Geometric Relations between Rigid Bodies (Part 2)
From Semantics to Software

Tinne De Laet, Steven Bellens, Herman Bruyninckx, and Joris De Schutter

October 5, 2012

***Abstract* - Rigid bodies are essential primitives in the modelling of robotic devices, tasks, and perception. Basic geometric relations between rigid bodies include relative position, orientation, pose, linear velocity, angular velocity, twist, force, torque, and wrench. In Part 1, we explicitly stated the semantics of all coordinate-invariant properties and operations, and, more importantly, all the choices that are made in coordinate representations of these geometric relations. This resulted in a set of concrete suggestions for standardizing terminology and notation. In this tutorial, we show how the proposed semantics allow us to write fully unambiguous software interfaces, including automatic checks for semantic correctness of all geometric operations on rigid-body coordinate representations.**

1 Introduction

A main characteristic of robotics is that it involves three-dimensional motion of rigid bodies (manipulated objects, robot links, mobile bases, ...). In Part 1 [3], we explicitly stated the semantics of geometric relations between rigid bodies (relative position, orientation, pose, linear velocity, angular velocity, twist, force, torque, and wrench). This resulted in a set of concrete suggestions for standardizing terminology and notation.

In this tutorial, we show how the proposed semantics allow us to write fully unambiguous software interfaces, including automatic checks for semantic correctness of all geometric operations on rigid-body coordinate representations. As a teaser, box "Common error 1: Logic errors in geometric relation calculations" shows how one of the common errors listed in Part 1 can be prevented using the software presented in this tutorial.

This tutorial is supplementary to the software web page [2], containing a more detailed explanation on the C++ libraries implementing the geometric semantics for rigid body calculations.

Following the same pattern as Part 1, this tutorial uses a running example to explain the different aspects. First, four use cases involving manipulations of geometric relations between rigid bodies are proposed. Each of the use cases is illustrated using the running example. Next, a software design implementing the semantics for geometric relations between rigid bodies and inspired by the different use cases is proposed. Using the running example, a particular C++ implementation of this design referred to as "geometric_relations_semantics" [2] is demonstrated on the running example for all four use cases. The geometric_relations_semantics library provides semantic checking for calculations with geometric relations between rigid bodies on top of existing geometric libraries, which only work on specific coordinate representations. To our knowledge the proposed software is the first to offer a semantic interface for geometric operation software libraries.

It is assumed that the readers are familiar with the notation and basic concepts of the semantic representation of geometric relations and the corresponding semantic operations introduced in Part 1 [3].

2 Running Example

In this tutorial we use a running example from robotics to illustrate how the software helps to perform semantic checking for geometric relations between rigid bodies. The running example involves a robot that spray paints a cylindrical object, as illustrated in Figure 1. The cylindrical object is fixed in the environment, while the robot holds the spray gun.

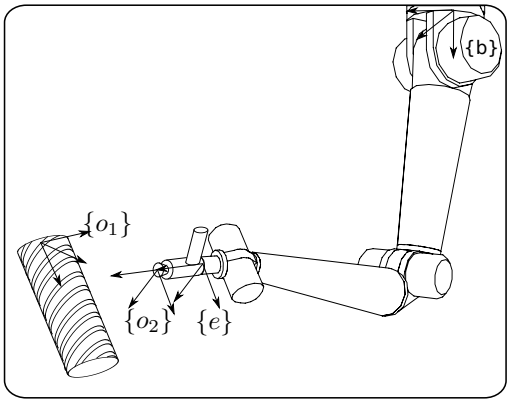

Figure 1: Running example of robot performing spray painting operation

Common error 1: Logic errors in geometric relation calculations

A lot of logic errors can occur during geometric relation calculations. For instance the inverse of Position $(e|\mathcal{C}, f|\mathcal{D})$ is Position $(f|\mathcal{D}, e|\mathcal{C})$ (point and reference point switch) while the inverse of LinearVelocity $(e|\mathcal{C}, \mathcal{D})$ is LinearVelocity $(e|\mathcal{D}, \mathcal{C})$ (point does not change). A second example appears when composing the relations involving three rigid bodies: in order to get the geometric relation of body $\mathcal{C}$ with respect to body $\mathcal{D}$, one can compose the geometric relation between $\mathcal{C}$ and a third body $\mathcal{E}$ with the geometric relation between body $\mathcal{E}$ and body $\mathcal{D}$ (and not the geometric relation between body $\mathcal{D}$ and body $\mathcal{E}$ for instance).

The C++ code in Listing 1 and the program output in Listing 2 show how the software proposed in this tutorial prevents each of the two above logic errors.

Listing 1: Logic errors in geometric calculations - C++

```
// Inversion
PositionSemantics pos("e","C","f","D");
PositionSemantics pos_inverse = pos.inverse();
LinearVelocitySemantics linVel("e","C","D");
LinearVelocitySemantics linVel_inverse = linVel.↩
    inverse();
// Composition
PoseSemantics pose1("g","g","C","h","h","D");
PoseSemantics pose2("i","i","E","h","h","D");
PoseSemantics pose_composition =compose(pose1,pose2);
```

Listing 2: Logic errors in geometric calculations - Output

```
// Inversion
Result: Inverse of Position(e|C,f|D) is Position(f|D,e|C)
Result: Inverse of LinearVelocity(e|C,D) is LinearVelocity(e|D,C)
// Composition
Semantic output: Composition of Pose((g,[g])|C,(h,[h])|D) and Pose((i,[i])|E,(h,[h])|D) is NOK since:
  * Either the reference point, reference orientation frame, and reference body of Pose((g,[g])|C,(h,[h])|D) ↩
      have to be equal to the point, orientation frame, and body of Pose((i,[i])|E,(h,[h])|D) respectively
  OR
  the point, orientation frame, and body of Pose((g,[g])|C,(h,[h])|D) have to be equal to the reference point, ↩
      reference orientation frame, and reference body of  Pose((i,[i])|E,(h,[h])|D) respectively.
```

The running example does not aim at showing all the possible errors the geometric semantics software can prevent but rather to give a robotics show case of the software[1].

To complete the painting task, the robot program has to determine the joint angles of the robot holding the spray gun such that a predefined relative pose between the spray gun and cylindrical object is obtained.

