Journal of Software Engineering for Robotics 7(1), July 2016, 36-54
ISSN: 2035-3928

RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages

Marco Frigerio[1,*] Jonas Buchli[2] Darwin G. Caldwell[1] Claudio Semini[1]

[1] Department of Advanced Robotics, Istituto Italiano di Tecnologia, via Morego, 30, 16163 Genova, Italy
[2] Agile & Dexterous Robotics Lab, ETH Zürich, John-von-Neumann Weg 9, 8093 Zürich, Switzerland

Abstract—This paper presents the *Robotics Code Generator* (RobCoGen), an open source program that can generate an efficient implementation of rigid body dynamics and kinematics algorithms for articulated robots, such as humanoids and manipulators. New Domain Specific Languages (DSLs) for the specification of robot models and coordinate transforms lie at the core of our software. The technology of DSLs and code generation allows users to deal only with high level information (like a robot model), and relieves them from low level coding of critical routines. The generated code is efficient, tailored to each robot and suitable for applications ranging from simulations to hard real time robot control. This paper describes the rationale and the development of RobCoGen, as well as experiments that include comparisons with other programs: results show a comparable efficiency with code optimized using symbolic simplification (average execution time ratio of 1.22 with SD-FAST) and better performance than a general purpose library (ratio of 3.24 with RBDL). Our design criteria were not limited to efficiency, though, but we considered ease of use and flexibility as equally relevant.

Index Terms—Software, articulated robots, dynamics, code generation, domain specific language

1 INTRODUCTION

The software for articulated robots, like simulators [1], [2] and model–based controllers [3], [4], make extensive use of algorithms and expressions related to kinematics and dynamics of multi-rigid-body systems. Despite an established theoretical understanding of the algorithms, sound implementations still demand significant resources; this fact has a negative impact on the progress of the state of the art in robotics.

The lack of reusable components based on general software models, in addition to the inherent complexity of kinematics / dynamics, are some of the reasons of the problem. Coding becomes even more challenging when the need for hard real time at high frequency (e.g. control loops in the order of $10^2 Hz$) demand for efficient code fulfilling further constraints.

This paper introduces the Robotics Code Generator (RobCoGen), a computer program we developed to generate optimized code for the dynamics and the kinematics of articulated robots [5]. The idea behind RobCoGen is to relieve the user from the manual development of code which is crucial yet complex, necessary for most applications yet rarely the actual focus of the research: namely, state-of-the-art rigid body dynamics algorithms, coordinate transformation matrices and Jacobian matrices (i.e. kinematics algorithms). RobCoGen allows its end users to deal only with high level information, the minimum amount required to define an instance of a problem, and then leave a computer program to deal with implementation details, by means of code generation. For example, if one needs to implement the coordinate transforms from A to B, one shall only be required to tell the relative position of A with respect to B. RobCoGen is based on new computer languages designed for the specification of the high level input for the code generator.

RobCoGen currently generates C++ and MATLAB code. In the case of coordinate transforms also MAXIMA code is generated; MAXIMA is the symbolic engine used by RobCoGen (see Section 5.1). The support for more programming languages can be implemented within the same framework. The generated code can be used within applications that include simulations but also hard real–time robot controllers, as the C++ code is fast and has deterministic execution time. The wide range of possible applications is one of the valuable features of the tool.

Regular paper – *Manuscript received August 29, 2015; revised January 28, 2016.*

- *This work was supported by Fondazione IIT. J.B. is supported by a Professorship Award from the Swiss National Science Foundation.*

The current code generation targets include state-of-the-art efficient algorithms for multibody dynamics: the Composite Rigid Body Algorithm for the computation of the joint space inertia matrix, the Newton-Euler algorithm for inverse dynamics and the Articulated Body Algorithm for forward dynamics [6]. An optimized, tailored instance of the algorithms is generated for each robot model given as input. In addition, RobCoGen generates an implementation of arbitrary coordinate transformation matrices and geometric Jacobians.

