Automatic Modelica Package and Model Generation from Templates and Data Files with Python, Exemplified with URDF

Antoine Pignède[1] Carsten Oldemeyer[1]

[1]German Aerospace Center (DLR), Institute of Robotics and Mechatronics, 82234 Weßling, Germany
{Antoine.Pignede,Carsten.Oldemeyer}@dlr.de

Abstract

Creating correct Modelica models and packages from templates and data files describing a multi-physics system is useful in numerous situations. For example, in laying out power plants, designing airplane air conditioning or chemical reactions (Barth and Fay 2013; Santillan Martinez et al. 2018; Ramonat et al. 2025). This publication shows two possibilities of doing this with an example from robotics design and simulation. The `URDFModelica` library contains templates for links, joints and whole robots that can be mobile or stationary. The library also has a Python script that takes a valid URDF file as input. With minimal manual processing of this input file, a complete robot simulation package is created automatically. Alternatively, the input file, translated to a Modelica record, can be used to set the parameters and connections of a generic robot model. The `URDFModelica` library has already been successfully used for quick generation of first Modelica simulations of existing robots or robots yet to be fully developed. The general structure and approach can be adapted to other application domains without much effort. `URDFModelica` has been released as open source Modelica library on GitHub.

Keywords: model and package generation, Python Modelica processing, URDF, robotics

1 Introduction

There are numerous examples through all domains where Modelica is one among several tools for design, development, modeling, simulation, analysis etc. Keeping a consistent set of parameters through all tools is crucial but can be challenging and the same also holds for model structures and elements interaction. Ideally, there is one data file or set of files that compiles all this information in usable form and with consistent units. Examples of widely accepted data formats are **J**ava**S**cript **O**bject **N**otation *JSON*[1], E**x**tensible **M**arkup **L**anguage *XML*[2] and **Y**AML **A**in't **M**arkup **L**anguage™*YAML*[3].

But even if the data file fulfills these requirements, there needs to be some function in the tools to read and possibly write the data file. The `ExternData`[4] Modelica library (Beutlich and Winkler 2021) for example, does that. It reads files at initialization to set parameters, initial values and lookup tables. But the Modelica user still needs to create the models.

Taking robotics as example, there are situations when one data file can contain more information than just parameter and initial values. The **U**nified **R**obot **D**escription **F**ormat *URDF*[5] [6], introduced by the **R**obot **O**perating **S**ystem *ROS*[7] (Macenski et al. 2022), describes a robot as a set of links with appropriate attributes and joints that connect links together, more details in subsection 2.1. As such it can not only be used to set parameters but to create whole packages automatically. This works well because the model structure is fixed before initialization. URDF is supported by many CAD software packages. Simulation tools, e. g. CoppeliaSim[8] and Isaac[9], can build simulation models starting from a URDF file. The similar capability for Modelica is newly introduced in this work. The new `URDFModelica` library developed at the German Aerospace Center (DLR) parses a valid URDF file and automatically creates a complete robot simulation model as a new Modelica package or as a Modelica record that configures one model using variable-length arrays. For the simulation model as Modelica package, it creates separate models and parameter records for each link and joint as well as the robot model, that collects all elements and connects them to chains. It is thus possible to adapt all elements individually whenever needed without affecting all elements of the same kind. The separation between parameters and models also permits an update function that only rewrites parameter values. For the simulation model as single Modelica record, a generic robot model is contained in the library. This has two arrays of variable length for links and joints. Connections between these and for the flanges of active joints are managed by two matrices generated from the URDF source by the Python script. This way to automatically create a robot simulation needs

[1] `https://www.json.org/`
[2] `https://www.w3.org/XML/`
[3] `https://yaml.org/`
All URLs in footnotes are accessed July 22, 2025.

[4] `https://github.com/modelica-3rdparty/ExternData`
[5] `http://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/URDF-Main.html`
[6] `https://wiki.ros.org/urdf`
[7] `https://www.ros.org/`
[8] `https://coppeliarobotics.com/`
[9] `https://developer.nvidia.com/isaac`

considerably less code but one cannot change individual elements in contrast to the simulation as new package.

Translation in the other direction, i. e. from Modelica to URDF, is currently not implemented and not planned for the near future because no application has been identified.