The first step when solving geometry problems consists of identifying the rigid bodies and the frames attached to them. In the running example we have: $\{b\}$ attached to the base $\mathcal{B}$ of the robot, $\{e\}$ attached to the end-effector $\mathcal{E}$ of the robot, $\{o_2\}$ attached to the spray gun $\mathcal{O}_2$, and $\{o_1\}$ attached to cylindrical object $\mathcal{O}_1$. In the example the following poses are available: PoseCoord $(\{o_1\}|\mathcal{O}_1, \{b\}|\mathcal{B}, [b])$ determined by the mounting of the cylindrical object with respect to the robot base, PoseCoord $(\{o_2\}|\mathcal{O}_2, \{e\}|\mathcal{E}, [e])$ determined by the mounting of the spray gun on the robot end-effector, and PoseCoord $(\{o_2\}|\mathcal{O}_2, \{o_1\}|\mathcal{O}_1, [o_1])$ determined by the desired relative spray-painting pose.

In order to find the joint angles of the robot, the robot programmer has to find the pose of the robot end-effector with respect to the base, and then use the inverse kinematics of the robot. The solution of this problem has the following outline: (1) invert Pose $(\{o_2\}|\mathcal{O}_2, \{e\}|\mathcal{E})$ to obtain the pose of the end-effector with respect to the spray gun, Pose $(\{e\}|\mathcal{E}, \{o_2\}|\mathcal{O}_2)$; (2) compose Pose $(\{e\}|\mathcal{E}, \{o_2\}|\mathcal{O}_2)$ and Pose $(\{o_2\}|\mathcal{O}_2, \{o_1\}|\mathcal{O}_1)$ to obtain the pose of the robot end-effector with respect to the cylindrical object, Pose $(\{e\}|\mathcal{E}, \{o_1\}|\mathcal{O}_1)$; and (3) compose Pose $(\{e\}|\mathcal{E}, \{o_1\}|\mathcal{O}_1)$ and Pose $(\{o_1\}|\mathcal{O}_1, \{b\}|\mathcal{B})$ to obtain the pose of the robot end-effector with respect to the robot base, Pose $(\{e\}|\mathcal{E}, \{b\}|\mathcal{B})$. Finally, the inverse kinematics of the robot is used to calculate the joint angles from the end-effector pose with respect to the base, Pose $(\{e\}|\mathcal{E}, \{b\}|\mathcal{B})$.

3 Use Cases

Regarding calculations with geometric relations between rigid bodies four use cases are relevant. We explain and illustrate them using the running example below.

3.1 Coordinate Calculations

The first use case is the one that involves *numerical calculations using particular coordinate representations*, but without semantic checking. This is the level all available geometry libraries such as KDL (Kinematics and Dynamics Library) [5], ROS (Robot Operating System) geometry stack [4], and RL (Robotics Library) [1] work on.

[1] Section 6 contains more show cases on how the presented software can prevent common errors in geometric calculations.

Box 1: Different use cases for running example

1. **Coordinate Calculations**

$$ {}^{\{e\}|\mathcal{E}}_{\{o_2\}|\mathcal{O}_2}\boldsymbol{T} = {}^{\{o_2\}|\mathcal{O}_2}_{\{e\}|\mathcal{E}}\boldsymbol{T}^{-1} \quad (1a) $$

$$ {}^{\{e\}|\mathcal{E}}_{\{o_1\}|\mathcal{O}_1}\boldsymbol{T}, = {}^{\{o_2\}|\mathcal{O}_2}_{\{o_1\}|\mathcal{O}_1}\boldsymbol{T} \; {}^{\{e\}|\mathcal{E}}_{\{o_2\}|\mathcal{O}_2}\boldsymbol{T}, \quad (1b) $$

$$ {}^{\{e\}|\mathcal{E}}_{\{b\}|\mathcal{B}}\boldsymbol{T} = {}^{\{o_1\}|\mathcal{O}_1}_{\{b\}|\mathcal{B}}\boldsymbol{T} \; {}^{\{e\}|\mathcal{E}}_{\{o_1\}|\mathcal{O}_1}\boldsymbol{T}, \quad (1c) $$

2. **Semantics Checking**

$$ \text{Pose}\left(\{e\}\,|\mathcal{E},\{o_2\}\,|\mathcal{O}_2\right) = \text{inverse}(\text{Pose}\left(\{o_2\}\,|\mathcal{O}_2,\{e\}\,|\mathcal{E}\right)), \quad (2a) $$

$$ \text{Pose}\left(\{e\}\,|\mathcal{E},\{o_1\}\,|\mathcal{O}_1\right) = \text{compose}\left(\text{Pose}\left(\{e\}\,|\mathcal{E},\{o_2\}\,|\mathcal{O}_2\right),\text{Pose}\left(\{o_2\}\,|\mathcal{O}_2,\{o_1\}\,|\mathcal{O}_1\right)\right), \quad (2b) $$

$$ \text{Pose}\left(\{e\}\,|\mathcal{E},\{b\}\,|\mathcal{B}\right) = \text{compose}\left(\text{Pose}\left(\{e\}\,|\mathcal{E},\{o_1\}\,|\mathcal{O}_1\right),\text{Pose}\left(\{o_1\}\,|\mathcal{O}_1,\{b\}\,|\mathcal{B}\right)\right). \quad (2c) $$

3. **Coordinate Semantics Checking**

$$ \text{PoseCoord}\left(\{e\}\,|\mathcal{E},\{o_2\}\,|\mathcal{O}_2,[o_2]\right) = \text{inverse2}(\text{PoseCoord}\left(\{o_2\}\,|\mathcal{O}_2,\{e\}\,|\mathcal{E},[e]\right)), \quad (3a) $$

$$ \text{PoseCoord}\left(\{e\}\,|\mathcal{E},\{o_1\}\,|\mathcal{O}_1,[o_1]\right) = \text{compose}\left(\text{PoseCoord}\left(\{e\}\,|\mathcal{E},\{o_2\}\,|\mathcal{O}_2,[o_2]\right),\text{PoseCoord}\left(\{o_2\}\,|\mathcal{O}_2,\{o_1\}\,|\mathcal{O}_1,[o_1]\right)\right), \quad (3b) $$

$$ \text{PoseCoord}\left(\{e\}\,|\mathcal{E},\{b\}\,|\mathcal{B},[b]\right) = \text{compose}\left(\text{PoseCoord}\left(\{e\}\,|\mathcal{E},\{o_1\}\,|\mathcal{O}_1,[o_1]\right),\text{PoseCoord}\left(\{o_1\}\,|\mathcal{O}_1,\{b\}\,|\mathcal{B},[b]\right)\right). \quad (3c) $$

4. **Coordinate Calculations and Coordinate Semantics Checking**
Coordinate calculations using Equations (1a)-(1c) and coordinate semantics checking using Equations (3a)-(3c). Additionally, the semantic constraints imposed by particular coordinate representations are checked (see Part 1).