1.1 Motivation and contribution

The scarcity of truly reusable robotics software often results in wasting significant development time on the so called *non–problems*; The programmers have to face problems that have been thoroughly addressed from the theoretical viewpoint although practical and correct implementations can still be hard to achieve. This has the unfortunate consequence of turning these non–problems back to actual ones. Coordinate transformation matrices may seem a naive example, yet it is a clear case of a recurring non–problem of almost any robotics software application. On one hand, the mathematics of coordinate transforms is text–book knowledge, on the other hand the mathematical concepts can be confusing and error–prone when it comes to writing and validating a piece of software that deals with different coordinate systems.

Furthermore, addressing the requirements of hard real time is fundamental for the implementation of control algorithms running on real robot hardware; any failure can lead to severe consequences. Efficient code is desirable also for simulations and other applications like sampling based planning, where the faster the implementation of the dynamics, the better. Addressing the apparent trade–off between efficiency and flexibility/generality would substantially improve the effectiveness of software for robotics applications.

Our contribution to this end is a toolchain, RobCoGen, for the automatic code generation of algorithms for rigid body dynamics and kinematics, which can be used in simulation *and* control applications. RobCoGen represents a progress beyond state-of-the-art tools since it gathers different effective features – usually not available together.

- Ease of use: the user is required to deal only with high level information and is relieved from time consuming and error–prone code development. Dedicated computer languages give users a compact and readable format to specify such information.
- Reliability and robustness: an automated code generation process is repeatable and does not introduce occasional mistakes, as opposed to manual coding.
- Efficiency: the generated code can be optimized to address the speed and the constraints of fast real time robot controllers. Efficiency is achieved by generating robot–specific instances of the general algorithms.
- Flexibility: sound domain models make the software more general. Any articulated robot (without loops) is supported, arbitrary coordinate transforms can be chosen, etc. Code generation in different programming languages allows to target different platforms.
- Re-use and composability: using the generated code in a custom application is simple, thanks to the focus on specific, well defined algorithms (e.g. inverse dynamics) and modularity in the generated implementation.

In general, RobCoGen offers a good compromise between efficiency and flexibility/generality. A general purpose dynamics library (such as ODE [7]) is more flexible than RobCoGen in that it works with robot models that can be determined at run–time. However, dynamically building the robot model during system operation is rarely an actual requirement. Especially in the case of deployment on a real robot, when its dynamics is usually the only problem that matters: a dedicated implementation like robot-specific generated code is more effective to keep the system lightweight and efficient.

Finally, RobCoGen is open source and it is built with open source technologies only. Thus it is convenient for the user to install and use the toolchain, as well as possibly contribute to it. These points increase the chance of the adoption of our toolchain in the community.

1.2 The role of DSLs

A Domain Specific Language (DSL) is a computer language whose notation explicitly refers to concepts of a specific domain, so that its expressiveness is tailored for a particular class of problems. *External* DSLs have their own, custom grammar and therefore are independent languages, whereas *internal* DSLs are realized by a specific use of a general purpose programming language, from which they inherit the basic syntax [8], [9].

RobCoGen is based on three external Domain Specific Languages that we designed (Kinematics-DSL, Motion-DSL, Transforms-DSL), which in turn are based on general domain models we devised. These DSLs can be thought of as *specification* languages, for the compact representation of coordinate transforms and the structure of articulated robots. We chose external DSLs because they define a textual format that is easily human and machine readable. Transforms and robot descriptions are simple enough for a text document to be convenient for the user. From the perspective of the DSL developer, instead, we could exploit the extensive support of modern language workbenches such as Xtext for Eclipse [10]. Because of our requirement to have code that can run efficiently also on real robots, typically within existing frameworks/platforms, code generation in multiple languages was a better choice than, for example, an internal DSL executed by an interpreter or a virtual machine.