`URDFModelica` is different from `ExternData`, but also different from `Modelica-Builder`[10]. It does not manipulate arbitrary Modelica files, but only translates from one format to Modelica syntax for the special application type of robotics simulation. It also is different from the supported automated design (Matei et al. 2024) whose aim is to automatically create Modelica models, too. But, `URDFModelica` only translates an existing design from one description language to another while the supported automated design uses optimization from basic elements to create new models fulfilling specific tasks.

All three categories, read parameters defined externally, manipulate individual Modelica files externally, create whole packages for special applications automatically, have their area of applications and are useful in many fields. The approach and structure of the new library explained here can be adapted to other application domains without much effort. Therefore, the interest of this publication may go beyond the Modelica robotics community.

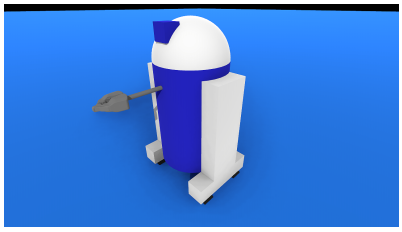

Figure 1. DLR Visualization 2 of the example mobile robot described in URDF and simulated in Dymola.

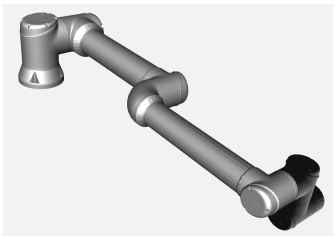

Figure 2. Standard Library Animation of the example stationary robot described in URDF and simulated in OpenModelica.

As appetizer, Figure 1 and Figure 2 show a mobile and a stationary robot created and simulated with `URDFModelica`. These are the R2D2-inspired example robot of the official ROS 2 documentation and the UR10e manipulator from Universal Robots[11]. They are contained as examples in the library, that has been released as open source Modelica library in Summer 2025: **`https://github.com/DLR-RM/urdfmodelica`**.

[10] `https://github.com/urbanopt/modelica-builder`

[11] `https://www.universal-robots.com/products/ur10e/`

The purpose of this publication is twofold. The first purpose is to generally explain how package and model generation can be done. The methodology is adaptable to other domains and other similar research questions.

The second purpose is to introduce the new `URDFModelica` library. This is split into two sections because two approaches for simulating robots based on URDF are implemented. The *package* approach and the *model structure defining record* approach. Independent of this, there are two variants for visualization, either with multibody animation from the Standard Library or the DLR Visualization 2 (Kümper et al. 2021).

Therefore, the text is structured along the two approaches, first the package then the record approach. Each time, a concept is explained theoretically first and then exemplified. Before this, the data source file and a general overview of `URDFModelica` are explained because they are the basis for both approaches.

2 Data Source File

Automatic package or model generation at first needs a good data source. There are numerous formats for this that cannot be covered entirely here, but some general guidelines with Modelica in mind can be given.

Ideally, one should adapt a few things before the Modelica package is created. For example, the export may result in wrong data types like real-valued color triplets instead of the integer-valued triplets required by Modelica. Most probably, file paths to resources such as lookup tables or CAD files for visualization must be changed in order for Modelica to find them. The recommendation is to put all these files in the resources directory of the Modelica package and refer to them relatively following the pattern `filename="modelica://Package/Resources/..."`. These steps are easier if the format is human-readable, but that is not a hard requirement.

Unfortunately, users tend to stretch, go beyond or abuse the capabilities or intentions of the data formats even if they are standardized. For example, JSON does not include comments, although they are occasionally seen in practice. YAML, as a second example, is not standardized. This makes handling, especially of large files, difficult and checking correctness is nearly impossible.

Based on experience of the authors, it often is beneficial not to include absolutely everything in the data file but rather restrict to the minimum that everybody agrees on. The automatically created package will almost always need post processing anyway.

2.1 Example: URDF and Modelica

> «*URDF* (**U**nified **R**obot **D**escription **F**ormat) is a file format for specifying the geometry and organization of robots in ROS.»[12]

[12] `http://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/URDF-Main.html`

DOI 10.3384/ecp218497

URDF is in XML style and thus human-readable. A robot is a set of interconnected links and joints. Links have inertial properties, and optionally visual or collision properties. Joints have a type (fixed, rotational, translational, etc.), an axis, a parent link, a child link and a few other optional properties. URDF reserves keywords for more elements such as sensors and special Gazebo[13] elements, but these are not adopted by every software. For example, the `urdf_parser_py`[14] Python module, on which the conversion from URDF to Modelica is based on, only parses links and joints.