For the running example, one option is to use the 4×4 homogeneous transformation matrix [6] ${}^{\{g\}|\mathcal{C}}_{\{h\}|\mathcal{D}}\boldsymbol{T}$ as coordinate representation for the pose PoseCoord $(\{g\}\,|\mathcal{C},\{h\}\,|\mathcal{D},[h])$. In this case, the inverse and the compositions needed in the solution outline (Section 2) are obtained by doing a matrix inversion and matrix multiplications, respectively, as shown in Box 1 “Different use cases for running example”.

3.2 Semantics Checking

The second use case is the one that involves *semantic checking*, but that performs abstraction of the actual coordinate frames and of the particular coordinate representations. In this case the semantics of the geometric relations concerning the point, orientation frame, body, reference point, reference orientation frame, and reference body are used and checked explicitly. Furthermore, the semantics of the solution of every operation can be automatically inferred. This semantics checking use case is especially useful for checking if the solution outline that a programmer has in mind is correct, and if all the needed geometric information is available.

For the running example (see Box 1 “Different use cases for running example”), the semantics of the poses resulting from the inverse and the composition can be automatically inferred from the arguments. Furthermore, the composition operations can be checked semantically, by checking if the frame and body of one pose in the composition are equal to the reference frame and reference body, respectively, of the other pose in the composition.

3.3 Coordinate Semantics Checking

The third use case is the one that involves *coordinate semantics checking*, but that performs abstraction of the particular coordinate representations. In this case the semantics of the geometric relations concerning the point, orientation frame, body, reference point, reference orientation frame, reference body, *and* coordinate frame are used and checked explicitly. As for the semantics checking use case, the coordinate semantics of the solution of every operation can be automatically inferred. But additionally to the semantics checking use case, this use case also considers the semantics of the coordinate frame. The coordinate semantics checking use case is, just as the semantics checking use case, useful for checking if the solution outline that a programmer has in mind is correct (including coordinate frame checking), and if all the needed geometric information is available.

For the running example (see Box 1 “Different use cases for running example”), the coordinate semantics of the poses resulting from the inverse and the composition can be automatically inferred from the arguments. Furthermore, the composition operations can be checked semantically, by checking (additionally with respect to the semantics checking use case) if the coordinate frame of the arguments equals the reference coordinate frame of the poses.

3.4 Coordinate Calculations and Coordinate Semantics Checking

The fourth use case *combines coordinate semantics checking with numerical calculations using particular coordinate representations*. In this case not only the semantics of the

geometric relations concerning the point, orientation frame, body, reference point, reference orientation frame, reference body, *and* coordinate frame are used and checked explicitly; additionally the semantic constraints imposed by particular coordinate representations are checked (see Part 1). Finally, the numerical calculations using particular coordinate representations are automatically generated from the semantic equations.

For the running example (see Box 1 "Different use cases for running example"), as explained in the previous use case, the coordinate semantics of the poses resulting from the inverse and the composition can be automatically inferred from the arguments and the composition operations can be checked semantically. Additionally, the semantic constraints imposed by the homogeneous transformation matrix (the point and coordinate frame belong to the same frame, the reference point and reference orientation frame belong to the same reference frame, and the coordinate frame is equal to the reference orientation frame) can be checked. Finally, the numerical operations of the particular coordinate representations (see Box 1 Eq. (1a)-(1c)) can be automatically obtained from the semantic equations (see Box 1 Eq. (3a)-(3c)): when using homogeneous transformation matrices, the inverse operator is replaced by a matrix inverse and the pose compositions are replaced by matrix multiplications (where the multiplication order can be derived from the semantics of the poses).

4 General Software Design

The goal of the software is to implement the geometric relation semantics proposed in Part 1 [3], offering support for semantic checks for rigid body relations. This will avoid commonly made errors, and hence reduce application (and, especially, system integration) development time considerably. This section presents the general software design that is inspired by the four use cases defined in Section 3.

4.1 Design ideas

The design of software providing semantic checking for calculations with geometric relations between rigid bodies has three important requirements: (1) it has to accommodate the four use cases discussed before (Section 3), (2) it has to be built on top of existing geometric libraries already providing support for geometric calculations on specific coordinate representations, and (3) it has to allow for efficient code at runtime. These three design criteria are discussed in more detail below.

To **accommodate the four use cases**, the software has to allow for calculations only covering numerical operations on particular coordinate representations, (coordinate) semantics checking, or both. This can be achieved by constructing multiple software entities, each directly related to one of the use cases, on which all kinds of geometric relation operations can be performed.

The goal of the software is to provide semantic checking for calculations with geometric relations between rigid bodies **on top of existing geometry libraries**, since there are already a lot of libraries with good support for geometric calculations on specific coordinate representations. Furthermore, the design should be such that the effort to extend an existing geometry library with semantic support is very limited. Therefore, the design idea is to create a (templated) core library for the semantic checking providing all the necessary semantic support for geometric relations (relative positions, orientations, poses, linear velocities, angular velocities, twists, forces, torques, and wrenches) and the operations on these geometric relations (composition, integration, inversion, ...). If a programmer wants to extend the geometry libraries he uses with semantic checking, the programmer can build a new library, depending on the core library, in which only a limited number of functions have to be implemented. These functions make the connection between semantic operations (for instance composition) and actual coordinate representation calculations (for instance multiplication of homogeneous transformation matrices).

The software still has to allow for **efficient code at runtime**, which is of particular importance in robotics applications. To this end the programmer has to be able to enable or disable the semantic checking depending on the actual use case. For instance, when developing new software for an application, there is the need to perform semantic checking for all geometric calculations. After this development phase, however, there is no further need to check the "internal" geometric calculations. It is however still important to check the "interface" of the developed software when integrating it into a bigger application.

