Specifying Robotic Scenarios A challenge of applying BDD to any complex domains is to deal with the myriad variations that may exist for the same scenario. Alferez et al.[2] proposed to define scenario templates for common system behaviours, e.g. in their use case for operations on different data structures. Concrete BDD scenarios can then be generated from these templates depending on the particular data object being tested.

We aim to apply a similar idea for representing robotic scenarios, whose variability dimensions can be vastly more numerous and complex compared to the application investigated by Alferez et al.[2]. To this end, we analysed rulebooks from several robotic benchmarks and competitions[6] to identify common elements used

[94] https://hbrs-sesame.github.io/bdd-dsl/
[95] https://json-ld.org/
[96] https://cucumber.io/docs/gherkin/reference/

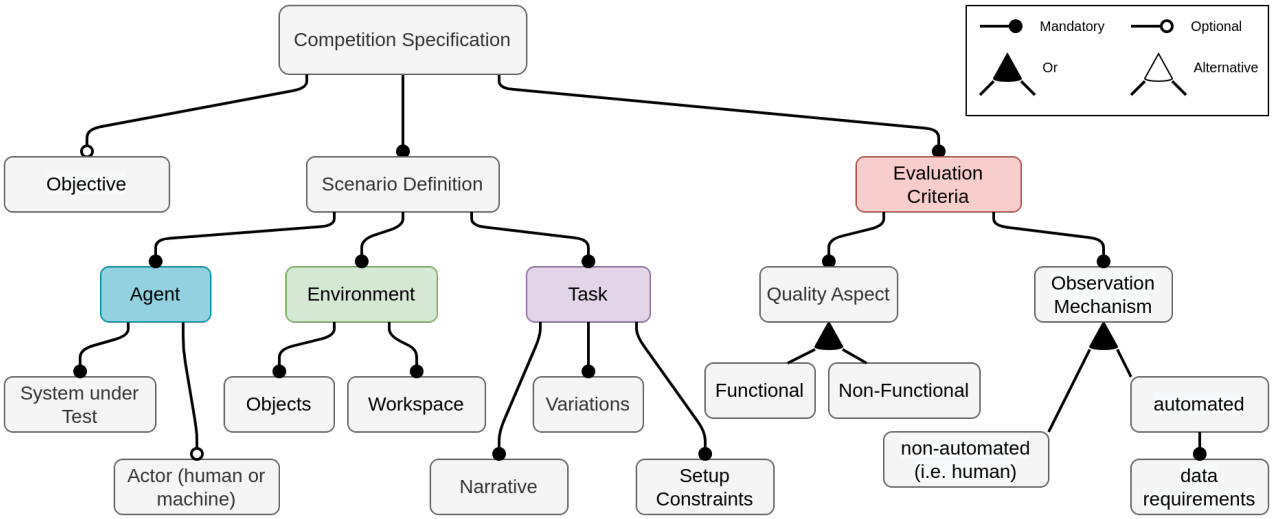

Figure 26: A Feature Model of Robotic Competitions

to describe test scenarios at these events. We consolidate our findings is a Feature Model[97], shown in Figure 26. This serves as the basis for the metamodels described below.

5.1.2 Metamodel Description

We choose to represent our metamodels and models for specifying BDD scenarios with the JSON-LD Schema[98]. The metamodels described below can be found in the following files:

- agent.json[99]: Metamodel for specifying agents in a scenario
- environment.json[100]: Metamodel for specifying elements in the environment of a robotic scenario
- task.json[101]: Metamodel for specifying task-related concepts and relations in a robotic scenario
- event.json[102]: Metamodel for specifying event-driven coordination of robot behaviours
- bdd.json[103]: Metamodel for specifying BDD templates and their variants

For an overview of main JSON-LD keywords used in our models, please take a look at our modelling tutorial[104]. More details on this standard can be found on the official online documentation[105]. For brevity, we use compact IRIs[106] (i.e. use `:` to separate prefix and suffix) when referring to metamodels concepts and relations below. Prefixes in this section:

- `agn`: `https://hbrs-sesame.github.io/metamodels/agent#`
- `env`: `https://hbrs-sesame.github.io/metamodels/environment#`
- `task`: `https://hbrs-sesame.github.io/metamodels/task#`
- `evt`: `https://hbrs-sesame.github.io/metamodels/coordination/event#`
- `bdd`: `https://hbrs-sesame.github.io/metamodels/acceptance-criteria/bdd#`

[97] https://en.wikipedia.org/wiki/Feature_model
[98] https://json-ld.org/
[99] https://hbrs-sesame.github.io/metamodels/agent.json
[100] https://hbrs-sesame.github.io/metamodels/environment.json
[101] https://hbrs-sesame.github.io/metamodels/task.json
[102] https://hbrs-sesame.github.io/metamodels/coordination/event.json
[103] https://hbrs-sesame.github.io/metamodels/acceptance-criteria/bdd.json
[104] https://github.com/comp-rob2b/modelling-tutorial#json-ld
[105] https://www.w3.org/TR/json-ld/
[106] https://www.w3.org/TR/json-ld/#compact-iris

Agent

- `agn:Agent`: we adopt the definition from the IEEE Standard Ontologies for Robotics and Automation[9] (cf. prov:Agent[107]): "Something or someone that can act on its own and produce changes in the world."
- `agn:of-agent`: composition relation with a `agn:Agent` instance.
- `agn:has-agent`: aggregation relation with a `agn:Agent` instance.

Environment

- `env:Object`: Physical objects in the environment which an agent may interact with. Note that instances of `env:Object` are not limited to objects that the agent can move around like bottles or cups, but can also include typically stationary objects like tables or sofas.
- `env:Workspace`: Abstract space in which an agent may operate. Instances can be areas surrounding objects like tables or kitchen counters, or rooms in a flat. A `env:Workspace` instance may contain other instances, e.g. a living room can contain a workspace surrounding the coffee table.
- `env:has-object`, `env:has-workspace` represents aggregation relation to objects and workspaces.
- `env:of-object` represents composition relation to an object, e.g. a property of an object.

Task

- `task:Variation`: possible variations of a scenario. A `task:Variation` instance can be associated with a `bdd:ScenarioVariable` via the `bdd:of-variable` relation, in which case the instance denotes possible variations of the variable.
- `task:can-be`: represents an aggregation relation from a `task:Variation` instance to the possible entities of the variation.

Coordination

- `evt:Event`: a time instant, conforms with time:Instant[108] from the Time Ontology in OWL (cf. prov:InstantaneousEvent[109]).
- `evt:has-event`: aggregation relation with a `evt:Event` instance.

BDD Scenario Templates and Variants

- `bdd:Scenario`: represents a BDD scenario.
- `bdd:of-scenario`: composition relation to `bdd:Scenario`
- `bdd:GivenClause`, `bdd:WhenClause`, `bdd:ThenClause`: represents the three basic elements of a BDD formulation.
- `bdd:given`, `bdd:when`, `bdd:then`: composition relations that link a `bdd:Scenario` to *exactly one* instance of `bdd:GivenClause`, `bdd:WhenClause`, `bdd:ThenClause`, correspondingly.
- `bdd:WhenEvent`: associates a `bdd:WhenClause` and a `evt:Event` instances using the `bdd:of-clause` and `evt:has-event` relations, respectively. Note that different interpretation exists that explain the semantics of the *When* clause in BDD. In the current iteration of `bdd-dsl`, we choose to simply associate *When* with `evt:Event` instances to denote the start of the behaviour being tested.
- `bdd:ScenarioVariant`: aggregate instances of `task:Variation` with relation `bdd:has-variation`; has a composition relation `bdd:of-scenario` with a `bdd:Scenario` instance.