URDF has become one of the most popular formats in the robotics community (Tola and Corke 2023). Tola and Corke (2024) lists and discusses a comprehensive compilation of URDF robot models through the world. A large number of known CAD software are able to export their models as URDF, the ROS 2 documentation export page has a good overview with links to respective manuals.[15]

Thanks to the already existing Python parser for URDF, minimal preprocessing of the data source file is needed. Most important is that file paths point to resources that Modelica can find. The parser does not check for consistent values, e. g. whether an axis attribute is a unit vector, and recognizes only the minimal set of elements and attributes that everybody agrees on. For example, the ROS 2 documentation defines contact coefficients for the link collision model that are not recognized by `urdf_parser_py`.[16]

3 `URDFModelica` Library

The `URDFModelica` library contains records, models and resources to easily set up a simulation from an existing URDF file. See the library tree Figure 3. The two approaches described here, are contained in the library.

3.1 `URDFModelica` Links

The link model can be any rigid robot body or represent some other important robot part like a camera position or a measurement point. Associated with a link model is a link parameter record that has three records: inertial, visual and collision. These three parameter records themselves contain further records down the hierarchy until primitive data types such as mass and inertia real values, integer color vectors or strings for paths to CAD files.

A link always has an inertial component, that is a `Body` from the multibody Standard Library with parameters from the corresponding record, which can be zero.

Link visualization is optional. It is always included, but only active if a Boolean parameter is true. In the "main" variant, it is simulated using the visualizers from the Modelica multibody visualizers. The `geometry` type that is part of the link parameter record specifies which visualizer to conditionally include. The user may also choose to replace the Standard Library visualizers with the corresponding elements from the DLR Visualization 2 library with a simple script option.

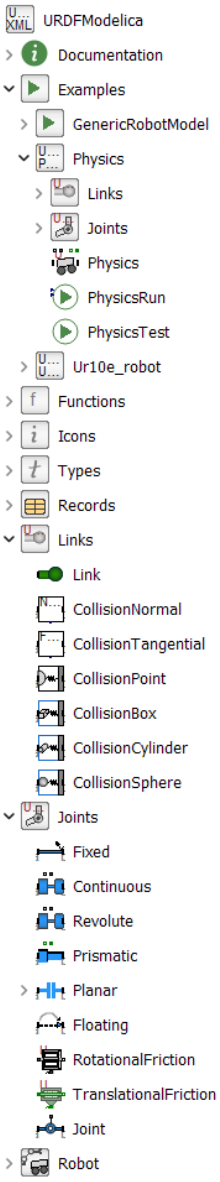

Figure 3. URDFModelica library tree.

Collision of links is optional, too. Again, it is always included in the template and active only if a Boolean parameter is true. Because the focus of `URDFModelica` is on the package generation and translation from URDF but not on contact detection and dynamics, collision is not simulated but only visualized. The same visualizers from Standard Library or DLR Visualization 2 as for link visualization are included equally in each link that has collision information. Color is pink for easier distinction. Collision can be globally turned off with a parameter that is propagated through the whole robot.

The link model stands as is and does not need to be processed by the automatic package generation to work. One can add a link to any multibody model of Modelica because all parameters have default values and optional components are conditional. The advantage of the URDF link compared to a Standard Library body is the attached parameter record with which it is easy to have different rigid bodies with optional visual and collision models. Additionally, the resulting Modelica model uses the same parameters as defined by URDF, which makes the backwards translation (currently not automatized) simple.

3.2 `URDFModelica` Joints

Joints connect links together. Different from links, there is not one joint with conditional components, but different models for each type. Still, all joint models have an instance of the same joint parameter record. This record has an enumerated type and nine records, e. g. for the parent and child link names, the joint axis, limits for effort and velocity, friction and calibration information. Note that not all information contained in the URDF joint description (and `URDFModelica` joint parameter record) are used in Modelica simulations.

[13] `https://gazebosim.org/home`

[14] `https://github.com/ros/urdf_parser_py`

[15] `http://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/Exporting-an-URDF-File.html`

[16] `http://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/Adding-Physical-and-Collision-Properties-to-a-URDF-Model.html#contact-coefficients`

There are six types. All of them first apply a Standard Library `FixedRotation` to set their origin.