4.2 Design

For every geometric relation (position, orientation, pose, linear velocity, angular velocity, twist, force, torque, and wrench) the core library contains four classes. Here we will explain the design with the position geometric relation. All other geometric relations have a similar design. For the position geometric relation the four classes are:

- **PositionCoordinates**<**PositionT**>: This templated class contains the actual coordinate representation of the geometric relation, for instance a position vector. The template is the actual geometry object (of an existing geometry library) that will be used as a coordinate representation, for instance a KDL::Vector.
- **PositionSemantics**: This class contains the semantics of the (coordinate-free) Position geometric relation. For instance in this case it contains the informa-

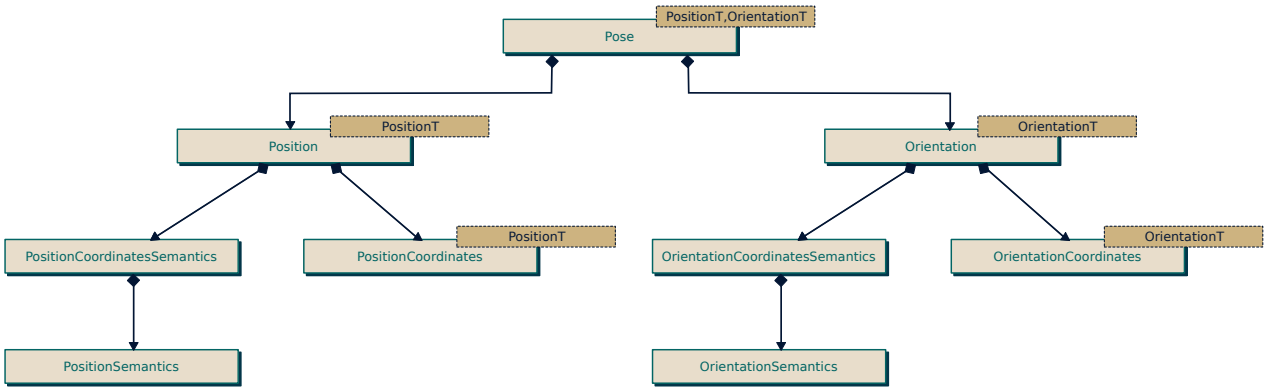

Figure 2: Pose<PositionT,OrientationT> UML diagram

tion on the point, reference point, body, and reference body.

- **PositionCoordinatesSemantics**: This class contains a PositionSemantics object of the geometric relation at hand and the extra semantic information concerning the coordinate frame in which the coordinates are expressed.

- **Position<PositionT>**: This templated class is a composition of a PositionCoordinatesSemantics object and a PositionCoordinates object. Therefore, it contains both the information of the actual coordinate representation and the semantics. Again, the template is the actual coordinate representation (of an existing geometry library), for instance a KDL::Vector.

Note that all four of the above 'levels' directly correspond to the four use cases. The UML diagram in Figure 3 shows the design based on composition outlined above.

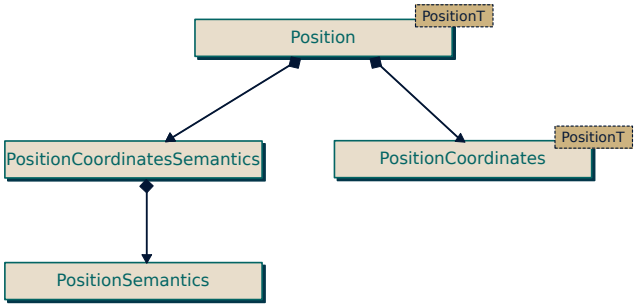

Figure 3: Position<PositionT> UML diagram

The geometric relations pose, twist, and wrench need an additional representation since they can, beside being represented as a single geometric entity, as explained above, also be represented as a composition of two other geometric relations (Pose = Position + Orientation, Twist = LinearVelocity + AngularVelocity, Wrench = Force + Torque). For example, in one case we want to use a homogeneous transformation matrix as a coordinate representation of a pose, and in this case we also want, for efficiency reasons, to do direct calculations on the homogeneous transformation matrices. In another case we want to represent the pose as the composition of a position (with for instance a position vector as a coordinate representation) and an orientation (with for instance Euler angles as a coordinate representation). The UML diagram in Figure 2 illustrates this design using the composition of two other geometric relations.

5 Using Geometric Relations Semantics Software

This section shows how the software design proposed in Section 4 facilitates the four use cases for calculations with geometric relations proposed in Section 3, using the running example presented in Section 2. To this end a C++ implementation of the above described design, which is publicly and freely available under de LGPLv2.1/BSD open-source license [2], is used to present code fragments that implement the solution outlined in Section 2 for all four use cases. While this paper only shows the screen output (which can be turned off), the actual C++ implementation additionally returns a boolean value indicating if the geometric operation was semantically correct or not. This way, semantic errors can be captured and appropriately taken care of. In the code fragments, the kinematic and dynamic library KDL [5] is used as the geometry library providing a particular coordinate representation.

5.1 Coordinate Calculations

As explained in Section 3.1, the first use case works directly at the coordinate level without using semantic information.

Listing 1 and Listing 2 show the C++ code and the program output of the running example, respectively, for the first use case. The particular coordinate representation is the homogeneous transformation matrix, in this case implemented using a KDL::Frame. The operations inverse

(line 14) and compose (lines 16 and 18) are directly translated to coordinate calculations available in KDL, in this case a matrix inversion and matrix multiplication. While the numerical output is valuable (Listing 2), there is no way to check if it is actually correct. For instance, are the KDL::Frames multiplied in the correct order or does it even make semantic sense to compose these two?

