ESROCOS
STRUCTURAL MODELLING
DOCUMENT
ESROCOS_D4.2

<table>
<thead>
<tr>
<th>Due date of deliverable:</th>
<th>31-08-2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start date of project:</td>
<td>01-11-2016</td>
</tr>
<tr>
<td>Duration:</td>
<td>27 months</td>
</tr>
<tr>
<td>Topic:</td>
<td>COMPET-4-2016 Building Block a) Space Robot Control Operating System</td>
</tr>
<tr>
<td>Work package:</td>
<td>4100</td>
</tr>
<tr>
<td>Lead partner for this deliverable:</td>
<td>KUL</td>
</tr>
</tbody>
</table>

This project has received funding from the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement No 730080.

<table>
<thead>
<tr>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>CO-1</td>
</tr>
<tr>
<td>CO-2</td>
</tr>
</tbody>
</table>

Prepared by: ESROCOS team
Approved by: M. Frigerio
Authorized by: H. Bruyninckx
Code: ESROCOS_D4.2
Version: 1.1
Date: 25/02/2019

© ESROCOS Consortium 2019, all rights reserved
DOCUMENT STATUS SHEET

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Pages</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>31/08/2018</td>
<td>36</td>
<td>First issue of the document.</td>
</tr>
<tr>
<td>1.1</td>
<td>25/02/2019</td>
<td>36</td>
<td>Document made public per action FRM_A01 of the Final Review Meeting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Added copyright statement.</td>
</tr>
</tbody>
</table>

NOTICE

The contents of this document are the copyright of the ESROCOS Consortium and shall not be copied in whole, in part or otherwise reproduced (whether by photographic, reprographic or any other method) and the contents thereof shall not be divulged to any other person or organisation without the prior written consent of the ESROCOS Consortium. Such consent is hereby automatically given to the European Commission and PERASPERA PSA to use and disseminate.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCUMENT STATUS SHEET</td>
<td>2</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>3</td>
</tr>
<tr>
<td>LIST OF TABLES AND FIGURES</td>
<td>5</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>1.1. PURPOSE</td>
<td>6</td>
</tr>
<tr>
<td>1.2. SCOPE</td>
<td>6</td>
</tr>
<tr>
<td>1.3. CONTENTS OF THIS DOCUMENT</td>
<td>6</td>
</tr>
<tr>
<td>1.4. DEFINITIONS AND ACRONYMS</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1. Definitions</td>
<td>6</td>
</tr>
<tr>
<td>1.4.2. Acronyms</td>
<td>6</td>
</tr>
<tr>
<td>2. REFERENCES</td>
<td>8</td>
</tr>
<tr>
<td>1.5. APPLICABLE DOCUMENTS</td>
<td>8</td>
</tr>
<tr>
<td>1.6. REFERENCE DOCUMENTS</td>
<td>8</td>
</tr>
<tr>
<td>3. CONTEXT</td>
<td>9</td>
</tr>
<tr>
<td>4. OVERVIEW OF THE TOOL</td>
<td>10</td>
</tr>
<tr>
<td>4.1. USAGE</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1. Bootstrap sequence</td>
<td>10</td>
</tr>
<tr>
<td>4.1.2. Example</td>
<td>10</td>
</tr>
<tr>
<td>4.2. CODE REPOSITORIES</td>
<td>11</td>
</tr>
<tr>
<td>5. ROBOT MODEL</td>
<td>12</td>
</tr>
<tr>
<td>5.1. META-MODEL</td>
<td>12</td>
</tr>
<tr>
<td>5.2. DIGITAL FORMAT</td>
<td>12</td>
</tr>
<tr>
<td>5.3. URDF CONVERSION</td>
<td>14</td>
</tr>
<tr>
<td>5.3.1. URDF generator</td>
<td>15</td>
</tr>
<tr>
<td>5.4. LIMITATIONS</td>
<td>15</td>
</tr>
<tr>
<td>6. USER REQUEST MODEL (&quot;QUERY&quot;)</td>
<td>16</td>
</tr>
<tr>
<td>6.1. FORMAT</td>
<td>16</td>
</tr>
<tr>
<td>6.1.1. Heading</td>
<td>16</td>
</tr>
<tr>
<td>6.1.2. Frames</td>
<td>16</td>
</tr>
<tr>
<td>6.1.3. Solvers</td>
<td>16</td>
</tr>
<tr>
<td>6.1.3.1. Forward kinematics</td>
<td>16</td>
</tr>
<tr>
<td>6.1.3.2. Inverse kinematics</td>
<td>17</td>
</tr>
<tr>
<td>6.2. LIMITATIONS</td>
<td>17</td>
</tr>
<tr>
<td>7. GENERATED CODE</td>
<td>19</td>
</tr>
<tr>
<td>7.1. BACKENDS</td>
<td>20</td>
</tr>
<tr>
<td>7.2. USING THE CODE</td>
<td>20</td>
</tr>
<tr>
<td>7.3. TESTS</td>
<td>20</td>
</tr>
<tr>
<td>7.3.1. Numerical tests</td>
<td>20</td>
</tr>
<tr>
<td>7.3.2. Coverage and integrity</td>
<td>21</td>
</tr>
<tr>
<td>7.4. INTEGRATION WITH TASTE</td>
<td>21</td>
</tr>
<tr>
<td>8. EXAMPLE - UR5</td>
<td>22</td>
</tr>
</tbody>
</table>
8.1.CONNECTIVITY AND ORDERING ................................................................. 22
8.2.ZERO CONFIGURATION ......................................................................... 23
8.3.LINK FRAMES ....................................................................................... 23
8.4.JOINT FRAMES ...................................................................................... 24
  8.4.1.Joint status polarity ............................................................................ 25
8.5.GEOMETRICAL PROPERTIES ................................................................. 25
9. AUTOMATIC TYPE CONVERSIONS ............................................................ 27
  9.1.MOTIVATION ......................................................................................... 27
  9.2.DESCRIPTION ......................................................................................... 28
    9.2.1.DProto model .................................................................................... 28
    9.2.2.DProto composition .......................................................................... 29
    9.2.3.Conversion between datatypes by means of dproto models .............. 30
      9.2.3.1.Direct PLAIN Conversion ............................................................. 31
      9.2.3.2.Direct Conversion by alias ........................................................... 31
      9.2.3.3.Direct Conversion by conversion function .................................. 32
      9.2.3.4.Direct composite conversion ....................................................... 33
      9.2.3.5.INDIRECT conversion ................................................................ 34
    9.2.4.Implementation details ...................................................................... 35
  9.3.LIMITATIONS .......................................................................................... 35
LIST OF TABLES AND FIGURES

Table 1-1. Definitions ........................................................................................................... 7
Table 1-2. Acronyms ............................................................................................................. 7
Table 2-1. Applicable Documents ........................................................................................ 8
Table 2-2. Reference Documents ........................................................................................ 8
Table 9-1. Conversion between datatypes by means of dproto models .............................. 30

Figure 4-1. Overview of the code generation tool ................................................................. 10
Figure 7-1. Code generation for interoperability with TASTE ................................................ 21
Figure 8-1. The UR5 robot ..................................................................................................... 22
Figure 8-2. The zero configuration of the UR5 ................................................................. 23
Figure 8-3. The Cartesian coordinate frames for the links of the UR5 ............................. 24
Figure 8-4. The link frames on the assembled UR5 robot, in the zero configuration .......... 24
Figure 8-5. The shoulder-pan joint (in dark) attached to the base link ............................... 25
Figure 8-6. The elbow joint attached to the upper-arm link ............................................... 25
Figure 9-1. Indirect conversion ............................................................................................. 34
1. INTRODUCTION

1.1. PURPOSE

The PERASPERA OG1 activity is devoted to the design of a Robot Control Operating Software (RCOS) that can provide adequate features and performance with space-grade Reliability, Availability, Maintainability and Safety (RAMS) properties. The goal of the ESROCOS project is to provide an open source framework which can assist in the development of flight software for space robots. By providing an open standard which can be used by research labs and industry, it is expected that the Technology Readiness Level (TRL) can be raised more efficiently, and vendor lock-in through proprietary environments can be reduced. Current state-of-the-art robotic frameworks are already addressing some of these key aspects, but mostly fail to deliver the degree of quality expected in the space environment. In the industrial robotics world, manufacturers of robots realise their RCOS by complementing commercial real-time operating systems, with proprietary libraries implementing the extra functions.