In summary, external DSLs were the right implementation technology for our needs. In fact we believe this choice to be a distinctive feature of RobCoGen that compares favorably with other similar software packages, as those mentioned in Section 2.

1.3 Overview

RobCoGen is essentially a Java program that uses the code generation functions of the DSLs described in this paper. The core of RobCoGen is the Kinematics-DSL, which defines a new file format for the description of robot models and whose generators deal with the rigid body dynamics algorithms. A robot model in the form of a document of the Kinematics-DSL is the main input for RobCoGen (see an example in Figure 5).

The DSLs and generators that address the problem of coordinate transforms (the Motion-DSL and Transforms-DSL) constitute in fact an independent software. That is, one can generate the implementation of arbitrary coordinate transforms without any robot model, since coordinate transforms is an independent domain (the examples in Figure 4 refer indeed to a generic use case that is not necessarily related to robotics). RobCoGen itself uses such a software to generate the coordinate transforms for the particular case of articulated robots. The development of a standalone code generator for coordinate transforms is a further contribution of our work.

The structure of the central part of this paper reflect the logical development process of RobCoGen: analysis of the domain, design of the domain specific languages, implementation of the code generators. These three subjects are addressed respectively in Section 3, 4 and 5. The tool itself as a whole is the topic of Section 6, while Section 7 illustrates various experimental results. Section 8 discusses open issues and future developments, while the final remarks of Section 9 close the paper.

Before all that, the next Section gives a brief overview of related work.

2 Related work

For the past few years, the literature on robotics software has often discussed the lack of standards, models, rigorous structure and principles in the development process, even after decades of research in the robotics field [11]. The available experience does not get transferred effectively in the software development process. To address these issues, the fields of Model Driven Engineering and Domain Specific Languages have received growing attention by the robotics community [12], [13]. For an extensive survey about the use of DSLs in robotics, for example, refer to [9].

The generation of code exploiting symbolic math engines is not a new idea, and some examples can be found already in old literature [14]. A well known, commercial computer program implementing this idea is SD-FAST [15]. SD-FAST produces C or Fortran implementation of the equations of motions and various analysis routines for a wide variety of mechanical systems. It supports also closed loop mechanisms and various joint types. In comparison with SD-FAST, RobCoGen provides several advantages: it supports two modern programming languages (C++ and MATLAB) and can be extended with others. The C++ classes generated by RobCoGen are more modular than the C code of SD-FAST, they do not impose any policy on their usage and therefore are more reusable (no state machines nor global variables). Our Kinematics-DSL for the robot description looks simpler than the SD-FAST file format and at the same time it is more flexible (e.g. it allows an arbitrary orientation of the reference frames of the mechanism, at the default joint configuration). Furthermore, a DSL created with Xtext-Eclipse comes with a dedicated editor with syntax checks. In addition, RobCoGen is open source, which is advantageous for the final user, and opens the way to possible development contributions from the community.

Similar comments apply to Robotran [16]. Robotran has a graphical editor for the robot model, it generates symbolic equations of motion and performs simulations interacting with MATLAB. Robotran is also quite sophisticated and can handle a variety of mechanisms including closed loops and vehicles. It includes its own symbolic engine that allows the generation of optimized code in C language. Simulations though, have to be performed with MATLAB and Simulink, which refer to the C code and perform the numerical integration. So Robotran is tied to these programs, and does not support explicitly the generation of standalone C code – although one can try to extract some routines out of it.

In contrast to SD-FAST and Robotran, the code generators of RobCoGen use symbolic simplification marginally, only for the coordinate transforms, as detailed in Section 5. Although symbolic simplification possibly results in code with the minimum number of operations, there are a few potential disadvantages with that approach: the reduction of vector operations to scalar operations, which may make poor use of possible specialized hardware for vector arithmetic [6]; also, for the same reason, the generated code would typically be much less readable and compact. A sophisticated symbolic engine is required, which can interpret actual algorithms and is able to deal with potentially thousands of equations. From the developer perspective, the interaction with the symbolic engine could make the code generation more complicated, as opposed to a set of text templates which embed the logic of the algorithm, as in the case of RobCoGen (see Section 5 for more details).