- `Fixed` joints rigidly connect two links together without any degree of freedom.
- `Continuous` joints connect two links around a `Revolute` from the multibody Standard Library whose axis is set by the `Axis` parameter record. The joint flanges are also propagated to the outside such that the angle can be controlled. An integer parameter can optionally be set to mark the flange number for easier reference in a robot diagram.
- `Revolute` is the same as `Continuous` with the difference that limits for angle (upper and lower limit), angular velocity absolute value and effort (torque) absolute value are checked against at. A warning is output if a limit is violated.
- `Prismatic` is the same as `Revolute` for translational connection between links using `Prismatic` from the multibody Standard Library. All prismatic joints have limits for position (upper and lower limit), velocity absolute value and effort (force) absolute value. These are zero by default as defined by URDF. A warning is output if a limit is violated.
- `Planar` joints connect two links together with three degrees of freedom around a `Planar` from the multibody Standard Library. The `Axis` parameter record from URDF is perpendicular to the unconstrained degrees of freedom. A function calculates the simplest possible required n_x parameter by exchanging two vector elements of which one is not zero such that it can be negated.
- `Floating` joints connect two links together with all six degrees of freedom around a `FreeMotion` from the multibody Standard Library.

`Revolute` and `Prismatic` joints may have friction. URDF reserves two parameters for this: a friction and a damping coefficient. Therefore, simple friction models are also contained in `Joints`. The friction force f_r depending on velocity v is

$$f_r = R_v v + \frac{2R_h}{1+\exp(-sv)} - R_h$$

that is continuously differentiable. Parameters are the viscous friction coefficient R_v (URDF damping), stiction coefficient R_h (URDF friction) and steepness s of stiction (default is 100). Rotational friction is the same with torque and angle replacing force and position.

The joint models stand as is and do not need to be processed by the automatic package generation to work. One can add any joint to any multibody model of Modelica because all parameters have default values and optional components are conditionally included.

3.3 `URDFModelica` Background Information

`URDFModelica` only relies on the Modelica Standard Library (`4.0.0`) and needs Python to translate from URDF to Modelica. It has been developed and tested on OpenModelica `v1.24.0`[17] (Fritzson et al. 2020) and Dymola `2024x`[18]. The *model structure defining record* approach however does not yet work with OpenModelica. Instead of the default Standard Library visualizers, one may also choose to do object visualization with the DLR Visualization 2 library (Kümper et al. 2021). Both variants are provided in the library and an option on translation lets the user decide which variant to take. The "visualization 2" version has been developed and tested only with Dymola because DLR Visualization 2 currently does not support OpenModelica. The restricted free community edition[19] suffices to run the examples.

The Python code has been developed and tested with Python `3.12` from the official sources[20]. The module `urdf_parser_py` has been installed with `pip`.

4 Automatic Package Generation in Modelica

Creating a Modelica package automatically is not a trivial undertaking, even with well-defined data source and templates. A general approach that is adaptable to many domains and situations is explained in this section. This is the first approach of this publication subsequently referred to as *package* approach.

In overview terms, there is a Modelica library that contains template models, parameter records definitions and resources. Among the resources is a Python translation script and space for necessary files for the automatically created package to work correctly. These can be lookup tables, CAD files, JSON initialization files, etc. The separation of parameter records and Modelica models is an important part of the proposed automatic package generation. Counterintuitively, created records and files do not extend form the templates but do full copies of their content. The advantage of individual model alteration is judged higher than the drawback that one cannot simply change all similar models in one step. This and more aspects will be elaborated in the following subsections.

4.1 Modelica Records

A central part of the suggested package structure is to separate parameters from models, a general pattern that is widely applied in modeling and simulation. Thus, the first task is the definition of Modelica records. Here, one must replicate the elements contained in the data source file that

[17] `https://openmodelica.org/`

[18] `https://www.3ds.com/products/catia/dymola`

[19] `https://www.sr-scil.de/simulationsbibliotheken/kommerziell-verfuegbar/visualization/`

[20] `https://www.python.org/`

DOI 10.3384/ecp218497

are potentially needed for the simulation model in Modelica syntax. A hierarchical approach and providing default values is recommended.

When the automatic package generation is run, instances of the parameter records are created and given the values of the data source file. These copies are given the same name as the model they refer to, with "Parameters" as suffix and saved as single files in the new package.