[107] https://www.w3.org/TR/prov-o/#Agent
[108] https://www.w3.org/TR/owl-time/#time:Instant
[109] https://www.w3.org/TR/prov-o/#InstantaneousEvent

- `bdd:UserStory`: aggregate instances `bdd:ScenarioVariant` with relation `bdd:has-criteria`.
- `bdd:ScenarioVariable`: represents points of variation for a scenario.
- `bdd:of-variable`: composition relation to a `bdd:ScenarioVariable`.
- `bdd:IsHeldPredicate`, `bdd:IsNearPredicate`: domain-specific predicates relevant to a pickup task.
- `bdd:TimeConstraint`: constraint on when the predicate of `bdd:FluentClause` must hold.
- `bdd:FluentClause`: represents a BDD clause as a [fluent(https://en.wikipedia.org/wiki/Fluent_(artificial_intelligence))], i.e. a condition evaluated at a point in time. A fluent clause has aggregation relations with one predicate, e.g. a `bdd:IsHeldPredicate` instance, and one `bdd:TimeConstraint` instance. The relations are `bdd:predicate` and `bdd:time-constraint`, respectively.
- `bdd:clause-of`: aggregation relation between `bdd:FluentClause` instances and instance of `bdd:GivenClause` and `bdd:ThenClause`. This relation allows for extending a BDD scenario template with any number of clauses.
- `bdd:ref-object`, `bdd:ref-workspace`, `bdd:ref-agent`: A `bdd:FluentClause` instance can associate with a `bdd:ScenarioVariable` instance via these relations, which constrain the semantic of the variable in the context of the `bdd:FluentClause` instance.

5.2 Tutorial: Modelling and Generating Gherkin features for A Simple Pickup task

This tutorial will first showcase how concepts from our metamodels[110] can be used to model acceptance criteria of a simple pickup task as Behaviour-Driven Development (BDD) scenarios, as well as how variations of such scenarios can be introduced in the model. An example is then presented to show how to transform this model into a Gherkin feature for integration with appropriate BDD toolchains, e.g. `behave`[111] for the Python language.

5.3 Example: BDD Scenario for a Robotic Pickup Task

Consider a simple robotic pickup task, where a robot must pick an object from a surface. A typical BDD scenario for such a task, when realized in the Gherkin format, may look something as follows:

```
Scenario: pickup scenario
  Given an object is located on the table
  When the robot starts picking
  Then the object is held by the robot
```

In robotics, even such a simple task can vary in many dimensions: the objects to be picked up, the operational space in which the task take place, the robotic systems that carry out the task, or the mechanisms available for verifying the different BDD clauses. While some mechanisms are available in Gherkin to deal with variations of scenarios, applying existing BDD approaches like Gherkin directly to robotic scenarios remains challenging. These challenges are discussed in more details in our workshop paper [6]. The rest of this tutorial will present the process of composing a scenario template for such a pickup task, as well as how to introduce variations to the template for concrete scenarios. Finally, we will show how Gherkin feature files similar to the snippet above can be generated from the scenario variant model using our library.

[110] https://hbrs-sesame.github.io/bdd-dsl/docs/bdd-concepts.md
[111] https://behave.readthedocs.io