Existing software to calculate the dynamics of multibody systems is obviously not limited to symbolic engines and code generators. RBDL [17] is a generic C++ library following the standard approach of an API requiring a robot model, in the form of a data structure, as an argument. RBDL implements the same dynamics algorithms as RobCoGen and it uses the Eigen library for linear algebra, as in the C++ code generated by RobCoGen. For these reasons RBDL is ideal to be compared to our work (Section 7.3).

Another well known C++ library in the robotics community is KDL, which is part of the Orocos project [18]. KDL aims at providing an uniform API for the forward and inverse kinematics problem. As far as dynamics is concerned, at the time of writing it seems to be limited to inverse dynamics

for linear chains. RobCoGen supports forward kinematics and forward/inverse dynamics, for kinematic trees. Inverse kinematics is not supported as an analytical solution might not exist, and code generation would not be particularly effective for the generic numerical solvers, as opposed to dynamics algorithms (see Section 5.2).

Metapod [19] is an interesting application of the technique of C++ template meta-programming. It exploits the power of the C++ compiler and templates, to instantiate an optimized, robot-specific implementation of rigid body dynamics algorithms. Although conceptually very similar to RobCoGen, this approach has a few drawbacks: it is limited to C++; template meta-programming is complex, both at development and debugging time, and it is perhaps not an ideal choice for non-trivial computations such as optimized multibody dynamics. On the other hand, the implementation of RobCoGen is a standard program that operates on a normal data structure representing the robot; thus it leverages the flexibility of a regular programming language. RobCoGen is also not limited to the dynamics algorithms, as it also generates reusable components that address only the coordinate transforms for the robot.

The Stanford Whole-Body Control (S-WBC) project [20], was initially released in 2009 as a result of the efforts of the Stanford Robotics Lab to bring the operational space formulation into software for the community. The S-WBC comprises a library for joint-space kinematics and dynamics, on top of which additional components provide the abstractions required by operational space control. Specific effort has been spent on the configurability of the framework. Our approach shares with the S-WBC some goals such as flexibility and robustness, however RobCoGen is not a full-featured framework; it focuses on self-contained algorithms to be even more general and be used within other applications/frameworks. The S-WBC itself might be an example of these. In fact, our previous work [21] contains some efficiency comparisons between RobCoGen and the kinematics/dynamics engine used by the S-WBC.

An interesting work, motivated by the need for standard models of recurring concepts in robotics, specifically the geometric relations between rigid bodies such as poses, velocities, coordinates, can be found in [22], [23]. The authors address the issue of the ambiguities of existing implementations, which arise from the lack of a standard notation and especially from the assumptions implicitly hidden in the software. These assumptions determine incompatibilities, high costs for system integration and the risk of implementing physically inconsistent operations. Our work stems from similar premises, but while (a part of) RobCoGen provides the correct implementation of any coordinate transform, the focus of the cited work is to ensure a consistent usage of the transforms at run time. The two approaches can thus be considered complementary, to an extent.

Other examples of modeling and description languages for multibody systems, which compare to our Kinematics-DSL, can also be found in existing packages and programs. In the OpenHRP simulator [24], the language for the models comes from the 3D modeling field, and mixes graphical aspects and sensors with kinematics parameters. In the Robot Operating System ROS [25], the URDF file format based on XML is the standard. The URDF also mixes kinematic/dynamics parameters with visualization information. In addition, XML is harder to read and maintain as compared to a dedicated solution such as our Kinematics-DSL. To the best of our knowledge, the OpenHRP format and the URDF do not come with efficient code generation capabilities.