The purpose of this Structural Modelling Document is to illustrate the rationale and the usage of the kinematics modeling tool which is part of the ESROCOS toolchain. The tool is basically a code generator, which takes as input a robot model and a declarative model of some desired kinematic solvers.

1.2. SCOPE

This document is an outcome of the WP 4100 "Robot modelling Development" of the ESROCOS project. The ESROCOS framework is a set of tools and software components that support the development of robotics applications with demanding RAMS requirements. It consists of Robot Control Operating System (RCOS) components, and RCOS Development Environment (RDEV) tools.

1.3. CONTENTS OF THIS DOCUMENT

This document contains a description of the code generation tool for kinematics solvers developed by KUL as part of the ESROCOS project. The document provides an overview of the rationale behind the tool, as well as a detailed description of its components and its usage. In fact, the document focuses on usage instructions from the end user’s perspective. A detailed example about modeling the UR5 manipulator is also included.

The document contains the following sections:

- Section 1: Introduction.
- Section 2: References. Lists of applicable and reference documents that are relevant to the structure and contents of this report.
- Section 3: Context. Presents some general information about kinematic solvers and related concepts.
- Section 4: Overview of the Tool. Provides introductory instructions to get started with the tool.
- Section 5: Robot Model. Describes the structure and format of the language used to model the robot.
- Section 6: User Request Model. Describes approach and the format to model queries on the robot kinematics.
- Section 7: Generated Code. Provides information about the code generation process and results.
- Section 8: Example – UR5: Presents an example of the usage of the tool to model a COTS manipulator.
- Section 9: Automatic Type Conversions: Describes the mechanisms for type conversions that enable integration with TASTE.
1.4. DEFINITIONS AND ACRONYMS

1.4.1. DEFINITIONS
Concepts and terms used in this document and needing a definition are included in the following table:

<table>
<thead>
<tr>
<th>Table 1-1. Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept / Term</td>
</tr>
<tr>
<td>-</td>
</tr>
</tbody>
</table>

1.4.2. ACRONYMS
Acronyms used in this document and needing a definition are included in the following table:

<table>
<thead>
<tr>
<th>Table 1-2. Acronyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acronym</td>
</tr>
<tr>
<td>COTS</td>
</tr>
<tr>
<td>ESA</td>
</tr>
<tr>
<td>ESROCOS</td>
</tr>
<tr>
<td>KUL</td>
</tr>
<tr>
<td>RCOS</td>
</tr>
<tr>
<td>RDEV</td>
</tr>
<tr>
<td>SW</td>
</tr>
<tr>
<td>WP</td>
</tr>
</tbody>
</table>
2. REFERENCES

1.5. APPLICABLE DOCUMENTS

The following documents, of the exact issue shown, form part of this document to the extent specified herein. Applicable documents are those referenced in the Contract or approved by the Approval Authority. They are referenced in this document in the form [AD.x]:

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AD.1]</td>
<td>Peraspera: D3.1 Compendium of SRC Activities (for call 1)</td>
<td></td>
</tr>
<tr>
<td>[AD.2]</td>
<td>Guidelines for strategic research cluster on space robotics technologies horizon 2020 space call 2016</td>
<td></td>
</tr>
<tr>
<td>[AD.4]</td>
<td>ESROCOS System Requirements (D1.2) issue 1.1. GMV 20172/17 V2/17</td>
<td>23/06/2017</td>
</tr>
<tr>
<td>[AD.5]</td>
<td>ESROCOS Interface Control Document (D2.3) issue 2.0. GMV 20177/17 V2/17</td>
<td>23/06/2017</td>
</tr>
<tr>
<td>[AD.7]</td>
<td>ESROCOS Unitary Testing Plan</td>
<td>23/06/2017</td>
</tr>
<tr>
<td>[AD.8]</td>
<td>ESROCOS Detailed Design Document</td>
<td>31/01/2018</td>
</tr>
<tr>
<td>[AD.9]</td>
<td>ESROCOS Test and Integration Plan</td>
<td>31/01/2018</td>
</tr>
<tr>
<td>[AD.10]</td>
<td>ESROCOS Software Configuration File</td>
<td>31/01/2018</td>
</tr>
</tbody>
</table>

1.6. REFERENCE DOCUMENTS

The following documents, although not part of this document, amplify or clarify its contents. Reference documents are those not applicable and referenced within this document. They are referenced in this document in the form [RD.x]:

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Title</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>[RD.1]</td>
<td>Composable motion stacks in component-based digital robotic platforms: knowledge models, information architectures, self-X implementations</td>
<td>April 2018</td>
</tr>
<tr>
<td>[RD.2]</td>
<td>KinDSL format description, online: <a href="https://robcogenteam.bitbucket.io/rmodel.html">https://robcogenteam.bitbucket.io/rmodel.html</a></td>
<td>August 2018</td>
</tr>
</tbody>
</table>
3. CONTEXT

A kinematic solver is a numerical algorithm that performs some computations about the geometry of a mechanism. For example, to calculate ("solve") the position and orientation of a specific link of the mechanism, depending on its current configuration. The class of mechanisms we consider is that of "kinematic trees", articulated multi-rigid-body assemblies like industrial manipulators or humanoid robots.

Implementing a solver means transforming a declarative, symbolic model – e.g. a solver for "the position of body B1 with respect to body B2" – to an imperative program in machine language, suitable for actual execution on a digital platform. The implementation requires several choices addressing the freedoms left by the declarative model, as well as the technological options. Only a complete set of such choices uniquely identifies the semantics of a digital implementation.

These so called "grounding choices" include the mathematical representations (e.g. quaternion/rotation matrix), coordinates (e.g. which Cartesian frame to use), units of measure, as well as digital data (data structure, numerical precision), programming language, external libraries.

A limitation of the traditional APIs is the inevitable lack of support for grounding choices different than those taken (knowingly or not!) at design time. On the other hand, a design accounting for the whole combinatorial amount of possibilities is perhaps theoretically possible, but unmaintainable. In addition, the implicit nature of some of the choices makes it harder to use the library correctly. Often, the implicit choices/ambiguities pertain "high-level" semantics and not only "details" like the units of measure; for example, tacitly assuming that "Jacobian" refers to the relative velocity of the end-effector with respect to the base of the robot, when in fact the pair of bodies that uniquely identify a Jacobian is arbitrary.

Software dealing with geometric primitives (thus kinematics and robotics software in general) is particularly affected by these issues.

The purpose of the tool subject of this document, is to automate the implementation process of a solver. The tool is a proof of concept of a methodology, where the traditional implementation process is replaced by code generation, given user-defined models and configuration items that constitute a complete specification; grounding choices are exposed explicitly, granting the user more flexibility and at the same time enforcing awareness.

In its current version, the tool is not really "complete"; it itself still imposes some choices and supports a limited set of options. Other limitations include the support for revolute/prismatic joints only, thus it is not possible to model mechanisms such as the human skeletal structure.

The generated code is very tailored for specific tasks and not composable at all, because composition is envisaged at the level of the models. On the other hand, the generated code is likely to be more efficient than a traditional implementation, as the prior knowledge of the robot model allows to perform some optimizations. In general, the methodology we described is suitable for deployment on systems where the required solvers are well known, possibly with specific constraints about performance, real-time, reliability.
4. OVERVIEW OF THE TOOL

Figure 4-1 shows an overview of the tool described in this document. Given a robot model and a user request, the first component ("ILK generator") generates a model of the requested kinematics solver independent of the choice of programming language or computational hardware ("ILK" stands for Intermediate Language for Kinematics).

Afterwards, this model is “compiled” into actual C/C++ code that can in turn be compiled on the target platform.

The purpose of the intermediate state between the inputs and the C code is to separate the two distinct steps of solver development: devising the pure algorithm and then turning into computer code. The ILK model of the solver makes it possible to develop different compilers to target different languages/environments, without affecting the functionality of the ILK generator. At the moment, C++ (using the Eigen library) is supported.

The other sections of this document describe in more details the inputs and outputs the code generation tool.

4.1. USAGE

Both the generator and the compiler are command line tools taking text files as inputs and producing text files as output.

For building instructions and prerequisites please refer to the readme file of the two tools.

For usage instructions, please also refer to the readme file, as well as the command-line help of the two commands (use the --help switch).