5.3.1 Specifying BDD Acceptance Criteria for A Pickup Task

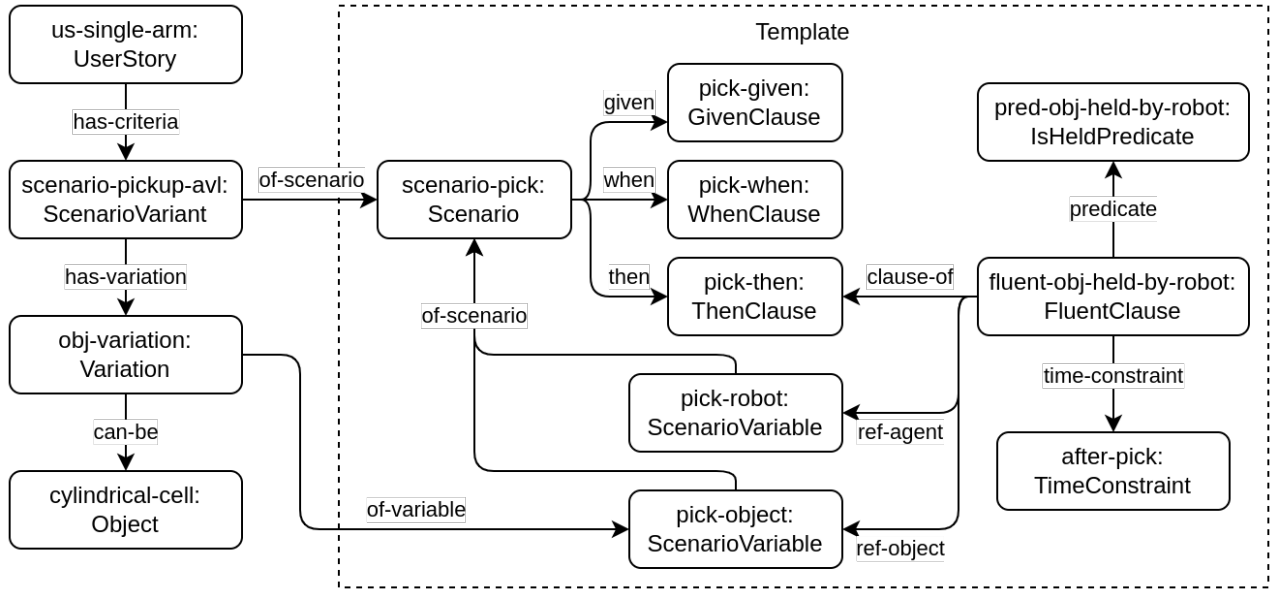

Figure 27: Partial example of a BDD scenario template and variant for the pickup task.

As mentioned in the description of our metamodels in Section 5.1, our models are graphs represented using the JSON-LD schema. Figure 27 shows part of such a graph, which consists of a BDD scenario template and corresponding variants for a simple pickup task. The motivation for this template-variant design is discussed in more details Section 5.1.1. Complete JSON-LD models for the scenario template[112] and variant[113], along with their generated visualization[114] are publicly available for download. The rest of this section will walk through the process of composing these models from our metamodels.

Specifying Scenario Templates First, we define the skeleton of the BDD scenario for the pickup task: a `bdd:Scenario` having composition relation to exactly one instance of `bdd:GivenClause`, `bdd:WhenClause`, and `bdd:ThenClause`.

```
{ "@id": "pick-given", "@type": "bdd:GivenClause" },
{ "@id": "pick-when", "@type": "bdd:WhenClause" },
{ "@id": "pick-then", "@type": "bdd:ThenClause" },
{
  "@id": "scenario-pick", "@type": "bdd:Scenario",
  "bdd:given": "pick-given",
  "bdd:when": "pick-when",
  "bdd:then": "pick-then"
}
```

Next, we define `bdd:ScenarioVariable` instances, which are points of variation of the scenario template. Here, we may change the object, workspace, and agent in different variants of the pickup scenario.

[112] https://hbrs-sesame.github.io/models/acceptance-criteria/bdd/templates/pickup.json
[113] https://hbrs-sesame.github.io/models/acceptance-criteria/bdd/pickup-variants.json
[114] https://hbrs-sesame.github.io/bdd-dsl/assets/img/bdd_graph.svg

```
{
  "@id": "pick-object", "@type": "bdd:ScenarioVariable",
  "bdd:of-scenario": [ "scenario-pick" ]
},
{
  "@id": "pick-workspace", "@type": "bdd:ScenarioVariable",
  "bdd:of-scenario": [ "scenario-pick" ]
},
{
  "@id": "pick-robot", "@type": "bdd:ScenarioVariable",
  "bdd:of-scenario": [ "scenario-pick" ]
}
```

Having defined the variables, we can now create `bdd:FluentClause` instances and attach them to `pick-given` and `pick-then` using the `bdd:clause-of` relation to extend `scenario-pick` with concrete clauses. Here, we have made a choice to represent BDD clauses as fluents[115], i.e. time-dependent predicates. The composable design allows us to make this choice without limiting our metamodel to this single representation. Other representations of BDD clauses can still be attached to `pick-given` and `pick-then` using the `bdd:clause-of` relation. More details on composable design can be found on the corresponding discussion on our kinematic chain modelling tutorial[116]. Furthermore, the `bdd:clause-of` relation can be used to attach any clauses to `pick-given` and `pick-then`, allowing extending `scenario-pick` with any number of use-case specific clauses.

```
{ "@id": "pred-obj-held-by-robot", "@type": "bdd:IsHeldPredicate" },
{ "@id": "after-pick", "@type": "bdd:TimeConstraint" },
{
  "@id": "fluent-obj-held-by-robot",
  "@type": "bdd:FluentClause",
  "bdd:clause-of": [ "pick-then" ],
  "bdd:predicate": "pred-obj-held-by-robot",
  "bdd:time-constraint": "after-pick",
  "bdd:ref-object": "pick-object",
  "bdd:ref-agent": "pick-robot"
}
```