The XACRO language, also part of ROS, extends the URDF with macros which improve the flexibility of the language. An interesting feature enabled by macros is modular descriptions, so that a part of a robot can be described only once but referenced multiple times in the full model; for example, the leg of a humanoid, which is repeated twice. This feature is not yet available in our Kinematics-DSL, but it is worth including in future developments.

Part of ROS as well, the *tf* library [26] provides runtime support for the management of coordinate transforms in a robotics application. On the other hand our code generator focuses on the correct implementation of specific transforms, and as such it addresses complementary requirements. For example, some of the transforms that the *tf* takes care of disseminating to the entire system, at runtime, could be implemented with our tool.

General purpose modeling languages also exist, like Modelica [27]. Modelica is a multi–domain, object–oriented modeling language for a variety of physical systems, used also in industry. Comprehensive and very general languages like Modelica, though, may prove harder to learn (version 3.3 of the language specification is 282 page long) and not as efficient as a specialized DSL, for very specific requirements such as code generation for specific algorithms used in robotics.

3 Domain analysis

This section briefly describes our analysis of the domains of interest: coordinate transforms and representation of articulated robots. We identified the relevant features of these domains, for the practical, ultimate purpose of generating code with limited user intervention. In the case of coordinate transforms, our aim was to identify the attributes required for unambiguous specifications; a prerequisite for an automatic code generation program, and a typical issue with standard approaches. In the case of the representation of robots, our goal was to formalize an established robotics convention (the kinematic tree representation), to base the design of our Kinematics-DSL on a solid foundation.

We represent the results of this analysis in the form of UML class diagrams, as capturing the static relationships among the concepts of the domain suffices for our purpose [28]. A class diagram is a compact representation, suitable as a reference for software development (in our case, for the development of the grammar of the DSLs – Section 4).

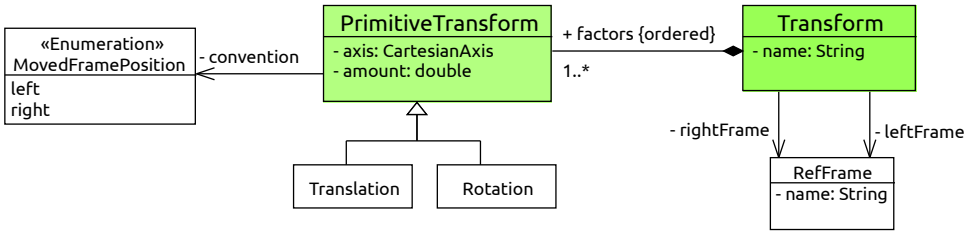


Fig. 1. The UML class diagram for coordinate transforms. A generic, named `Transform` results from the product of several `PrimitiveTransform`s, which correspond to pure rotations or translations. The `MovedFramePosition` property allows to determine uniquely any primitive transform.

3.1 Coordinate transforms

A coordinate transformation matrix X (or just "coordinate transform") maps one coordinate vector to another coordinate vector representing the same abstract vector (e.g. a relative position), but in a different coordinate system, in general translated and rotated with respect to the original one:

$$p_B = {}_B\mathbf{X}_A \cdot p_A$$

where A and B are two frames and p is the coordinate vector. In the rest of the paper we assume that we deal only with right-handed, orthogonal coordinate frames, that the sign of rotation angles is given by the right hand rule and that the matrix left–multiplies a column vector of coordinates.[1]

With reference to the layout of the general equation above, one can say that the transform maps the coordinates on its right to the coordinates on its left. Let us then introduce the terms *left frame* and *right frame* that refer to the position of the subscripts of $\mathbf{X}$ and therefore to the frames involved in the transform (respectively B and A in the example): the *right frame* (A) is the frame in which the multiplied coordinates are expressed; the *left frame* (B) is the frame in which the resulting coordinates are expressed.[2]