4.1.1 `URDFModelica` Records

A URDF-described robot is a set of links and joints. Thus, there is not one but two records on the top of the hierarchy.

4.2 Modelica Template Models

Having template Modelica models helps to automatically create useful Modelica packages. One could create partial templates and then extend the concrete instances. However, a different approach is put forward here. Having copies of the templates allows to change individual models after generation. The drawback that changes cannot be applied to all models at once, is deemed less important. If needed, packages can be recreated at any moment, but this comes at the cost of losing individual modifications.

A design choice concerns the number of template models and enumerated types. One can decide to have one template model that fits all purposes and selects its conditional components based on the variant. Otherwise, one can choose to define one, generally simpler, template model for each variant. The decision which approach to use, depends on preference, data source, and mainly on the interface to other parts of the model. Less important for the decision are the icons and number of components in each template. Both approaches are implemented in the example of this publication. There is one template for all links, regardless whether they are boxes or cylinders, and six templates for joints, one for each joint type defined by the data source standard. All joint types have their own template because some of them have flanges in addition to the two multibody frames needed by all joints. All links use the same template because they all only have one multibody frame to attach them to joints or other links. Conditional components inside a link do not introduce much complexity. On the contrary, adaptation to other link types, e. g. converting a CAD defined wheel to a cylinder, is made easy.

A model always needs the corresponding parameter record. The template itself has its parameter record as replaceable component. This permits instances of the template model to stand for themselves and be added manually to any Modelica model. Then, one can select existing records, possibly overwrite the defaults and instantiate new models. When the automatic package generation is run, the replaceable parameter record is not copied. Instead, the correct record, that is the one with the same name plus "Parameters" suffix, is added without the "replaceable" keyword because one should not need to replace the parameter record with another one in post processing. The instance name is always "parameters" for easy reference.

Other contents of templates can be divided in required and optional components. Required components, e. g. interface, are simply copied when automatically created. Optional components are declared conditional. This is a second requirement so that instances of a template can stand for themselves. When the automatic package generation is run, one could copy all optional components relying on Modelica conditional components. However, the Python script only copies the needed optional components to make the new model smaller.

4.2.1 `URDFModelica` Templates

Because a URDF-described robot is a set of links and joints, there are two packages with templates. Thus, the automatic package generation will create two subpackages for links and their parameters as well as for joints and their parameters.

Link Template

The link model described in subsection 3.1 serves as template for links described by URDF.

When a new package is created, copies of the link template and record are put in a new `Links` subpackage for each link. Thus, the link models can be adapted independently of the parameter definitions. Similarly, parameter values can be updated without modifying the link models.

The correct visualization and collision components (box, cylinder, sphere, CAD file), if any, are selected from all components contained in the template based on the `geometry` type that is part of the link parameter record.

Joint Templates

The joint models described in subsection 3.2 serve as template for joints described by URDF.

When a new robot package is created, copies of joints of appropriate type are put in the new `Joints` subpackage for each joint. A joint parameter record copy is added for each joint, too. Thus, the joint models can be adapted independently of the parameter definitions, parameters which can be updated at any time without modifying the joint models.

Different from links, joints have no conditional components because there is not one general joint template, but six more specialized joint templates.

4.3 Aggregate Modelica Template Model

The last required template is the one that combines all elements together to the whole system. This template is mostly empty apart from the interface. When the automatic package generation is run, the previously created models of the subsystems are added to a new system model and connected together based on the data source file. This is done in the Python script with string manipulation functions that create correct Modelica syntax before and after the "equation" keyword.