Listing 1: Coordinate Calculations

```
1 //Construction of KDL coordinates
2 Rotation coordRotO1_B = Rotation::EulerZYX(0,0,0);
3 Vector coordPosO1_B(-0.5,0.5,0.5);
4 Rotation coordRotO2_E = Rotation::EulerZYX(0,0,0);
5 Vector coordPosO2_E(0,0,0.1);
6 Rotation coordRotO2_O1 = Rotation::EulerZYX(0,-M_PI↩
      /2,0);
7 Vector coordPosO2_O1(0.5,0,0.0);
8
9 //Construction of poses
10 PoseCoordinates<KDL::Frame> poseO1_B(KDL::Frame(↩
      coordRotO1_B,coordPosO1_B));
11 PoseCoordinates<KDL::Frame> poseO2_E(KDL::Frame(↩
      coordRotO2_E,coordPosO2_E));
12 PoseCoordinates<KDL::Frame> poseO2_O1(KDL::Frame(↩
      coordRotO2_O1,coordPosO2_O1));
13 //Step1: Determine poseE_O2 by inverting poseO2_E
14 PoseCoordinates<KDL::Frame> poseE_O2 = poseO2_E.↩
      inverse();
15 //Step2: Determine poseE_O1 by composing poseE_O2 and↩
       poseO2_O1
16 PoseCoordinates<KDL::Frame> poseE_O1 = compose(↩
      poseO2_O1,poseE_O2);
17 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
18 PoseCoordinates<KDL::Frame> poseE_B = compose(↩
      poseE_O1,poseO1_B);
```

Listing 2: Coordinate Calculations - Output

```
1 //Construction of poses
2 Result: poseO1_B = [1,0,0;0,1,0;0,0,1][-0.5,0.5,0.5]]
3 Result: poseO2_E = [1,0,0;0,1,0;0,0,1][0,0,0.1]]
4 Result: poseO2_O1 = [0,0,-1;0,1,0;1,0,0][0.5,0,0]]
5 //Step1: Determine poseE_O2 by inverting poseO2_E
6 Result:  poseE_O2 = [1,0,0;0,1,0;0,0,1][0,0,-0.1]]
7 //Step2: Determine poseE_O1 by composing poseE_O2 and↩
       poseO2_O1
8 Result: poseE_O1 = [0,0,-1;0,1,0;1,0,0][0.6,0,0]]
9 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
10 Result: poseE_B = [0,0,-1;0,1,0;1,0,0][0.1,0.5,0.5]]
```

5.2 Semantics Checking

As explained in Section 3.2, the second use case works at the semantic level while performing abstraction of particular coordinate representations and the coordinate frames.

Listing 3 and Listing 4 show the C++ code and the program output of the running example, respectively, for the second use case. To emphasize the changes of the semantics checking (second use case) with respect to the coordinate calculations (first use case) in Listing 1 and Listing 2, they are underlined.

As the program output shows, the semantics of the result of a semantic operator are automatically determined from the semantics of the operator arguments (lines 7, 10, and 13). Furthermore, the semantic correctness of all operations is checked, as shown in the program output (lines 6, 9, and 12).

Listing 3: Semantics Checking

```
1 //Construction of poses
2 PoseSemantics poseO1_B("o1","o1","O1","b","b","B");
3 PoseSemantics poseO2_E("o2","o2","O2","e","e","E");
4 PoseSemantics poseO2_O1(↩
      "o2","o2","O2","o1","o1","O1");
5 //Step1: Determine poseE_02 by inverting poseO2_E
6 PoseSemantics poseE_O2 = poseO2_E.inverse();
7 //Step2: Determine poseE_01 by composing poseE_O2 and↩
       poseO2_O1
8 PoseSemantics poseE_O1 = compose(poseO2_O1,poseE_O2);
9 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
10 PoseSemantics poseE_B = compose(poseE_O1,poseO1_B);
```

Listing 4: Semantics Checking - Output

```
1 //Construction of poses
2 Result: poseO1_B = Pose((o1,[o1])|O1,(b,[b])|B)
3 Result: poseO2_E = Pose((o2,[o2])|O2,(e,[e])|E)
4 Result: poseO2_O1 = Pose((o2,[o2])|O2,(o1,[o1])|O1)
5 //Step1: Determine poseE_02 by inverting poseO2_E
6 Semantic output: Pose((o2,[o2])|O2,(e,[e])|E).inverse↩
      () is OK.
7 Result: poseE_O2 = Pose((e,[e])|E,(o2,[o2])|O2)
8 //Step2: Determine poseE_01 by composing poseE_O2 and↩
       poseO2_O1
9 Semantic ouput: Composition of Pose((o2,[o2])|O2,(o1↩
      ,[o1])|O1) and Pose((e,[e])|E,(o2,[o2])|O2) is ↩
      OK.
10 Result: poseE_O1 = Pose((e,[e])|E,(o1,[o1])|O1)
11 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
12 Semantic output: Composition of Pose((e,[e])|E,(o1,[↩
      o1])|O1) and Pose((o1,[o1])|O1,(b,[b])|B) is OK.
13 Result: poseE_B = Pose((e,[e])|E,(b,[b])|B)
```

5.3 Coordinate Semantics Checking

As explained in Section 3.3, the third use case works at the coordinate semantic level while performing abstraction of particular coordinate representations. In addition to the second use case, the semantics involved by the coordinate frame are tested.

Listing 5 and Listing 6 show the C++ code and the program output of the running example, respectively, for the third use case. To emphasize the changes of the coordinate semantics checking (third use case) with respect to the semantic checking (second use case) in Listing 3 and Listing 4, they are underlined.

As the program output shows, the coordinate frame semantics of the result of a semantic operator are automatically determined from the semantics of the operator arguments, in addition to the semantic output we already ob-

tained for the second use case (lines 8, 12, and 16). Furthermore, the semantic correctness concerning the coordinate frames of all operations (lines 7, 11, and 15) is checked in addition to the checks of the second use case (lines 6, 10, and 14), as shown in the program output (Listing 6).