In the example above, we define `fluent-obj-held-by-robot`, which asserts that the object is held by the robot at the end of the picking behaviour. This `bdd:FluentClause` instance is a composition linking to several elements in the template:

- Predicate `pred-obj-held-by-robot` is an instance of the domain-specific `bdd:IsHeldPredicate` concept for representing the fact that a robot is holding an object.
- Instances of `bdd:ScenarioVariable`, namely `pick-object` and `pick-robot`, which are subjects of `pred-obj-held-by-robot` in this context.
- Instance `after-pick` of type `bdd:TimeConstraint` which represents *when* `pred-obj-held-by-robot` should hold true.

The use of `bdd:IsHeldPredicate` implies a constraint that the subjects of `pred-obj-held-by-robot` must represent objects and agents in the scenario. Here, the use of `bdd:ref-object` and `bdd:ref-agent` indicates that `pick-object` is the object and `pick-robot` is the agent in this context.

[115] https://en.wikipedia.org/wiki/Fluent_(artificial_intelligence)
[116] https://github.com/comp-rob2b/modelling-tutorial

Additionally, the model at this point contains no assumption about how the fact that a robot is holding an object can be verified. Further transformations and/or generations can be introduced to produce concrete, executable implementations for verification.

Specifying A Concrete Scenario Variant The BDD scenario template defined above can now be extended with concrete variations, e.g. for generating concrete Gherkin feature files as shown in Section 5.3.2. This is done via linking the `bdd:ScenarioVariable` instances above to concrete instances of objects, workspaces and agents. For example, consider the use case where we want to test the pickup behaviour in the robotics lab at Bonn-Rhein-Sieg University using battery cells sent from AVL[117] (An use case partner of the SESAME project), as well as some objects readily available in the lab.

Specifying Concrete Agents, Environment, and Coordination Models We first need concrete models of the objects, workspaces, and agents that we may test with:

```
{ "@id": "bottle", "@type": "env:Object" },
{ "@id": "pouch1", "@type": [ "bdd:Object", "avl:PouchCell" ] },
{ "@id": "cylindrical1", "@type": [ "bdd:Object", "avl:PrismaticCell" ] },
{ "@id": "dining-table-ws", "@type": "env:Workspace" },
{ "@id": "kinova1", "@type": [ "agn:Agent", "kinova:gen3-robots" ] },
{ "@id": "kinova2", "@type": [ "agn:Agent", "kinova:gen3-robots" ] }
```

We also need a coordination model which defines the event that denotes the start of the pickup behaviour, which we can associate with `pickup-when`.

```
{ "@id": "pickup-start", "@type": "evt:Event" }
```

Specifying Variations Using the `task:Variation` concept, we can associate the `bdd:ScenarioVariable` instances above with possible entities via the `task:can-be` relation:

```
{
  "@id": "obj-variation", "@type": "task:Variation",
  "bdd:of-variable": "pick-object",
  "task:can-be": [ "bottle", "pouch1", "cylindrical1" ]
},
{
  "@id": "ws-variation", "@type": "task:Variation",
  "bdd:of-variable": "pick-workspace",
  "task:can-be": [ "dining-table-ws" ]
},
{
  "@id": "robot-variation", "@type": "task:Variation",
  "bdd:of-variable": "pick-robot",
  "task:can-be": [ "kinova1", "kinova2" ]
}
```

The `pick-when` can be associated with the concrete event `pickup-start`.