4.3.1 `URDFModelica` Robot Template

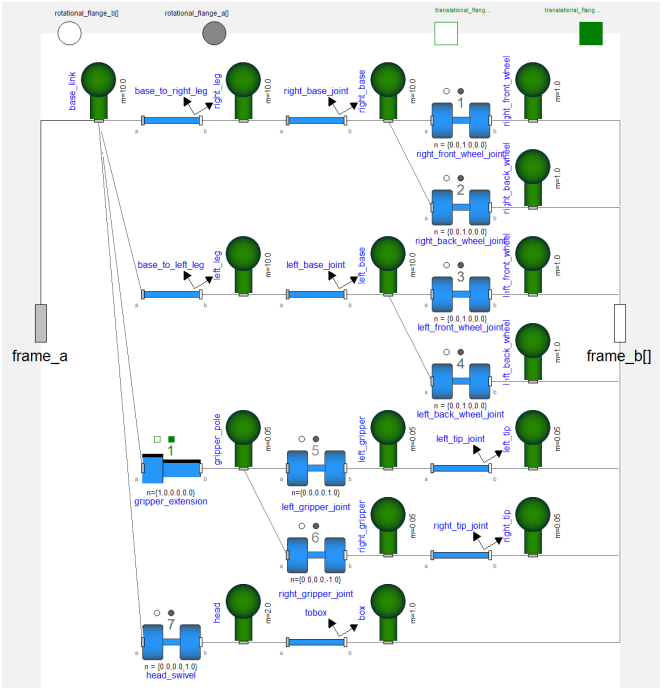

Figure 4. Graphics view of the example mobile robot.

The `Robotname` model puts all links and joints together and connects them according to the parent and child link string contained in each joint record. The graphics view of the R2D2-inspired example (Figure 1) is shown in Figure 4. The first link is connected to a `frame_a` on the left. All links that are leaves of chains are connected to a vector of `frame_b` on the right of the model to allow for interaction with external components. Additionally, rotational and translational flanges are propagated to the outside for every continuous, revolute and prismatic joint. The correct array sizes are calculated automatically. For easier reference in the joint icons, the flange number is also given as parameter to the concerned joints. The robot model diagram size is automatically adjusted to show all elements. This is the last model relying on the URDF source description, the further two Modelica models are suggestions for usage.

4.4 Modelica Usage Models

Further templates for how to create example models or FMUs may also be convenient for users.

4.4.1 `URDFModelica` Usage Models

For example, `RobotnameRun` builds an environment around `Robotname`. There is a multibody `World`, and outside inputs for position and angle of joints. `RobotnameRun` is one possible model to compile as FMU[21] (Blochwitz et al. 2012).

`RobotnameTest` is a model to quickly check that the package has been created correctly by providing some simple inputs to all needed signals.

[21] `https://fmi-standard.org/`

4.5 Python Script

The actual generation of the Modelica package is done with a Python script. Modelica also has text and file manipulation functions, but lacks the versatility and ease of Python. For example, there are parser modules for XML or JSON files that make the translation from these formats to Modelica syntax a lot easier.

Inputs to the Python script are the data source file and the location where to save the new package. The latter must be divided into two inputs, the path to the directory where the Modelica package structure begins, and the Modelica package inside this package structure. This is needed because each Modelica file begins with the keyword “within” and the Modelica package structure.

4.5.1 Package Generation

The Python script in general performs the following tasks.

1. Parse the input file. This is one of the main reasons to use Python because there already are modules for many data formats such as JSON, XML or YAML.
2. For each element
   (a) Create a new parameter record by translating the parsed information to Modelica syntax. Save the parameter record in the given package structure, possibly in further subpackages, with element name plus “Parameters” suffix.
   (b) Create a new model by copying only the needed contents of the template. Add the parameter record as “parameters” instance. Save the model in the given package structure, possibly in further subpackages, with element name.
3. Create the model that combines all elements to the system. Add all elements and connect them based on the actual source. Save this model in the given package structure with system name.
4. Optionally create and save a model that shows how to use the system.
5. Create the package order file of the new package.
6. Create the package definition file of the new package.
7. Update the package order one hierarchy level higher.

Note also that it is recommended to convert all names to title case, i. e. first letter upper case and the rest lower case, and that they should not contain "-" (minus) signs or be reserved keywords.

4.5.2 Parameter Update

One main reason for the division between parameters and models is to be able to update parameters without rewriting the whole package. This is possible if only numerical parameter values change in the data source file. In that

DOI 10.3384/ecp218497

case, the same Python script as above only recreates the parameter record files and skips all the rest.

Because the models of elements and the overall system model remain unchanged, it is not possible to change element types or connections in update mode. If the data source has such changes, one must completely rewrite the whole package or create a new one.

4.5.3 `URDFModelica` Python Script for Package Approach

The Python script `urdf_to_package.py` in the resources does the translation from URDF to Modelica syntax and creates or updates the Modelica package. It relies on the existing Python URDF parser but does not need further information or other non-standard modules otherwise.