Listing 5: Coordinates Semantics Checking

```
1 //Construction of poses
2 PoseCoordinatesSemantics poseO1_B("o1","o1","O1","b",↩
      "b","B","b");
3 PoseCoordinatesSemantics poseO2_E("o2","o2","O2","e",↩
      "e","E","e");
4 PoseCoordinatesSemantics poseO2_O1("o2","o2","O2","o1↩
      ","o1","O1","o1");
5 //Step1: Determine poseE_02 by inverting poseO2_E
6 PoseCoordinatesSemantics poseE_O2 =poseO2_E.inverse()↩
      ;
7 //Step2: Determine poseE_01 by composing poseE_O2 and↩
       poseO2_O1
8 PoseCoordinatesSemantics poseE_O1 = compose(poseO2_O1↩
      ,poseE_O2);
9 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
10 PoseCoordinatesSemantics poseE_B = compose(poseE_O1,↩
      poseO1_B);
```

Listing 6: Coordinates Semantics Checking - Output

```
1 //Construction of poses
2 Result: poseO1_B = Pose((o1,[o1])|O1,(b,[b])|B,[b])
3 Result: poseO2_E = Pose((o2,[o2])|O2,(e,[e])|E,[e])
4 Result: poseO2_O1 = Pose((o2,[o2])|O2,(o1,[o1])|O1,↩
      [o1])
5 //Step1: Determine poseE_02 by inverting poseO2_E
6 Semantic output: Pose((o2,[o2])|O2,(e,[e])|E).inverse↩
      () is OK.
7 CoordSemantic output: Pose((o2,[o2])|O2,(e,[e])|E,[e↩
      ]).inverse() is OK.
8 Result: poseE_O2 = Pose((e,[e])|E,(o2,[o2])|O2,[o2])
9 //Step2: Determine poseE_01 by composing poseE_O2 and↩
       poseO2_O1
10 Semantic output: Composition of Pose((o2,[o2])|O2,(o1↩
      ,[o1])|O1) and Pose((e,[e])|E,(o2,[o2])|O2) is ↩
      OK.
11 CoordSemantic output: Composition of Pose((o2,[o2])|↩
      O2,(o1,[o1])|O1,[o1]) and Pose((e,[e])|E,(o2,[o2↩
      ])|O2,[o2]) is OK.
12 Result: poseE_O1 = Pose((e,[e])|E,(o1,[o1])|O1,[o1])
13 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
14 Semantic output: Composition of Pose((e,[e])|E,(o1,[↩
      o1])|O1) and Pose((o1,[o1])|O1,(b,[b])|B) is OK.
15 CoordSemantic output: Composition of Pose((e,[e])|E,(↩
      o1,[o1])|O1,[o1]) and Pose((o1,[o1])|O1,(b,[b])|↩
      B,[b]) is OK.
16 Result: poseE_B = Pose((e,[e])|E,(b,[b])|B,[b])
```

5.4 Coordinate Calculations and Coordinate Semantics Checking

As explained in Section 3.4, the fourth use case combines coordinate semantics checking with numerical calculations using particular coordinate representations. So, in fact the fourth use case combines the first and third use cases.

Listing 7 and Listing 8 show the C++ code and the program output of the running example, respectively, for the fourth use case. To emphasize the changes of the coordinate calculations and coordinate semantics checking (fourth use case) with respect to the coordinate semantic checking (third use case) in Listing 5 and Listing 6, they are underlined.

As the program output shows, the actual numerical calculations with the particular coordinate representations are performed on top of the semantic checks (lines 8, 12, and 16). Furthermore, the semantic constraints imposed by particular coordinate representations are checked: in this particular case the homogeneous transformation matrix coordinate representation imposes the constraints that the point and coordinate frame belong to the same frame, the reference point and reference orientation frame belong to the same reference frame, and the coordinate frame is equal to the reference orientation frame (lines 7, 11, and **??**).

Listing 7: Coordinate Calculations and Coordinate Semantics Checking

```
1 //Construction of KDL coordinates
2 Rotation coordRotO1_B = Rotation::EulerZYX(0,0,0);
3 Vector coordPosO1_B(-0.5,0.5,0.5);
4 Rotation coordRotO2_E = Rotation::EulerZYX(0,0,0);
5 Vector coordPosO2_E(0,0,0.1);
6 Rotation coordRotO2_O1 = Rotation::EulerZYX(0,-M_PI↩
      /2,0);
7 Vector coordPosO2_O1(0.5,0,0.0);
8 //Construction of pose coordinates
9 PoseCoordinates<KDL::Frame> poseCoordO1_B(KDL::Frame(↩
      coordinatesRotO1_B,coordinatesPosO1_B));
10 PoseCoordinates<KDL::Frame> poseCoordO2_E(KDL::Frame(↩
      coordinatesRotO2_E,coordinatesPosO2_E));
11 PoseCoordinates<KDL::Frame> poseCoordO2_O1(KDL::Frame↩
      (coordinatesRotO2_O1,coordinatesPosO2_O1));
12
13 //Construction of poses
14 Pose<KDL::Frame> poseO1_B("o1","o1","O1","b","b","B",↩
      "b",poseCoordO1_B);
15 Pose<KDL::Frame> poseO2_E("o2","o2","O2","e","e","E",↩
      "e",poseCoordO2_E);
16 Pose<KDL::Frame> poseO2_O1("o2","o2","O2","o1","o1","↩
      O1","o1",poseCoordO2_O1);
17 //Step1: Determine poseE_02 by inverting poseO2_E
18 Pose<KDL::Frame> poseE_O2 = poseO2_E.inverse();
19 //Step2: Determine poseE_01 by composing poseE_O2 and↩
       poseO2_O1
20 Pose<KDL::Frame> poseE_O1 = compose(poseO2_O1,↩
      poseE_O2);
21 //Step3: Determine poseE_B by composing poseE_O1 and ↩
      poseO1_B
22 Pose<KDL::Frame> poseE_B =compose(poseE_O1,poseO1_B);
```