When developing coordinate transforms by hand or when using existing code, it is easy to make mistakes because a unique definition of a transform requires information that is usually implicit, giving rise to ambiguities. Namely, for a primitive transform that involves a single rotation/translation, one must know the *direction* of the transform: for example, assume that frame B is rotated by an angle θ about the z axis, with respect to A; while this *information* cannot be misunderstood, an expression identified as a rotation matrix for the z axis (which we call $R_z(\theta)$) often does not tell explicitly whether it transforms vectors from B to A or vice versa. The ambiguity would be easily resolved by augmenting the generic transform $X(\theta)$ with a property stating whether the moved frame is the *left* or the *right* frame. If the available representation lacks this information, the user wastes time by investigating the conventions and by making mistakes. As an example, the code in the MATLAB robotics toolbox [29] uses one convention about the direction of the transform, while the 6D transforms introduced by Featherstone in [6], [30] use the other one. That is, a matrix and its transpose are called with the same name in two different software packages.

Taking into account the considerations above, we designed the meta–model shown in Figure 1. That is the basis for our DSL which allows the user to write unambiguous definitions of coordinate transforms (Section 4). In the meta–model, a `PrimitiveTransform` is associated to a pure translation/rotation along/about a single Cartesian axis only. The `MovedFramePosition` property disambiguates the direction of the mapping, as explained above. A generic `Transform` can be represented as an ordered sequence of primitive ones, and can also have a custom name. Instances of this class would be application-specific (e.g. the transform from foot frame to torso frame, in a legged robot). The important attributes `leftFrame` and `rightFrame` identify uniquely the role of the transform (e.g. as in ${}_{foot}\mathbf{X}_{torso}$, ${}_{torso}\mathbf{X}_{foot}$, etc.). Although a transform generally embeds both translation and rotation information, the separation modeled with `PrimitiveTransform` serves to identify simple terms for the user to *specify* a transform.

The diagram is roughly a model of abstract transforms whose actual type (e.g. homogeneous transforms, spatial vector transforms) is unspecified. One may think of instances of the main classes `PrimitiveTransform` and `Transform` as generic $n \times n$ matrices, composable by means of the regular matrix product. This idea of an abstract description turns out to be useful for code generation, as code for different kind of matrices can be generated given the same input.

3.1.1 Rigid motions

So far we have not discussed the problem of actually finding the sequence of primitive transforms that result in the desired transform. Manually finding the sequence is itself a confusing task, since the composition rules depend upon the direction of

1. This is almost always the case in literature and software, thus we believe it is not a strong limitation. It would be possible to avoid the assumption and make the model even more general, however we decided to avoid it to keep the DSL and the code generation software simple.

2. The notation with the double subscript on the left and on the right is very effective for the user also to understand how to compose multiple matrices, as nearby subscripts have to match (as in ${}_B\mathbf{X}_{CC}\mathbf{X}_A$).

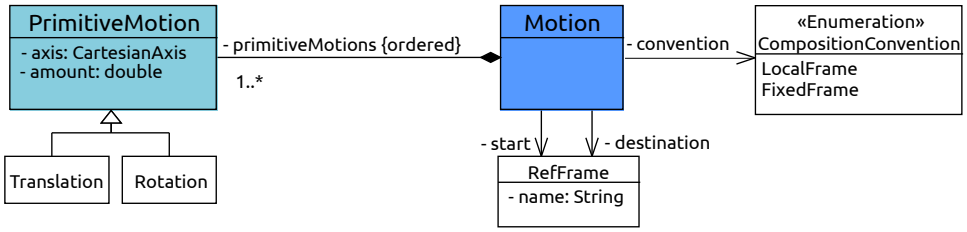


Fig. 2. A UML class diagram representing rigid motions. A `Motion` is a sequence of movements, rotations and translations, a generic body has to perform to go from one pose to another pose; poses are represented by reference frames. The interpretation of the ordered sequence of primitive motions is unique because of the `CompositionConvention` property.

the primitive transforms (discussed above) and the direction of the desired transform itself (i.e. ${}_B\mathbf{X}_A$ or ${}_A\mathbf{X}_B$). It turns out that the correct sequence can be determined from the specification of a rigid motion involving the same coordinate frames. A rigid motion is a sequence of translations and rotations of a body (or of a reference frame, which in this context is the same thing), that move it from a starting pose to a final pose.