4.1.1. BOOTSTRAP SEQUENCE

The commands sequence for the very first download and configuration of the tool might look like the following:

```bash
# Get the source code...
git clone git@github.com:ESROCOS/kin-gen.git
cd kin-gen/
git submodule update --init

# Now Follow the readme for the ilk-generator and the ilk-compiler, to install
# the dependencies.
# ...

# Build the ilk-generator ...
cd ilk-generator/
made
```
# Set up the environment

cd ..
export KIN_GEN_ROOT=`pwd`
export PATH=$PATH:$KIN_GEN_ROOT
eval `luarocks path`

4.1.2. EXAMPLE

For your reference, a typical usage of the two commands could be:

```
ilk-generator.sh --robot-model <modelFile> --query <queryFile> --output-dir <ILK srcs dir>
ilk-compiler.sh -b eigen -indir <ILK srcs dir> --outdir <C++ srcs dir>
```

# move to <C++ sources dir>
make

4.2. CODE REPOSITORIES

These are the URL of the Git repositories with the source code of the tool. The first is the git-superproject, containing the two components (ilk-generator and ilk-compiler) as submodules:

https://github.com/ESROCOS/kin-gen.git

The individual repositories are:

ilk-generator:

https://github.com/ESROCOS/ilk-generator.git

ilk-compiler:

https://github.com/ESROCOS/ilk-compiler.git
5. ROBOT MODEL

The primary input of the code generation tool is a robot model, i.e. a description of the robot for which code has to be generated.

5.1. META-MODEL

The required robot model conforms to the **kinematic-tree** meta-model widely used in robotics: robots are mechanisms formed by **rigid bodies** connected by joints, which are really motion constraints; the topology of the mechanism is that of a tree, i.e. a graph with no loops (kinematic loops are currently not supported). The robot model format supports prismatic and revolute joints, which suffice to represent the vast majority of articulated robots such as manipulators and grippers. Another entity in the meta-model is the Cartesian reference frame; frames are placed on the rigid bodies as the basis to **measure** the geometrical and inertial properties of the mechanism (frames and transformations conform in fact to the meta-model described in [RD.1]).

5.2. DIGITAL FORMAT

The concrete, digital format of a robot model was created for another tool, and it is best documented in the web page [RD.2]. In short, it is a text file with a custom syntax, with a “paragraph” for each link and joint of the robot. The file format is called “Kinematics-DSL” (**KinDSL** in short).

Section 8. below illustrates the step by step procedure required to model the UR5 robot, according to the documentation cited above. For reference, though, here is the complete model:

```plaintext
Robot ur5
{
    RobotBase base {
        inertia_properties {
            mass = 4.0
            CoM = (0.000000,0.000000,0.025000)
            lx = 0.006930
            ly = 0.006930
            lz = 0.007200
            lxz = 0.000000
            lyz = 0.000000
        }
        children {
            shoulder via shoulder_pan
        }
    }
    link shoulder {
        id = 1
        inertia_properties {
            mass = 3.7
            CoM = (0.000000,0.001930,-0.025610)
            lx = 0.012710
            ly = 0.012690
            lz = 0.006670
            lxz = 0.000000
            lyz = 0.000000
        }
        children {
            upper_arm via shoulder_lift
        }
    }
    link upper_arm {
        id = 2
        inertia_properties {
            mass = 8.393
            CoM = (0.000000,-0.024200,0.212500)
        }
    }
}
```
\[ \begin{align*}
  I_x &= 0.609370 \\
  I_y &= 0.604450 \\
  I_z &= 0.017150 \\
  I_{xy} &= -0.000000 \\
  I_{xz} &= 0.000000 \\
  I_{yz} &= -0.043160 \\
\end{align*} \]

```json
} children {
  forearm via elbow
}
}

link forearm {
  id = 3
  inertia_properties {
    mass = 2.275
    CoM = (0.000000, 0.026500, 0.119930)
    I_x = 0.082630
    I_y = 0.081030
    I_z = 0.003420
    I_{xy} = 0.000000
    I_{xz} = 0.000000
    I_{yz} = 0.007230
  }
  children {
    wrist_1 via wr1
  }
}

link wrist_1 {
  id = 4
  inertia_properties {
    mass = 1.219
    CoM = (0.000000, 0.110950, 0.016340)
    I_x = 0.017410
    I_y = 0.002410
    I_z = 0.016240
    I_{xy} = 0.000000
    I_{xz} = 0.000000
    I_{yz} = 0.002210
  }
  children {
    wrist_2 via wr2
  }
}

link wrist_2 {
  id = 5
  inertia_properties {
    mass = 1.219
    CoM = (0.000000, 0.110950, 0.110990)
    I_x = 0.017100
    I_y = 0.017100
    I_z = 0.001240
    I_{xy} = 0.000000
    I_{xz} = 0.000000
    I_{yz} = 0.000240
  }
  children {
    wrist_3 via wr3
  }
}

link wrist_3 {
  id = 6
  inertia_properties {
    mass = 0.1879
    CoM = (0.000000, 0.001160, 0.000000)
    I_x = 0.000320
  }
}
```
Note that the format also includes the inertia properties of the rigid links; however, inertia is currently not used by the ESROCOS KUL code generation tool, which deals only with kinematics.

5.3. URDF CONVERSION

The URDF is a robot model format that became popular because of being part of the ROS ecosystem. There exist some support for URDF also in the realm of CAD programs for mechanical design\(^1\), whose integration with existing modeling tool(chain) is very desirable. The kinematics subset of the COLLADA file format, originally selected for such purpose in the context of ESROCOS, does not seem to have good software support nor to be available as an export format in CAD programs.

\(^1\) An add-on for SolidWorks, from the ROS wiki: [http://wiki.ros.org/sw_urdf_exporter](http://wiki.ros.org/sw_urdf_exporter). A converter from the SimScape XML: [https://github.com/robotology/simmechanics-to-urdf](https://github.com/robotology/simmechanics-to-urdf) (there are commercial add-ons for some CAD programs to export to the SimScape format).
It is therefore convenient to have an automated conversion between the URDF and the KinDSL described in this document. The conversion is doable since (a subset of) the URDF format accounts for basically the same information of the KinDSL; however it is not possible to guarantee 100% accuracy in all circumstances due to some differences between the two formats².

The converter tool is hosted in the following git repository:

https://mfrigerio17@bitbucket.org/robcogenteam/urdf2kindsl.git

which is anyway included (as a submodule) in the main repository of the tool, cited above. For usage information, please refer to the command line help, accessible with:

./urdf2kindsl.py --help

5.3.1. URDF GENERATOR

The conversion in the opposite direction (KinDSL to URDF) is also possible. A dedicated option of the ilk-generator allows to convert the input robot model into URDF (rather than parsing the query and generating ILK source code, the default behaviour). Run the command like this:

ilk-generator.sh --robot-model <input KinDSL> --gen-urdf <output URDF>

5.4. LIMITATIONS

The KinDSL robot-model format accounts for prismatic or revolute joints only, and it does not support (yet) kinematic loops.

The model does not support heterogeneous units of measure; no specific systems of units is enforced, but all the numerical values must conform to one system only, for the generated code to be correct (for example, if the joint frame translation is expressed in meters, the moment of inertia should be in meters squared times the unit for inertia). In fact, this point might be seen as an optimization of the format, to relieve the user from specifying the unit of each numerical value.

² For example, the URDF allows to have fixed joints, the KinDSL does not. On the other hand, fixed joints are typically used to add dummy links for the sole purpose of attaching more frames to the model (every link has a frame by default). The KinDSL, instead, supports explicitly the attachment of additional frames. When encountering a fixed joint in the URDF, it is impossible to determine automatically whether it is there to induce another frame or for some other reason (e.g. to model the rigid attachment of a load, whose inertia might change often and it is thus convenient to keep it separate from the robot link).
6. USER REQUEST MODEL ("QUERY")

The second input file for the tool is the "query" file, a sort of configuration file specifying the code to be generated. This is also a text file, with a simple format. For example:

```plaintext
Robot ur5
Frames {
  fr_base, fr_forearm, fr_wrist_3, fr_elbow
}
FK fk1 {
  pose : fr_wrist_3 wrt fr_base
  pose : fr_wrist_3 wrt fr_elbow
  Jacobian : fr_wrist_3 wrt fr_base
}
FK fk2 {
  pose : fr_forearm wrt fr_base
}
IKvel ik1 {
  vect : linear
  frames: fr_wrist_3 wrt fr_base
}
IKpos ik2 {
  vect : pose
  frames: fr_wrist_3 wrt fr_base
}
```

6.1. FORMAT

6.1.1. HEADING

The file must start with a declaration of the robot it refers to, which must correspond to the name appearing in the robot model.

6.1.2. FRAMES

The Frames block lists the reference frames that will be used in the rest of the query. For each link and joint of the robot model, a corresponding reference frame is implicitly defined (see [RD.2]); the name template for frames is `fr_<link name>` or `fr_<joint name>`.