Given a URDF file it first translates all information into one Modelica record exactly replicating the URDF contents. Then it writes, or rewrites, the parameter records of links and joints in the specified package. If the user chooses the update mode, the script terminates. Otherwise the entire package structure, with joints models, link models, robot model and example files, is recreated. Details about specific steps in this process are given in the respective paragraphs about links, joints, etc. further above.

5 Simulation Based on Modelica Record

Instead of generating new packages for every source file, an alternative way is a model that adapts to the structure during translation. While Modelica is not capable of handling systems with variable structure during simulation, it supports component arrays with varying sizes as long as those are known during translation. This is the second approach of this publication subsequently referred to as *model structure defining record* approach.

While retrieving the connection information from the data source file with a Modelica function is theoretically possible, it does not work with the Dymola version used by the authors. The tool must be able to expand the subscripts of connected variables into literals. Hence, a `connectionMatrix` containing that information is written into the single, complete record by the Python script as a workaround and used as shown in Listing 1. Each connection corresponds to a matrix row containing the indices of connected elements in its columns. Note that Listing 1 is adapted to the URDF and Modelica example and thus has two arrays of different elements (links and joints) and one connection matrix, because URDF defines parent links and child links to each joint.

Similarly, matrices for outside connections must be retrieved from the data source file and added to the model. In the robotics example of this text, it is the flanges of rotational or translational joints.

Finally, knowing the base and end elements of chains is usually necessary for interfacing with the environment. Roots and leaves can be determined relatively easily from the `connectionMatrix`.

Compared to the package generation approach of section 4, the simulation based on a single record reduces the amount of generated code and therefore maintenance effort considerably. In fact, the same few lines of Modelica code are enough to simulate all translated data source files. This implementation lends itself to use-cases where the system is frequently changed and used without modifications, e. g. when choosing the right system for a given task based on a simulation study.

A drawback is the lack of accessible presentation of the mapping from component name to index in the component vector that is more user-friendly with the package generation approach, see Figure 4. Changes in individual components are not possible either. As of writing model translation fails with OpenModelica with preliminary analysis indicating an error in component array handling.

5.1 `URDFModelica`: Simulation Based on Modelica Record

The `URDFModelica` library also does simulation from URDF source in a generic form following the *model structure defining record* approach. The source file rigidly defines the number of joints and links as well as their connections. A literal URDF robot implementation consists of two variable size arrays of links and joints whose size is the same as their respective records in the `Robot` record. The DLR Robots library (Bellmann et al. 2020) frequently uses this alternative way to create robots from URDF.

Listing 1. Simplified Excerpt of Robot Model.

```
model GenericRobot
 (...)
 Records.Robot data;
 Integer nJ = size(data.joint, 1);
 Integer nL = size(data.link , 1);
 Integer cM[nJ, 2] = data.connectionMatrix;
 Joint joint[nJ](jointParams = data.joint);
 Link  link [nL](linkParams  = data.link);
equation
 for iJ in 1:nJ loop
  connect(link[cM[iJ, 1]].frame_a,
   joint[iJ].frame_a);
  connect(joint[iJ].frame_b,
   link[cM[iJ, 2]].frame_a);
 end for;
 (...)
end GenericRobot;
```

The Python script of subsubsection 4.5.3 also creates a parameter record `Robot`, that contains the complete translation in one file. The translated links and joints that are used individually in the *package* approach, first are compiled to two arrays. Then, two additional matrices are generated from the joint information.

- `connectionMatrix[numJoints,2]`, theoretically explained in section 5, in practice becomes an array where each joint corresponds to a matrix row containing parent and child link indices in its columns. Usage is shown in Listing 1.

- `moveableJoints[numAxes,2]` has the joint number and movable axis number for every `Continuous`, `Revolute` and `Prismatic` joint. This sets connection of joint flanges to outside components at model translation.

Another part of the interface is a base frame that is used to place the robot in its environment. Which link to connect to the base frame is determined based on the `connectionMatrix` with a Modelica function that searches for the single root of the connection tree.

Higher level components need to provide the URDF robot record to the robot model via modifier.

6 Conclusion and Further Work

Automatic package generation or parameterization of generic models can be very useful, especially to keep consistent parameters and structures of models. It simplifies the initial setup and encourages good modeling practices, such as separating parameters from models.

This text explains two approaches, *package* and *model structure defining record*, that are adaptable to many domains and applications without much effort. The new `URDFModelica` library implements both approaches. It exemplifies the approaches and shows their practicality. The library has been released open source under BSD license on GitHub in Summer 2025. First users report quick translation, even of large robots (> 25 joints), and in general stable execution.