[117] https://www.avl.com

```
{
  "@id": "pick-when-event", "@type": "bdd:WhenEvent",
  "bdd:of-clause": "pick-when", "evt:has-event": "pickup-start"
}
```

We can now define variant `scenario-pick-brsu` as a composition of the `task:Variation` instances, and user story `us-obj-transport` as a composition of scenario variants, one of which is `scenario-pick-brsu`.

```
{
  "@id": "scenario-pick-brsu", "@type": "bdd:ScenarioVariant",
  "of-scenario": "scenario-pick",
  "task:has-variation": [
    "obj-variation", "ws-variation", "robot-variation"
  ]
},

{
  "@id": "us-obj-transport", "@type": "bdd:UserStory",
  "bdd:has-criteria": [ "scenario-pick-brsu" ]
}
```

`bdd-dsl` provide the `load_metamodels` utility method for initializing a `rdflib.Graph`[118] object with our metamodels[119]. Models can then be loaded to the graph with the `parse`[120] method:

```
from bdd_dsl.utils.json import load_metamodels

g = load_metamodels()
g.parse("path/to/model.json", format="json-ld")
```

5.3.2 Generating Gherkin Features from BDD Models

After creating scenario variants and templates, we can transform these models into other formats for use with existing tools. For example, we can generate from our BDD user stories Gherkin feature files, which has wide support for test automation in most programming languages, e.g. behave[121] library in Python. In the following example, we use the Jinja[122] template engine for the final text generation step.

Extracting Relevant Information and Transforming BDD Models Extracting the relevant information from a `rdflib.Graph` object can be done with SPARQL queries[123]. Additionally, JSON-LD Framing[124] can force a specific tree layout to the graph structure of the model, which can be easier to generate to text targets, as will be shown.

[118] https://rdflib.readthedocs.io/en/stable/apidocs/rdflib.html#graph
[119] https://github.com/hbrs-sesame/metamodels
[120] https://rdflib.readthedocs.io/en/stable/apidocs/rdflib.html#rdflib.graph.Graph.parse
[121] https://behave.readthedocs.io
[122] https://jinja.palletsprojects.com/
[123] https://rdflib.readthedocs.io/en/stable/intro_to_sparql.html
[124] https://www.w3.org/TR/json-ld-framing/

`bdd-dsl` provides several utilities for querying and framing models composed using our metamodels. The following example shows how BDD user stories akin to the example above can be transformed using the library's Python API:

```python
from pyld import jsonld
from bdd_dsl.models.queries import BDD_QUERY
from bdd_dsl.models.frames import BDD_FRAME
from bdd_dsl.utils.json import query_graph, process_bdd_us_from_graph

bdd_result = query_graph(g, BDD_QUERY)
model_framed = jsonld.frame(bdd_result, BDD_FRAME)

# alternatively, there's also an utility function that executes the above
# as well as doing some further cleanup of the result
cleaned_bdd_data = process_bdd_us_from_graph(g)
```

The code snippet above should produce the following JSON when run on the above model:

```
"data": [{
  "name": "us-obj-transport",
  "criteria": [{
    "name": "scenario-pick-brsu",
    "scenario": {
      "name": "scenario-pick",
      "then": {
        "clauses": {
          "name": "fluent-obj-held-by-robot",
          "agents": { "name": "pick-robot" },
          "objects": {"name": "pick-object"}
        },
      },
      "when": {
        "name": "pick-when",
        "trans:has-event": { "name": "pickup-start" }
      }
    },
    "variations": [
      {
        "name": "obj-variation",
        "entities": [
          {"name": "bottle"}, {"name": "pouch1"}, {"name": "cylindrical1"}
        ],
        "trans:of-variable": {"name": "pick-object"}
      },
      ...
    ]
  }]
}]
```

Generating from Jinja Templates The extracted and transformed JSON data can be used to automatically render feature files using Jinja[125] with the template below:

[125] https://jinja.palletsprojects.com/api/

```
Feature: {{ data.name }}
{% for scenario_data in data.criteria %}
  Scenario Outline: {{ scenario_data.name }}
    {% for clause in scenario_data.given_clauses %}
    {{ clause|safe }}{% endfor %}

    When "{{ scenario_data.when_event }}"
    {% for clause in scenario_data.then_clauses %}
    {{ clause|safe }}{% endfor %}

    Examples:
    |{% for var_name in scenario_data.variables %} {{ var_name }} |{% endfor %}
    {% for entity_data in scenario_data.entities %}|{%
      for entity_name in entity_data %} {{ entity_name }} |{% endfor %}
    {% endfor %}
{% endfor %}
```

The up-to-date version of this template is available online[126] for download. `bdd-dsl` also provides utilities to further process the transformed data shown above before performing the final text transformation with Jinja. The code snippet below should generate one feature file for each `bdd:UserStory` instance.

```
from bdd_dsl.utils.jinja import load_template, prepare_gherkin_feature_data

# load Jinja template
feature_template = load_template("feature.jinja", "/template/directory")

# loop through data transformed using the code snippet in the previous section
for us_data in processed_bdd_data:
  us_name = us_data["name"]
  prepare_gherkin_feature_data(us_data)
  # the rendered text can be written normally to a file to produce
  # the Gherkin feature
  feature_content = feature_template.render(data=us_data)
```

The generation result should produce be a valid Gherkin feature file like shown below:

```
Feature: us-obj-transport

  Scenario Outline: scenario-pick-brsu
    Given "<pick_object>" is located at "<pick_workspace>"
    When "pickup-start"
    Then "<pick_object>" is held by "<pick_robot>"

    Examples:
    | pick_object | pick_workspace | pick_robot |
    | env:brsu/bottle | env:brsu/dining-table | agn:brsu/kinova1 |
    | env:avl/cylindrical1 | env:brsu/dining-table | agn:brsu/kinova2 |
    ...
```

The generated feature files can then be used with existing BDD frameworks, e.g. behave[127], for test automation. Currently, the tools necessary to generate executable implements to verify the generated scenarios in simulation are still under development. Once ready, this tutorial will be extended to include usage of these new tools.

[126] https://hbrs-sesame.github.io/models/acceptance-criteria/bdd/feature.jinja
[127] https://behave.readthedocs.io

References

[1] D. North, "Behavior Modification: The evolution of behavior-driven development," *Better Software*, vol. 2006, no. 3, pp. 26–31, Jun. 2006.

[2] M. Alferez, F. Pastore, M. Sabetzadeh, L. Briand, and J.-R. Riccardi, "Bridging the gap between requirements modeling and behavior-driven development," in *2019 ACMIEEE 22nd Int. Conf. Model Driven Eng. Lang. Syst. MODELS*, Sep. 2019, pp. 239–249.

[3] S. Parra, A. Ortega, S. Schneider, and N. Hochgeschwender, "A thousand worlds: Scenery specification and generation for simulation-based testing of mobile robot navigation stacks," in *IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 2023.

[4] S. Schneider, N. Hochgeschwender, and H. Bruyninckx, "Domain-specific languages for kinematic chains and their solver algorithms: Lessons learned for composable models," in *IEEE international conference on robotics and automation (ICRA)*, 2023.

[5] D. Nurchalifah, S. Blumenthal, L. Lo Iacono, and N. Hochgeschwender, "Analysing the safety and security of a UV-c disinfection robot," in *IEEE international conference on robotics and automation (ICRA)*, 2023.

[6] M. Nguyen, N. Hochgeschwender, and S. Wrede, "An Analysis of Behaviour-Driven Requirement Specification for Robotic Competitions," *RoSE International Workshop on Robotics Software Engineering*, 2023.

[7] S. Schneider and H. Bruyninckx, "Exploiting linearity in dynamics solvers for the design of composable robotic manipulation architectures," in *2019 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 2019, pp. 7439–7446. doi: 10.1109/IROS40897.2019.8968500[128].

[8] R. Featherstone, *Rigid body dynamics algorithms*. 2008-01-01, 2008.

[9] "IEEE standard ontologies for robotics and automation," *IEEE Std 1872-2015*, pp. 1–60, 2015, doi: 10.1109/IEEESTD.2015.7084073[129].