The physical location of the link-frames and joint-frames on the actual bodies of the robot is described in detail in the documentation of the robot model format [RD.2].

Note that the robot model allows to "attach" more named reference frames to any link, explicitly; such frames can be referenced in the query by their name (without any arbitrary "fr_").

6.1.3. SOLVERS

The rest of the query consists in a list of requests for the actual code generation. Currently, code can be generated for **Forward** and **Inverse Kinematics** solvers. Each solver in the query will result in a separate function in the generated code.

6.1.3.1. FORWARD KINEMATICS

The schema for a FK block is the following:

```plaintext
FK <user name for the solver>
{
  
}
Besides the name, the specification consists of a possibly empty sequence of desired relative poses and Jacobians. A relative pose is defined by a pair of frame names <target> and <reference>, meaning “the pose of target with respect to reference”. The same holds for Jacobians, although the definition would read as “the Jacobian for the relative velocity of target with respect to reference”.

The input for any FK solver is implicitly the joint status vector of the robot.

### 6.1.3.2. INVERSE KINEMATICS

Two different keywords exist for solvers at the velocity level or at the position level. The specification is however the same for both cases, and in includes the type of vectors of interest (translational, rotational, or full poses) and again a pair of reference frames. The schema for the text block is the following:

\[
\text{IK}\{\text{vel}|\text{pos}\} <\text{user name}> \\
\{ \\
\quad \text{vect} : \{\text{linear} \mid \text{angular} \mid \text{pose}\} \\
\quad \text{frames} : \text{<target> wrt <reference>}
\}
\]

So, for example, the text:

```
IKvel ik1 \\
\{ \\
\quad \text{vect} : \text{linear} \\
\quad \text{frames} : \text{fr_wrist_3 wrt fr_base}
\}
```

reads as: “generate an inverse kinematics solver for the linear velocity of the frame of wrist3 with respect to the frame of the base (obviously “wrist3” and “base” are just labels that are not subject of any semantic interpretation – their “meaning” comes from the definition of the same labels in the robot model). The other example:

```
IKpos ik2 \\
\{ \\
\quad \text{vect} : \text{pose} \\
\quad \text{frames} : \text{fr_wrist_3 wrt fr_base}
\}
```

models an IK solver at the position level, for the full position plus orientation (pose) of the frame wrist3 relative to the base frame.

The inputs for an IK solver are the desired position/orientation of the chosen frame, depending on the kind of solver which has been requested. The generated code will differ slightly depending on the case. These inputs are implicit and need not to appear in the query. For the first example above, the input is the Euclidean 3d vector expressing the linear Velocity of fr_wrist_3 relative to fr_base. For the second example, it is the relative pose between the two frames.

Note that, as opposed to forward kinematics, the specification of each solver contains no lists but only individual items. This difference simply reflects the different nature of the problem: forward kinematics allows to compute multiple quantities while sweeping over the kinematic tree; conversely, it makes no sense to solve inverse kinematics for two different frames during the same iterations.

### 6.2. LIMITATIONS

At the moment, the main limitations of the query language (and of the code generation tool itself) lie in the implicit assumptions/choices that we are still making. For example, it is not configurable nor explicit that the frame used for the coordinates is the frame that we called “reference”. For example, given a relative velocity vector defined by two frames, the basis for the corresponding coordinate vector could be yet another, third, frame; however, we currently use the frame “reference”.

ESROCOS © ESROCOS Consortium 2019, all rights reserved Structural Modelling Document
In the case of relative poses, we are currently imposing the choice of representation via homogeneous transformation matrices, which themselves impose the constraint of being expressed with coordinates of the frame "reference".

These choices are probably the most common in robotics software, yet they are not mandatory and it should be possible to configure the tool differently.
7. GENERATED CODE

As explained in Section 4., the tool generates a model of the requested solvers in a custom language (called ILK), and this gets in turn compiled into C/C++. At the moment, the ILK language is only used internally and it does not need to be exposed to the end user. Thus, this section shall focus on the usable C code which is generated in the last step.

The main generated files in the output directory (after execution of the ilk-compiler – see Section 4.1), are four: a Makefile, a header with common definitions about the robot, and the header/source pair with the actual code of the solvers. All the symbols are contained in a namespace named after the robot name.

For each solver request in the query input file (see Section 6.) a different function will be generated, named after the name given in the query itself. Any number of solvers can be generated, that is, the user can choose the list of functions to be generated. For forward kinematics, a single solver can actually compute multiple quantities (e.g. two relative poses and a Jacobian – obviously for the same joint status vector); the net effect is a quite flexible selection of the generated API.

For example, this is the main header file resulting from the query showed before, for the UR5 robot model:

```c
#ifndef _ILK_GEN_UR5_H_
#define _ILK_GEN_UR5_H_

#include <ilk/eigen/core-types.h>
#include <ilk/eigen/joint-transforms.h>
#include <ilk/eigen/ik.h>
#include "robot-defs.h"

namespace ur5 {

struct mc_config {
    mc_config();
    kul::pose_t fr_elbow__fr_upper_arm;
    kul::pose_t fr_shoulder_lift__fr_shoulder;
    kul::pose_t fr_shoulder_pan__fr_base;
    kul::pose_t fr_wr1__fr_forearm;
    kul::pose_t fr_wr2__fr_wrist_1;
    kul::pose_t fr_wr3__fr_wrist_2;
};

void fk1(const mc_config& mc, const ur5::joint_state& input,
    kul::pose_t fr_wrist_3__fr_base,
    kul::pose_t fr_wrist_3__fr_elbow,
    ur5::t_J_fr_wrist_3_fr_base& J_fr_wrist_3_fr_base);

void fk2(const mc_config& mc, const ur5::joint_state& input,
    kul::pose_t fr_forearm__fr_base);

void ik1(const mc_config& mc, const ur5::joint_state& q,
    const kul::vector3_t &vector,
    ur5::joint_state& q_guess);

void ik2(const mc_config& mc, const kul::ik_pos_cfg& cfg,
    const kul::vector3_t &desired_position,
    const kul::rot_m_t &desired_orientation,
    const ur5::joint_state& q_guess,
    ur5::joint_state& q);

void fk__ik1(const mc_config& mc, const ur5::joint_state& input,
    kul::pose_t fr_wrist_3__fr_base,
    ur5::t_J_fr_wrist_3_fr_base& J_fr_wrist_3_fr_base);
}
#endif //! _ILK_GEN_UR5_H_
```

The `struct` at the beginning is the container of the geometrical constants of the robot, in the form of constant poses (those relating the joint frame relative to the link frame supporting that joint). The generated functions (`fk1, ik2, etc.`) reflect the request in the query. For example, the first function has three output arguments (non-const C++ references) as requested (two relative poses and a Jacobian).

The last function (`fk__ik1`) is not a user’s request, it is another forward solver which is required in turn by the inverse solvers, and thus it has been generated automatically.
7.1. BACKENDS

In principle, the tool supports compilation of the ILK models into different target languages/platforms. In general, there would be a set of non-generated source code for each of such targets, containing some common facilities like data types. We call such a set a backend.

At the moment, the only target supported by the ILK compiler is C++ in combination with the Eigen library; therefore, the compiler is shipped with what we call the "Eigen backend", a small library to be built and installed in the target machine in order to compile the generated code. This is the library that defines most of the types visible in the code snippet above, such as kul::pose_t. For further information please refer to the readme file of the ilk-compiler repository.

7.2. USING THE CODE

Build the generated code simply by using make, after installing the backed (see above). The command will create a shared library with the symbols of the generated solvers. In the following, a simple example about how to invoke one of the generated solvers:

```cpp
#include <iostream>
#include <ilk/eigen/core-types.h>
#include "ur5.h"

static mc_config constants;

int main(int argc, char** argv)
{
    // define the variables for the outputs
    kul::pose_t wrist3__wrt__base;
    kul::pose_t wrist3__wrt__elbow;
    ur5::t_J_fr_wrist_3_fr_base J_wrist3__base;

    // define and initialize the joint status vector
    ur5::joint_state q;
    q.setZero();