Further work can go in different directions. While there is not much to add to the general and more theoretical terms, `URDFModelica` can be expanded for example for proper collision models (Buse et al. 2023) or include more advanced drives or expand the range of robot elements. Although the Python parser does not recognize sensors, the URDF documentation reserves keywords for IMUs, cameras etc. An in-house extension to the library that also considers this, is already in use at DLR. Once `URDFModelica` has been released as open source Modelica library, efforts for extending `urdf_parser_py` will be undertaken. More examples of how to use the created Modelica models is another possibility for further work. For example, calculation of Jacobian or Coriolis matrices are important in robotics research.

Finally, the claim that the approach can be adapted to other domains, remains to be proven.

Acknowledgements

The authors would like to thank Robert Reiser for being the first person to test the library and Bernhard Thiele for proof reading the manuscript.

This work was supported by the Helmholtz Association project iFOODis (contract no. KA-HSC-06 iFOODis).

References

Barth, M. and A. Fay (2013). "Automated generation of simulation models for control code tests". In: *Control Engineering Practice* 21.2, pp. 218–230. DOI: 10.1016/j.conengprac.2012.09.022.

Bellmann, T. et al. (2020). "The DLR Robots library – Using replaceable packages to simulate various serial robots". In: *Asian Modelica Conference 2020, 8-9 October 2020, Tokyo, Japan*. Linköping University Electronic Press, pp. 153–161. DOI: 10.3384/ecp2020174153.

Beutlich, T. and D. Winkler (2021). "Efficient Parameterization of Modelica Models". In: *14th International Modelica Conference, 20-24 September 2021*. Linköping University Electronic Press, pp. 141–146. DOI: 10.3384/ecp21181141.

Blochwitz, T. et al. (2012). "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models". In: *9th International Modelica Conference, 3-5 September 2012, Munich, Germany*. Linköping University Electronic Press, pp. 173–184. DOI: 10.3384/ecp12076173.

Buse, F. et al. (2023). "A Modelica Library to Add Contact Dynamics and Terramechanics to Multi-Body Mechanics". In: *15th International Modelica Conference, 9-11 October 2023, Aachen, Germany*. Linköping University Electronic Press, pp. 433–442. DOI: 10.3384/ecp204433.

Fritzson, P. et al. (2020). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control* 41.4, pp. 241–285. DOI: 10.4173/mic.2020.4.1.

Kümper, S. et al. (2021). "DLR Visualization 2 Library - Graphical Environments for Virtual Commissioning". In: *14th International Modelica Conference, 20-24 September 2021*. Linköping University Electronic Press, pp. 197–204. DOI: 10.3384/ecp21181197.

Macenski, S. et al. (2022). "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science robotics* 7.66, eabm6074. DOI: 10.1126/scirobotics.abm6074.

Matei, I. et al. (2024). "Modelica supported automated design". In: *American Modelica Conference 2024, 14-16 October 2025, Storrs, CT, USA*. Linköping University Electronic Press, pp. 43–52. DOI: 10.3384/ECP20743.

Ramonat, M. et al. (2025). "Automated Generation of Simulation Models Based on Plant Engineering Data". In: *Workshop der ASIM Fachgruppen GMMS – Grundlagen und Methoden in Modellbildung und Simulation und STS – Simulation Technischer Systeme, 10-11 April 2025, Oberpfaffenhofen, Germany*. ARGESIM Reports. ASIM and ARGESIM, pp. 79–86. ISBN: 978-3-903347-66-3.

Santillan Martinez, G. et al. (2018). "Automatic Generation of a High-Fidelity Dynamic Thermal-Hydraulic Process Simulation Model From a 3D Plant Model". In: *IEEE Access* 6, pp. 45217–45232. DOI: 10.1109/ACCESS.2018.2865206.

Tola, D. and P. Corke (2023). "Understanding URDF: A Survey Based on User Experience". In: *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE), 26-30 August 2023, Auckland, New Zealand*. IEEE. DOI: 10.1109/CASE56687.2023.10260660.

Tola, D. and P. Corke (2024). "Understanding URDF: A Dataset and Analysis". In: *IEEE Robotics and Automation Letters* 9.5, pp. 4479–4486. DOI: 10.1109/LRA.2024.3381482.