Listing 8: Coordinate Calculations and Coordinate Semantics Checking - Output

```
1 //Construction of poses
2 Result: poseO1_B = Pose((o1,[o1])|O1,(b,[b])|B,[b]) ↩
      with coordinates:[[1,0,0;0,1,0;0,0,1][-0.5,0.5,0.5]]
3 Result: poseO2_E = Pose((o2,[o2])|O2,(e,[e])|E,[e]) ↩
      with coordinates:[[1,0,0;0,1,0;0,0,1][0,0,0.1]]
4 Result: poseO2_O1 = Pose((o2,[o2])|O2,(o1,[o1])|O1,[↩
      o1]) ↩
      with coordinates:[[ 0,-0,-1;0,1,-0;1,0,0][0.5,0,0]]
```

```
//Step1: Determine poseE_O2 by inverting poseO2_E
Semantic output: Pose((o2,[o2])|O2,(e,[e])|E).inverse() is OK.
CoordSemantic output: Pose((o2,[o2])|O2,(e,[e])|E,[e]).inverse() is OK.
Result:  poseE_O2 = Pose((e,[e])|E,(o2,[o2])|O2,[o2]) with coordinates [[1,0,-0;0,1,0;0,0,1][-0,-0,-0.1]]
//Step2: Determine poseE_O1 by composing poseE_O2 and poseO2_O1
Semantic output: Composition of Pose((o2,[o2])|O2,(o1,[o1])|O1) and Pose((e,[e])|E,(o2,[o2])|O2) is OK.
CoordSemantic output: Composition of Pose((o2,[o2])|O2,(o1,[o1])|O1,[o1]) and Pose((e,[e])|E,(o2,[o2])|O2,[o2]) is OK.
Result: poseE_O1 = Pose((e,[e])|E,(o1,[o1])|O1,[o1]) with coordinates:[[0,0,-1;0,1,0;1,0,0][0.6,0,0]]
//Step3: Determine poseE_B by composing poseE_O1 and poseO1_B
Semantic output: Composition of Pose((e,[e])|E,(o1,[o1])|O1) and Pose((o1,[o1])|O1,(b,[b])|B) is OK.
CoordSemantic output: Composition of Pose((e,[e])|E,(o1,[o1])|O1,[o1]) and Pose((o1,[o1])|O1,(b,[b])|B,[b]) is OK.
Result: poseE_B = Pose((e,[e])|E,(b,[b])|B,[b]) with coordinates:[[0,0,-1;0,1,0;1,0,0][0.1,0.5,0.5]]
```

5.5 Geometric semantics software inside robotic applications

After studying the different use cases, the question remains when, how, and where the geometric semantics software should be used in robotic applications. While using the geometric semantics software requires an extra effort during code writing (defining semantics, using the semantic operations, . . .) we believe that this effort easily pays off, since it allows to detect errors in geometric calculations at an early stage. In particular we want to discuss the advantages of the geometric semantics software for two cases: (i) for writing new components or libraries (check internal semantics) and (ii) for interfacing existing components and/or libraries (check interface semantics).

The **first case (check internal semantics)** is relevant during component and library development involving geometric operations. While the geometric semantics software allows to detect errors during the development, the semantic correctness of the internal geometric operations is often guaranteed as soon as the development ends. Therefore, the semantic checks can be turned off (currently using compile flags). Even when the checks are turned off, there is still a small overhead associated with the geometric semantics (extra memory for storing the semantic information). Therefore, future work will consist of developing tools that generate code in which the semantics is stripped such that only the actual coordinate calculations are retained.

The **second case (check interface semantics)** is relevant when using existing components and libraries. These components and libraries often have as inputs and outputs predefined sets of geometric relations (for instance: the expected input is the pose of the end effector with respect to the base PoseCoord $(\{ee\}\,|\mathcal{EE},\{b\}\,|\mathcal{B},[b])$ and the output is the twist of the end effector with respect to the base TwistCoord $(ee|\mathcal{EE},\mathcal{B},[b])$). The correctness of the geometric calculations in the components and library can often only be guaranteed provided that the input is correct. To this end, the geometric semantics software helps to define the semantic interface of the components and libraries and allows us to check if the provided inputs and outputs are as expected during integration.

6 Common errors in geometric rigid-body relations calculations in robotics

In Part 1 [3], we identified five common errors that could be prevented by making the semantics underlying the geometric rigid-body relations explicit. On top of the teaser presented in the beginning of the tutorial handling the first common error (“Common error box 1”), the “Common error boxes 2–5” show how the other four common errors can be prevented using the software presented in this tutorial.

7 Extending geometry library with semantic checking

If a programmer wants to perform actual geometric relation calculations using his own geometry library not yet offering semantic support, the programmer can easily build semantic support on top of this library. Thanks to the software design (Section 4) based on composition, extending a geometry library with semantic support boils down to implementing a limited number of functions, which make the connection between semantic operations (for instance composition) and actual coordinate representation calculations (for instance multiplication of homogeneous transformation matrices). Thanks to the composition-based design there is no need to change the original class libraries. A tutorial explaining how to extend your geometry library with semantic checking is available [2].

8 Conclusion

This tutorial presented a software design founded on the semantics underlying the rigid-body geometric relations of position, orientation, pose, linear velocity, angular velocity, and twist (presented in Part 1 [3]), and shows how this software prevents common errors, and hence reduces application (and, especially, system integration) development time considerably.

This tutorial is supplementary to the software web page [2], containing a more detailed explanation on the C++ libraries implementing the geometric semantics for rigid

Common error 2: Composition of twists with different points

Composing twists requires a common point (i.e. the twists have to express the linear velocity of the same point on the body). By including the point of the twist in the semantic representation, this constraint can be checked explicitly. The C++ code in Listing 1 and the program output in Listing 2 show how the software proposed in this tutorial prevents composition of twists with different points.

Listing 1: Composition of twists with different point - C++

```
TwistSemantics twist1("e","C","D");
TwistSemantics twist2("f","D","E");
TwistSemantics twist_composition = compose(twist1,↩
    twist2);
```

Listing 2: Composition of twists with different point - Output

```
Semantic output: Composition of Twist(e|C,D) and Twist(f|D,E) is NOK since:
          * The point of Twist(e|C,D) and Twist(f|D,E) have to be equal.
```

Common error 3: Composition of geometric relations expressed in different coordinate frames

Composing geometric relations using coordinate representations like position vectors, linear and angular velocity vectors, and six-dimensional vector twists, requires that the coordinates are expressed in the *same coordinate frame*. By including the coordinate frame in the coordinate semantic representation of the geometric relations, this constraint can be checked explicitly. The C++ code in Listing 1 and the program output in Listing 2 show how the software proposed in this tutorial prevents composition of angular velocities in different coordinate frames.