    // call the solver
    ur5::fk1(constants, q, wrist3__wrt__base, wrist3__wrt__elbow, J_wrist3__base);
    std::cout << wrist3__wrt__base << std::endl;
    std::cout << wrist3__wrt__elbow << std::endl;
    std::cout << J_wrist3__base << std::endl;
    return 0;
}
```

7.3. TESTS

The ilk-compiler generates some simple tests along with the solver themselves. These tests are typically command line programs that somehow invoke the solvers to test them; in practice, the compiler creates additional source files with the main() function and some relate Makefile recipes.

In general, one or more tests are generated for each solver requested in the query, that is, for each function of the generated "library".

7.3.1. NUMERICAL TESTS

A generated test for forward kinematics reads a dataset with the joint configuration vector and then a sequence of the relative poses that the solver is supposed to compute. The expected results are compared with those computed by the generated solver, and the total, average residual is printed finally on standard output. The dataset is currently in a trivial text format; the test expects to see a joint status vector followed by as many poses as returned by the solver; for further details please see the documentation of the TextDataset class in test-utils.h in the folder ilk-compiler/backend/eigen/.

A generated test for inverse kinematics is instead standalone (no external dataset needed), as it verifies the consistency between forward and inverse kinematics, starting from a random joint status
vector. The design and code generation of more exhaustive tests for inverse kinematics is part of the ongoing developments.

7.3.2. COVERAGE AND INTEGRITY

The generated Makefile contains additional rules for the same/additional test source code, to build alternative binaries for debugging and code analysis. These binaries are functionally equivalent to the numerical tests mentioned above, but can be run in combination with tools like Valgrind and GNU gcov. The first can detect several common memory problems, whereas the second can generate code coverage reports.

7.4. INTEGRATION WITH TASTE

The integration with a TASTE-based application is envisaged as follows: alongside with the actual code for the kinematics solvers, the tool will generate some additional C glue code. Primarily, a header and source pair, resembling what would be generated by TASTE itself from a function block designed in the UI; the PI (Provided Interface) of this block will match the user query, i.e. it will exhibit a function for each requested solver. The generated source file will invoke in turn the actual solver providing the functionality. It will also contain the appropriate conversion functions between the ASN.1 data types (appearing in the signatures of the PI functions) and the data types used internally by the solver. See Figure 7-1.

![Figure 7-1. Code generation for interoperability with TASTE](link_to_image)

The code generation for TASTE integration is currently hosted in another repository: [https://github.com/ESROCOS/kin-gen-taste.git](https://github.com/ESROCOS/kin-gen-taste.git)

The tool requires two configuration files, `config.yaml` and `metadata.yaml`. The first one has to be manually customized, the second can be generated with the appropriate option (`--generate-metadata`) of the ilk-compiler. This configuration item is the mechanism to pass the information about the generated solver (such as the API) to the TASTE-generator.
8. EXAMPLE - UR5

This section walks the reader through the procedure to model the UR5 manipulator with the KinDSL format used by the code generation tool. As mentioned above, please refer to [RD.2] for the official documentation of the format; here, we will follow the procedure described on the webpage, for the specific case of the UR5.

8.1. CONNECTIVITY AND ORDERING

Figure 8-1 shows an image of the complete UR 5 robot. There are 7 rigid bodies connected by 6 revolute joints; it is a linear chain with no branches. It makes most sense, intuitively, to order the links from the base to the last link of the wrist articulation (top of the figure).

![Figure 8-1. The UR5 robot](image)

Therefore, the skeleton of the robot model would look like the following:

```plaintext
Robot ur5
{
  RobotBase base {
    children {
      shoulder via shoulder_pan
    }
  }
  link shoulder {
    id = 1
    children {
      upper_arm via shoulder_lift
    }
  }
  link upper_arm {
    id = 2
    children {
      forearm via elbow
    }
  }
  link forearm {
    id = 3
    children {
      wrist_1 via wr1
    }
  }
  link wrist_1 {
    id = 4
    children {
      wrist_2 via wr2
    }
  }
  link wrist_2 {
```

Note that this text is not (yet) a valid document as it is incomplete, yet it reflects the connectivity of the UR5 and the ordering that we chose. Note the numeric link id property, which need not appear in the base; in fact, this field is redundant for kinematic chains, as the numbering would be implied by the choice of the base and the parent-child relationships. However, in the presence of kinematics branches (e.g., the paws of a quadruped robot) the numbering is not unique, therefore the robot model format allows the user to choose. Thus the presence of the link id property.

8.2. ZERO CONFIGURATION

The zero configuration is the configuration of the robot which is conventionally represented with the zero joint status vector. It is entirely conventional and up to the user. We shall choose the configuration illustrated in Figure 8-2.

![Figure 8-2. The zero configuration of the UR5](image)

The choice of the zero configuration has no explicit effect on the model, therefore nothing has to be explicitly added to it.

8.3. LINK FRAMES

The next step of modeling is choosing the coordinate frames for each link of the robot, which will be used to measure all the quantities. The model format imposes the constraint of having the Z axis of the link-frame along the axis of the joint that is moving the link. Otherwise, the choice is free and up to the user.

Typically, but not necessarily, the frames would be chosen to minimize the number of rigid motions to go from one frame to the next one, although this might become clear only from the following sections and from experience.3

The frame of a fixed-base link, which is not moved by any joint, can be placed anywhere with respect to the link itself.

Figure illustrates the location of the frames we picked for our example. This step is also a purely conceptual, and no data has to be written yet into the model.

3 There is a sort of circular dependency between the zero configuration, the choice of the link frames, and the joint frame parameters that is impossible to illustrate with a linear, sequential description.
8.4. JOINT FRAMES

Consider the robot assembled, frozen in the zero configuration as illustrated in Figure 8-4.

Figure 8-4. The link frames on the assembled UR5 robot, in the zero configuration

For each joint of the robot, consider a new frame currently coinciding with the frame of the successor link; for example, for the shoulder-pan joint (the first joint, which rotates about the vertical axis) the frame coincides with the shoulder-link frame, in blue in the figure. The new frame is actually “attached” to the predecessor link (the robot base link, in our example).
Figure 8-5 illustrates this example, whereas Figure 8-6 shows the case of the elbow-joint frame attached to the upper-arm link.

Figure 8-5. The shoulder-pan joint (in dark) attached to the base link

Figure 8-6. The elbow joint attached to the upper-arm link

Note that any joint frame is fixed onto the link it is attached to; e.g. the shoulder-pan frame is fixed with respect to the base frame, their relative pose is a geometrical constant of the robot. When the joint moves, the successor link moves and so does its frame, but the frame of the joint itself is stationary (relatively to its link).

8.4.1. JOINT STATUS POLARITY

The Z axis of the joint frame is the joint axis. The right-hand rule for rotations determines the positive rotation direction, i.e. the direction in which the joint status variable increases its value. If one wants to swap this polarity, the Z axis must be turned to point to the opposite versus. This implies in turn to change the orientation of the link frame that specify the joint frame of interest (see above).

For prismatic joints, the positive translation direction coincide of course with the versus of the Z axis.

8.5. GEOMETRICAL PROPERTIES

It is now finally possible to measure the geometry of the robot. The model must contain all the relative poses in the form "joint frame with respect to link frame", which are all constant. The robot model format currently supports only one way of representing a relative pose, which is a translation vector and a triplet of Euler angles, representing three successive, intrinsic rotations about the X, Y and Z axis. The rotations are applied after the translation. The coordinates for this quantities are always of the link frame.

For example, the shoulder-pan joint is simply translated along the Z axis of the base frame, for 8.9 cm; the corresponding section of the model will then look like this:

```plaintext
r_joint shoulder_pan {
    ref_frame {
        translation = (0.0, 0.0, 0.089159)
        rotation = (0.0, 0.0, 0.0)
    }
}
```

The shoulder-lift joint frame, instead, is translated and rotated with respect to its carrier link, the shoulder (cf. Figure 8-5). The text of the model looks like the following:

```plaintext
r_joint shoulder_lift {
    ref_frame {
        translation = (0.135850, 0.0, 0.0)
        rotation = (0.0, PI/2.0, -PI/2.0)
    }
}
```
After filling in the values for all joints, the geometrical model is complete. At this point one would have to insert the inertia properties of the links, but we can skip that step as inertia is not used by the code generation tool.
9. AUTOMATIC TYPE CONVERSIONS

This section describes an additional tool being developed in the context of the ESROCOS project, for the generation of code performing type conversions. The tool will be integrated with the rest of the toolchain described before, in particular with the code generator for the TASTE integration.

9.1. MOTIVATION

Composing a robotic application with model-driven technologies requires several steps to embed a certain functionality in a software component, often leading to tedious and error-prone manual programming of the so-called “glue-code”.

ESROCOS tools already solve this partially, by means of the TASTE framework. TASTE adopts the AADL architectural language to describe the component interfaces and the composition of components that realizes the full software solution. In the same vein, TASTE uses ASN.1 as Interface Description Language (IDL) to specify the data exchanged among components (also called “Communication Object”) which in turn are transformed into compliant C data structures used in the concrete realization.

However, the user-code to embed in a component never uses the generated C data structures from ASN.1 models, because it was designed in full isolation from the component-based framework, or because the user-datatypes are more efficient for the realization of the functionality.

Besides, ASN.1 regards the digital data representation of the data, i.e., how the values are serialized in memory. ASN.1 does not have notion of the meaning of the stored values, which are instead interpreted by the software developer. For example, a ASN.1 type named "Position" which contains a tuple of floating values named "x", "y", and "z" (independently from the precision of the floating point value) has an easy interpretation for a skilled programmer in the domain of robotics, mechanics or physics, but it does not have any semantic meaning to the C compiler. In addition, ambiguity may arise even for skilled programmers. For example, it is hard to interpret an ASN.1 type named "Quaternion" defined as a list of four unnamed values ("SEQUENCE(SIZE(1..4)) OF T-Double"); in such case, the scalar value of the quaternion (usually indicated with the symbol "w"), could be the first or the last of the list. The same holds for the imaginary values of the quaternion: while the order is usually fixed, the naming convention is not, e.g., tuples \( <a,b,c> \) and \( <x,y,z> \).

Currently, the only semantic check between datatypes is the one performed by the compiler of typed languages (such as C/C++), which validates whether the data used as argument of a function is of the same datatype of the one expected in the signature.

In summary, there is a need for annotating and enriching the used datatypes in the implementation, whether they are modelled/generated from an IDL or not, in order to perform sanity checks on the semantics of the values, the choices of representation and eventually to perform automatic conversion between semantically compatible types.

The term “dproto-tool” in this document refers to a reference implementation of an automatic datatype converter (https://github.com/haianos/dconv-tool) that performs the conversion among semantically-complete datatypes descriptors.

The main goals and features are:

- static code generation: generate the operations to be performed for the requested data conversion;
- a common API based on the algebraic representation of the datatype, thus a semantic-centric accessors for the code generator hooks;
- multi-domain: the implementation enables the support of different domains. The current implementation focuses on the geometric domain, (geometric primitives) as shown in this document. See "Limitations" section for further details;
- Support for different IDLs.

The original purpose of this tool was to automate the glue-code generation necessary to embed the code generated by the kinematic solver generator in a TASTE component.

However, given the relevance of the tackled problem, the tool has been designed and developed as a stand-alone facility both for the conversion between custom and IDL-generated datatypes, and for the integration of heterogeneous functionalities within the same software component.
9.2. DESCRIPTION

9.2.1. DPROTO MODEL

A dproto is a semantically-complete model, an annotated version of a datatype expressed with an IDL or into a concrete programming language.

<table>
<thead>
<tr>
<th>Snippet of a dproto model &quot;Base_Position&quot; relative to an ASN.1 model (on the right)</th>
<th>ASN.1 model that the dproto &quot;Base_Position&quot; annotates (on the left), and its concrete generated C-type</th>
</tr>
</thead>
</table>
| dproto Base_Position :: geometric {
  semantic = Position
  coord = cartesian
  algebraic = position3_named
  ddr = {
    mid = "Base-Types.Wrappers-Vector3d"
    mmid = "ASN1"
  }
  dr = {0=x, 1=y, 2=z}
} | //ASN1 model
Wrappers-Vector3d ::= SEQUENCE
{ data SEQUENCE(SIZE(1 .. 3)) OF T-Double }
// C-code
typedef struct {
  int nCount;
  T_Double arr[3];
} Wrappers_Vector3d_data;

Keywords reference:

- **Semantic**: a valid semantic tag defined in the domain of reference (e.g., "geometric");

- **Coordinate Representation (coord)**: valid annotation of the coordinate representation choice of the (geometric) primitive for the semantic in subject, e.g., "cartesian" and "polar" if the semantic of the datatype is "Position";

- **Algebraic representation (algebraic)**:
  The choice of an algebraic representation is modeled with the keyword *algebraic* as reported below, and they can be of type (list of) Scalar, Vector or Matrix.
  - Scalar: a list of scalars, referred to by means of a set of symbols defined in the algebraic definition;
  - Vector: a vector with an integer index starting from 0. The definition must include the max size of the vector.
  - Matrix: a matrix with row and column indices, as in $M(r,c)$. The definition must include the max size of the matrix (rows, columns).

Concrete examples of user-defined algebraic definitions are reported below.

| algebraic position3_named :: Scalar{x,y,z} |
| algebraic position3 :: Vector{3} |
| algebraic rot_mx :: Matrix{3,3} |

- **Digital data representation (ddr)**:
  - **mmid**: meta-model identifier of the digital data representation. Currently supported values are ['ASN1', 'Eigen', 'c99'] (‘Eigen’ types are used internally in the kinematic solver generation tool);
  - **mid**: model (or model identifier) of the concrete datatype description, conforming to the indicated mmid. In this case, the value in the example refers to an ASN.1 module called "Base-Types" and the definition identifier is “Wrappers-Vector3d”;

- **Data representation (dr)**: this is a relation between the accessors used in the digital data representation (ddr, left-hand side) and the algebraic representation symbols (right-hand side). Accessors are chained with “dot notation” syntax. In the example above "Base_Position", the dproto is modeled as Scalar{x,y,z}, and accessing to the value ‘x’ is equivalent to access to the concrete referred datatype generated by the ASN.1 model “Wrappers-Vector3d” with index “0”, that is myval.x returns myval.arr[0].
Note that the constraints determining whether a dproto is well-formed depend on the domain definition. For example, the choice a coordinate representation (field coord) must be indicated within this domain. However, other domains may require different semantic information.

### 9.2.2. DPROTO COMPOSITION

A dproto definition can be expressed as a composition of existing dproto models. For example:

```plaintext
dproto Base_Pose :: geometric {
    semantic = Pose
    composes = {
        Position  = Base_Position
        Orientation = Base_Quaternion
    }
    dr = {
        orientation = Orientation
        position    = Position
    }
}
```

The composition above is semantically correct within the domain of reference “geometric”, if the concept of `Pose` is modeled explicitly as a composition of `Position` and `Orientation`:

```plaintext
semantic Pose = composition_of[Position, Orientation]
```

The keyword “composes” indicates the relation between the semantic composition (left-hand side of the relation) and the dproto used as reference model (right-hand side), with the modeling constraints that the semantics of the indicated dproto model matches with the left-hand side value. Algebraic representation and coordinate representation are inherited from the dproto of reference.

The digital representation (dr) determines the relation between the named accessor of the datatype and the semantic component. Therefore, `myvar.position` will refer to the value (data instance) conforming to a `Base_Position` dproto model, where `myvar` is a variable name associated to an instance of `Base_Pose` dproto model.

However, the composition above is not the only way to describe a dproto with `Pose` semantic. In fact, it is still possible to define a dproto with `Pose` semantic, as a whole, regardless of the composite nature of the concept of “pose”. For instance, in the case of the representation by homogeneous transformation matrix:

```plaintext
dproto kul_pose :: geometric {
    semantic  = Pose
    coord     = homogeneous_transformation
    algebraic = ht_matrix
    ddr = {
        mid  = "Matrix<4,4>"
        mmid = "Eigen"
    }
    dr = {
        {0,0}={0,0}, {0,1}={0,1}, {0,2}={0,2}, {0,3}={0,3},
        {1,0}={1,0}, {1,1}={1,1}, {1,2}={1,2}, {1,3}={1,3},
        {2,0}={2,0}, {2,1}={2,1}, {2,2}={2,2}, {2,3}={2,3},
        {3,0}={3,0}, {3,1}={3,1}, {3,2}={3,2}, {3,3}={3,3}
    }
}
```

In addition, because the concept of Pose is a composition at the semantic level, it is possible to define views over the `kul_pose` dproto. A view is a relation between the “internals” of the dproto as a whole, and another dproto that fully expresses part of the composite dproto, possible by means of composition
rules at semantic level. The mapping relation is expressed using the algebraic representation of the dproto model, as shown below.

<table>
<thead>
<tr>
<th>View relationship definitions about kul_pose</th>
<th>dproto models used in the view definitions reported on the left.</th>
</tr>
</thead>
<tbody>
<tr>
<td>view kul_pose.Position -&gt; kul_position {</td>
<td>dproto kul_position :: geometric {</td>
</tr>
<tr>
<td>{0,3} = x, {1,3} = y, {2,3} = z</td>
<td>semantic = Position</td>
</tr>
<tr>
<td>}</td>
<td>coord = cartesian</td>
</tr>
<tr>
<td>view kul_pose.Orientation -&gt; kul_rotation2</td>
<td>algebraic = position3_named</td>
</tr>
<tr>
<td>{0,0} = Xx, {0,1} = Yx, {0,2} = Zx,</td>
<td>ddr = {</td>
</tr>
<tr>
<td>{1,0} = Yx, {1,1} = Yy, {1,2} = Yz,</td>
<td>mid = &quot;Matrix&lt;3,1&gt;&quot;</td>
</tr>
<tr>
<td>{2,0} = Zx, {2,1} = Zy, {2,2} = Zz</td>
<td>mmid = &quot;Eigen&quot;</td>
</tr>
<tr>
<td>}</td>
<td>dr = {</td>
</tr>
<tr>
<td></td>
<td>{0,0} = x, {1,0} = y, {2,0} = z</td>
</tr>
<tr>
<td>//algebraic definitions used in the</td>
<td>}</td>
</tr>
<tr>
<td>//dproto definitions (on the right)</td>
<td></td>
</tr>
<tr>
<td>algebraic orient_rot_named ::</td>
<td>dproto kul_rotation2 :: geometric {</td>
</tr>
<tr>
<td>Scalar{Xx,Yx,Zx,Yy,Yz,Zx,Zy,Zz}</td>
<td>semantic = Orientation</td>
</tr>
<tr>
<td>algebraic position3_named :: Scalar{x,y,z}</td>
<td>coord = rot_matrix</td>
</tr>
<tr>
<td></td>
<td>algebraic = orient_rot_named</td>
</tr>
<tr>
<td></td>
<td>ddr = {</td>
</tr>
<tr>
<td></td>
<td>mid = &quot;Matrix&lt;3,3&gt;&quot;</td>
</tr>
<tr>
<td></td>
<td>mmid = &quot;Eigen&quot;</td>
</tr>
<tr>
<td></td>
<td>dr = {</td>
</tr>
<tr>
<td></td>
<td>{0,0} = Xx, {0,1} = Yx, {0,2} = Zx,</td>
</tr>
<tr>
<td></td>
<td>{1,0} = Yx, {1,1} = Yy, {1,2} = Yz,</td>
</tr>
<tr>
<td></td>
<td>{2,0} = Zx, {2,1} = Zy, {2,2} = Zz</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Finally, a dproto can be defined as nested composition of other composite dproto as well. Examples of this are reported in the following paragraphs.

9.2.3. CONVERSION BETWEEN DATATYPES BY MEANS OF DPROTO MODELS

This section explains the different types of conversion that the dconv-tool supports, resumed in the following table.

<table>
<thead>
<tr>
<th>Conversion type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct plain conversion</td>
<td>Same semantic, coordinate representation and algebraic.</td>
</tr>
<tr>
<td>Direct conversion by alias</td>
<td>Same semantic and coordinate representation, different algebraic and alias rule.</td>
</tr>
<tr>
<td>Direct conversion by conversion function</td>
<td>Same semantic, different coordinate representation and an external conversion function rule.</td>
</tr>
<tr>
<td>Direct composite conversion</td>
<td>One of the above, but the dproto are a composition of others dprotos.</td>
</tr>
<tr>
<td>Indirect conversion</td>
<td>Conversion performed indirectly, by means of a set of third dprotos.</td>
</tr>
</tbody>
</table>

9.2.3.1. DIRECT PLAIN CONVERSION

A direct plain conversion is possible when both the source and the target have the same semantics, coordinate representation and algebraic representation. In such cases, there is a direct mapping.
between the dproto models, and the only difference is at the level of the concrete datatype in the implementation. As an example, let us consider the conversion between Base_Position and kul_position dproto models:

<table>
<thead>
<tr>
<th>Dproto model example of Base_Position, used as source subject of the conversion</th>
<th>Dproto model example of kul_position, used as target subject of the conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>dproto Base_Position :: geometric {</td>
<td>dproto kul_position :: geometric {</td>
</tr>
<tr>
<td>semantic = Position</td>
<td>semantic = Position</td>
</tr>
<tr>
<td>coord = cartesian</td>
<td>coord = cartesian</td>
</tr>
<tr>
<td>algebraic = position3_named</td>
<td>algebraic = position3_named</td>
</tr>
<tr>
<td>ddr = {</td>
<td>ddr = {</td>
</tr>
<tr>
<td></td>
<td>mid = &quot;Base-Types.Wrappers-Vector3d&quot;</td>
</tr>
<tr>
<td></td>
<td>mmid = &quot;ASN1&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>dr = {</td>
</tr>
<tr>
<td></td>
<td>0=x, 1=y, 2=z</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

The statically generated (C/C++) body code of the conversion function is the following:

```c
lhs(0,0) = rhs.data.arr[0];
lhs(1,0) = rhs.data.arr[1];
lhs(2,0) = rhs.data.arr[2];
```

where:
- `rhs` is the input variable of the type associated to the Base_Position dproto (i.e., the C-data structure generated by the ASN.1 compiler by means of the definition Wrappers-Vector3d contained in Base-Types module)
- `lhs` is the output variable of the type associated to the kul_position dproto (i.e., an Eigen type/class Eigen::Matrix<double,3,1>).

The code is generated by means of generated accessors functions based on the digital representation (dr) relation, with "dot notation" syntax (cf. dproto model and generalised API for algebraic values).

### 9.2.3.2. DIRECT CONVERSION BY ALIAS

A direct conversion by alias occurs when the source and the target have the same semantic and coordinate representation but different algebraic representation. In this cases, if further knowledge has not been modeled, a request of conversion will fail.

It is therefore possible to model an alias relation that expresses the equivalence of the semantic content of the numerical values, by mapping two different algebraic representation definitions.

Let us consider a conversion from a new dproto model kul_position2 (reported below) and the Base_Position dproto of the previous example. In this case, the algebraic representation does not match, but the conversion still happens by exploiting the knowledge of the alias relationship, which maps the named scalar values to an index-based vector representation.

<table>
<thead>
<tr>
<th>Dproto model example of kul_position2, used as target subject of the conversion</th>
<th>algebraic definitions and and alias relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>dproto kul_position2 :: geometric {</td>
<td>algebraic position3 :: Vector{3}</td>
</tr>
<tr>
<td>semantic = Position</td>
<td>algebraic position3_named :: Scalar{x,y,z}</td>
</tr>
<tr>
<td>coord = cartesian</td>
<td>alias position3_named = position3 {</td>
</tr>
<tr>
<td>algebraic = position3</td>
<td>x = 0</td>
</tr>
<tr>
<td>ddr = {</td>
<td>y = 1</td>
</tr>
<tr>
<td></td>
<td>z = 2</td>
</tr>
</tbody>
</table>
The statically generated (C/C++) body code of the conversion function is the following:

```cpp
lhs.data.arr[0] = rhs(0,0);
lhs.data.arr[1] = rhs(1,0);
lhs.data.arr[2] = rhs(2,0);
```

It is important to note that an alias relation can also map algebraic of the same type, but with different symbols. This feature is very useful to guarantee the compatibility among semantic-equivalent dproto models using different naming conventions. Differences in the character case is a typical example:

```
algebraic position3_named :: Scalar{x,y,z}
algebraic position3_named_alternative :: Scalar{X,Y,Z}

alias position3_named = position3_named_alternative {x=X, y=Y, z=Z}
```

Therefore, the alias indicates a full equivalence between the defined algebraic representation, where only the symbolic name has changed.

Another example is using the alias for mapping a different element ordering in vectors:

```
algebraic position3 :: Vector{3}
algebraic position3_alternative :: Vector{3}

alias position3 = position3_alternative { 0=2, 1=1, 2=0}
```

9.2.3.3. DIRECT CONVERSION BY CONVERSION FUNCTION

A conversion is a feature of the language that allows to express the existence of a unidirectional conversion function between a dproto model and another. For example:

```
conversion QUAT -> ROT = quat2rot
```

This is a necessary feature to allow conversion of semantically-equivalent data models, but with different coordinate representation. In all this cases, the conversion is only possible if a corresponding implementation exists; in some cases the solution of the conversion is not necessarily unique, thus it must be treated with particular care.

The dconv-tool simply generates the modeled (C-)function code:

```
quat2rot(src,tgt);
```

Furthermore, this mechanism allows to create any user-defined conversion not directly supported by the dconv-tool, thus avoiding (by design) possible reference implementation shortcomings.
9.2.3.4. DIRECT COMPOSITE CONVERSION

A direct composite conversion is a direct conversion between two dproto models, where both are composed by other dproto definitions. As an example, we defined a nested composition, and the conversion from SuperPose to kul_superpose is performed.

<table>
<thead>
<tr>
<th>SuperPose dproto model, composed of a pair that must match Base_Pose dproto model definition (reported below.)</th>
<th>kul_superpose model, composed of a pair of kul_pose2 dproto model definition (reported below.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dproto SuperPose :: geometric {</td>
<td>dproto kul_superpose :: geometric {</td>
</tr>
<tr>
<td>semantic = PairPose</td>
<td>semantic = PairPose</td>
</tr>
<tr>
<td>composes = {</td>
<td>composes = {</td>
</tr>
<tr>
<td>Obj1 = Base_Pose</td>
<td>Obj1 = kul_pose2</td>
</tr>
<tr>
<td>Obj2 = Base_Pose</td>
<td>Obj2 = kul_pose2</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>dr = {</td>
<td>}</td>
</tr>
<tr>
<td>bar = Obj1</td>
<td>o1 = Obj1</td>
</tr>
<tr>
<td>foo = Obj2</td>
<td>o2 = Obj2</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Base_Pose dproto model, composition of Base_Position and Base_Quaternion</th>
<th>kul_pose2 dproto model, composition of kul_position and kul_quaternion</th>
</tr>
</thead>
<tbody>
<tr>
<td>dproto Base_Pose :: geometric {</td>
<td>dproto kul_pose2 :: geometric {</td>
</tr>
<tr>
<td>semantic = Pose</td>
<td>semantic = Pose</td>
</tr>
<tr>
<td>composes = {</td>
<td>composes = {</td>
</tr>
<tr>
<td>Position = Base_Position</td>
<td>Position = kul_position</td>
</tr>
<tr>
<td>Orientation = Base_Quaternion</td>
<td>Orientation = kul_quaternion</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>dr = {</td>
<td>}</td>
</tr>
<tr>
<td>orientation = Orientation</td>
<td>position = Position</td>
</tr>
<tr>
<td>position = Position</td>
<td>orientation = Orientation</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

The generated body code follows.

```c
// rhs, variable input conformant to SuperPose dproto model
// (C-struct generated from ASN1 definition)
// lsh, variable output conformant to kul_superpose dproto model
// (POD struct having a pair of Eigen::Matrix<double,4,4>)

lhs.o1.orientation(0) = rhs.bar.orientation.im.arr[0];
lhs.o1.orientation(1) = rhs.bar.orientation.im.arr[1];
lhs.o1.orientation(2) = rhs.bar.orientation.im.arr[2];
lhs.o1.orientation(3) = rhs.bar.orientation.re;
lhs.o1.position(0,0)  = rhs.bar.position.data.arr[0];
lhs.o1.position(1,0)  = rhs.bar.position.data.arr[1];
lhs.o1.position(2,0)  = rhs.bar.position.data.arr[2];
lhs.o2.orientation(0) = rhs.foo.orientation.im.arr[0];
lhs.o2.orientation(1) = rhs.foo.orientation.im.arr[1];
lhs.o2.orientation(2) = rhs.foo.orientation.im.arr[2];
lhs.o2.orientation(3) = rhs.foo.orientation.re;
lhs.o2.position(0,0)  = rhs.foo.position.data.arr[0];
lhs.o2.position(1,0)  = rhs.foo.position.data.arr[1];
lhs.o2.position(2,0)  = rhs.foo.position.data.arr[2];
```
9.2.3.5. INDIRECT CONVERSION

An indirect conversion is a conversion requiring finite, longer than one sequence of conversion steps of any of the previous listed types.

For the sake of brevity, the full dproto models definitions are not shown. However, the picture below illustrates an indirect conversion from Base_Quaternion (linked to an ASN.1 model) to kul_rotation (an Eigen type Matrix<double,3,3> used in the internals of the kinematic solver generator).

![Indirect conversion diagram]

In this case, the semantic of the data model is equivalent (the values represent an orientation), but the coordinate representation differs. In our example, a conversion relation does not exist in the knowledge space defined by the set of dproto definitions. However, Base_Quaternion can be converted to QUAT (corresponding to the plain C type double[4]), which in turn is associated to a conversion function (as shown in a previous example) towards ROT. Since ROT can be turned into a kul_rotation by means of a direct plain conversion, it turns out that the Base_Quaternion dproto is convertible to a kul_rotation dproto with the sequence of conversions illustrated above.

The generated body code for such transformation is reported below.

```cpp
//lhs→output variable, of type kul_rotation (Eigen::Matrix<double,3,3>)
//rhs→input variable, of type Base_Quaternion (C-struct generated with ASN1 model)

double QUAT_tmp_2[4];
double ROT_tmp_3[9];

QUAT_tmp_2[0] = rhs.im.arr[0];
QUAT_tmp_2[1] = rhs.im.arr[1];
QUAT_tmp_2[2] = rhs.im.arr[2];
QUAT_tmp_2[3] = rhs.re;
quat2rot(QUAT_tmp_2,ROT_tmp_3);

lhs(0,0) = ROT_tmp_3[0];
lhs(0,1) = ROT_tmp_3[1];
lhs(0,2) = ROT_tmp_3[2];
lhs(1,0) = ROT_tmp_3[3];
lhs(1,1) = ROT_tmp_3[4];
lhs(1,2) = ROT_tmp_3[5];
lhs(2,0) = ROT_tmp_3[6];
lhs(2,1) = ROT_tmp_3[7];
lhs(2,2) = ROT_tmp_3[8];
```
Obviously, performing an indirect conversion has a (deterministic) computational cost, directly related to the type and the number of conversion steps required. Furthermore, it implies the usage of temporary memory blocks (allocated in the stack) to store intermediate results.

In case indirect conversions are to be avoided in a concrete application, it is suggested to provide and model directly a conversion function from-to the dproto models subjects of the conversion.

More information, tests and examples can be found in the readme of the software repository. The few examples reported here are also used, among others, for unit testing purposes of the dconv-tool.

9.2.4. IMPLEMENTATION DETAILS

The concrete reference implementation has been developed as Lua a module. We chose the Lua language for a few reasons: easy interoperability with other languages; easy to embed within other tools or applications. Furthermore, the ilk-compiler is also implemented in the Lua language. The module is initialised taking as input ASN.1 models and a dproto file definition, and it exposes two high-level functions that are needed to the developer:

- `SDBLX`: generates the static generator and accessors of the indicated dproto. This function is typically not used directly by the casual developer, but it can be used to improve or implement alternative conversion functions.

- `convert[from,to]`: this function takes as input the identifier (name) of the dproto for which a directed conversion is requested. The result of this call is yet another function of type `generator(fd,<from-id>,<to-id>)`. The generator streams to the file descriptor (fd) the generated code, while `<from-id>` and `<to-id>` are strings that should contain the name of the variable, instance of the datatype referred by the dproto models “from” and “to”.

Please refer to the `readme.md`, tests and examples in the software repository for more details.

9.3. LIMITATIONS

At the time of writing, the reference implementation of “dconv-tool” still suffers of few limitations, which are shortly reported below. However, many of them will be solved in the future developments.

Moreover, support for the concrete “dconv-tool” implementation is extended to the natural duration of the ESROCOS project, and that includes bug fixes, enhancements and new features.

The known limitations are:

- `reconfigurability`: reconfigurability is still rather limited with respect the original vision. For example, the current version targets C-code generation, but other languages can also be supported;

- Reconfigurability is also limited to the domain of reference “geometric” of this work. This could be improved by allowing the user to define a schema of reference, thus defining other domains without the need (or with limited) amount of code in the dconv-tool itself;

- The tool has been primary designed to support geometric primitives used in the kinematic generator. However, the language dproto descriptors already provides the facilities to extend this to different domains;

- support to physical unit conversion for numerical values: this is still ongoing implementation efforts, not mature yet;

- Inference for complex, indirect conversions: the current solver implements a simple heuristic which does not guarantee to find the shortest solution, leading to the generation of non-minimal code.

Regardless of the limitations of the current implementation, and the usage context defined by TASTE and the kinematic solver generator, the dconv tool is an interesting outcome of the project providing hints for future research: the structural modelling primitives, and the methodology, with particular attention to the semantic attachment of concrete datatypes employed in the implementation.