Listing 1: Common error 3 - Composition of geometric relations expressed in different coordinate frames

```
AngularVelocityCoordinatesSemantics angVel1("C","D","↩
    r1");
AngularVelocityCoordinatesSemantics angVel2("D","E","↩
    r2");
AngularVelocityCoordinatesSemantics ↩
    angVel_composition = compose(angVel1,angVel2);
```

Listing 2: Common errors 3 - Composition of geometric relations expressed in different coordinate frames - Output

```
Semantic output: Composition of AngularVelocity(C,D) and AngularVelocity(D,E) is OK.
CoordSemantic output: Composition of AngularVelocity(C,D,[r1]) and AngularVelocity(D,E,[r2]) is NOK since:
  * Coordinate frame of AngularVelocity(C,D,[r1]) != coordinate frame of AngularVelocity(D,E,[r2])
```

Common error 4: Composition of poses and orientation coordinate representations in wrong order

The rotation matrix and homogeneous transformation matrix coordinate representations can be composed using simple multiplication. Since matrix multiplication is however not commutative, a common error is to use a wrong multiplication order in the composition. The correct multiplication order can however be directly derived when including the bodies, frames, and points in the coordinate semantic representation of the geometric relations. The C++ code in Listing 1 and the program output in Listing 2 show how the software proposed in this tutorial automatically determines the correct order of multiplication from the semantics when composing rotation matrices.

Listing 1: Composition of poses and orientation coordinate representations in wrong order - C++

```
//Construction
KDL::Rotation coordinatesRot1 = KDL::Rotation::EulerZYX(M_PI,M_PI/2,M_PI/3);
OrientationCoordinates<KDL::Rotation> orientCoord1(coordinatesRot1);
Orientation<KDL::Rotation> orient1("a","C","b","D","b",orientCoord1);
KDL::Rotation coordinatesRot2 = KDL::Rotation::EulerZYX(M_PI/4,M_PI/4,M_PI/2);
OrientationCoordinates<KDL::Rotation> orientCoord2(coordinatesRot2);
Orientation<KDL::Rotation> orient2("b","D","c","E","c",orientCoord2);
// wrong order of multiplication
KDL::Rotation coordinatesRot_composition_wrong = coordinatesRot1 * coordinatesRot2;
// correct order of multiplication
KDL::Rotation coordinatesRot_composition_correct = coordinatesRot2 * coordinatesRot1;
// automatic order by semantics
Orientation<KDL::Rotation> orient_composition = compose(orient1,orient2);
```

Listing 2: Composition of poses and orientation coordinate representations in wrong order - Output

```
// wrong order of multiplication
Result: coordinatesRot_composition_wrong = [-0.0794593,-0.786566,0.612372; -0.862372,0.362372,0.353553; -0.5,-0.5,-0.707107]
// correct order of multiplication
Result: coordinatesRot_composition_correct = [-0.707107,-0.683013,0.183013; 0.707107,-0.683013,0.183013; 0,0.258819,0.965926]
// automatic order by semantics
Semantic output: Composition of Orientation([a]|C,[b]|D) and Orientation([b]|D,[c]|E) is OK.
CoordSemantic output: Composition of Orientation([a]|C,[b]|D,[b]) and Orientation([b]|D,[c]|E,[c]) is OK.
Result: orient_composition = Orientation([a]|C,[c]|E,[c])= [-0.707107,-0.683013,0.183013; 0.707107,-0.683013,0.183013; 0,0.258819,0.965926]
```

Common error 5: Integration of twists when point and coordinate frame do not belong to same frame

A twist can only be integrated when it expresses the linear velocity of the origin of the coordinate frame the twist is expressed in. When including the point and the coordinate frame in the coordinate semantic representation of the twist, this constraint can be checked explicitly.

The C++ code in Listing 1 and the program output in Listing 2 show how the software proposed in this tutorial prevents the integration of twists when the point and coordinate frame do not belong to the same frame.

Listing 1: Integration of twists when point and coordinate frame do not belong to same frame -C++

```
TwistCoordinatesSemantics twist("e","C","D","r");
PoseCoordinatesSemantics pose_integrated = twist.integrate(1.0);
```

Listing 2: Integration of twists when point and coordinate frame do not belong to same frame - Output

```
CoordSemantic output: Twist(e|C,D,[r]).integrate() is NOK since:
  * Coordinate frame of Twist(e|C,D,[r]) != point of Twist(e|C,D,[r])
```

body calculations. This software is publicly and freely available under the dual LGPLv2.1/BSD open-source license. The design explained in this tutorial is, however, more generally applicable and can be used as a basis for libraries in other languages. Furthermore, we believe it is particularly suited as a programming language-independent Domain Specific Language (DSL) for geometric relations. Therefore, future work aims at developing a DSL founded on the semantics proposed in Part 1 and the software design presented in this tutorial, and tools to do the actual checking, generate verified code (model-to-text conversion), and transform between different coordinate representations (model-to-model conversion).

Acknowledgment

The authors gratefully acknowledge the financial support by KU Leuven's Concerted Research Action GOA/2010/011 *Global real-time optimal control of autonomous robots and mechatronic systems*, the FWO project G040410N *Autonomous manipulation using a flying robot*, the KU Leuven-BOF PFV/10/002 Center-of-Excellence Optimization in Engineering (OPTEC), and the European FP7 projects 2008-ICT-230902-ROSETTA 2008-ICT-231940-BRICS, and FP7-ICT-288533 RoboHow.Cog. Tinne De Laet is a Post-Doctoral Fellow of the Fund for Scientific Research–Flanders (F.W.O.) in Belgium.

References

[1] Robotics library. `http://sourceforge.net/apps/mediawiki/roblib`.

[2] Tinne De Laet and Steven Bellens. Geometric semantics software. `http://www.orocos.org/wiki/geometric-relations-semantics-wiki`, 2012. Last visited May 2012.

[3] Tinne De Laet, Steven Bellens, Ruben Smits, Erwin Aertbeliën, Herman Bruyninckx, and Joris De Schutter. Geometric relations between rigid bodies: Semantics for standardization. *IEEE Robotics and Automation Magazine*, 2012.

[4] Tully Foote. Robot Operating System (ROS) geometry stack. `http://ros.org/wiki/geometry`, 2008. Last visited 2012.

[5] Ruben Smits. KDL: Kinematics and Dynamics Library. `http://www.orocos.org/kdl`, 2001. Last visited 2011.

[6] Kenneth Waldron and James Schmiedler. Kinematics. In *Handbook of Robotics*, chapter A.1, pages 9–33. Springer-Verlag, Berlin, Heidelberg, 2008.