In fact, such a geometric description is really the essential, high level information underlying the problem, as it enables the computation of any transform involving the same frames; as such, it is the information users should preferably deal with. For example, a user would describe the location of the camera frame in the robot, rather than working out the coordinate transform, which is error prone. This facts motivate the analysis of rigid motions and the development of a corresponding DSL. Figure 2 shows the UML class diagram. As expected, the model resembles quite closely the previous one about transforms, since motions and coordinate transforms are tightly correlated – although the two domains do not coincide.

The class `PrimitiveMotion` represents either a pure rotation or translation of arbitrary amount, about or along a single Cartesian axis. A composite `Motion` is instead an *ordered sequence* of primitive motions (also called "motion steps"), and represents any rigid motion. In order for the step sequence to have a unique interpretation, it is necessary to declare with respect to which reference each step is defined. An intuitive choice is to assume each motion step to be performed with respect to the current moving frame, i.e. the frame resulting from all the previous motion steps. Another possibility is to use only the frame fixed in the initial pose, so that the axes for translations and rotations never change.[3] In the diagram of Figure 2, the property `CompositionConvention` refers to this point.

The meta–model for motions we have just described, as well as the previous one for coordinate transforms, include the properties that ensure that any conforming description (of transforms/motions) is uniquely interpretable. Such a property will propagate to the DSLs (Section 4) and will ultimately lead to more robust code generators (Section 5).

3. In this case the order of the primitive translations does not matter

3.2 Articulated robots

The very well know equation of motion for a multi–rigid–body system can be written in the following form:

$$\boldsymbol{\Gamma} = \mathbf{H}(\mathbf{q})\ \ddot{\mathbf{q}} \ + \ \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}) \tag{1}$$

where $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ are respectively the vectors of the scalar position, velocity and acceleration for each degree of freedom of the robot, while $\boldsymbol{\Gamma}$ represents the forces at the joints. $\mathbf{H}$ is the matrix that expresses the inertia seen locally at each joint, while $\mathbf{h}$ is a term that accounts for the position and the velocity dependent forces (and possibly other additional forces due to contacts, springs, etc.). In this work, we consider the dynamics algorithms to compute either $\boldsymbol{\Gamma}$ (inverse dynamics), $\ddot{\mathbf{q}}$ (forward dynamics) or the matrix $\mathbf{H}$ [6].

As noted in [6], in addition to inputs such as the position and velocity of the joints, these algorithms depend explicitly on the *model* of the robot that describes its kinematics and dynamics. The structure of these models is basically standard in robotics and mechanism analysis. In short, it is a topological description which represents the multibody system as a graph, where joints are arcs and bodies are nodes (quite the contrary of what graphical intuition might suggest). Joints have a type (e.g. revolute, prismatic) and bodies have inertia properties; geometric information about the position of the joints must also be provided [31], [6]. We shall call this abstraction the "kinematic tree" meta–model; the robot models we target in this work conform to such a meta–model. These robots include humanoids, quadrupeds, manipulators, or analogous articulated mechanisms. The dependency on the robot model is a key point about the nature of dynamics algorithms, which reveals the opportunity to implement specific *instances* of the general algorithms, tailored to specific robots (cf. Section 5.2).

Figure 3 shows an UML class diagram reproducing the key elements of the kinematic tree meta–model. The diagram is simple but general enough to account for any articulated robot without kinematic loops. A conforming robot model would contain all and only the information that fully specifies the

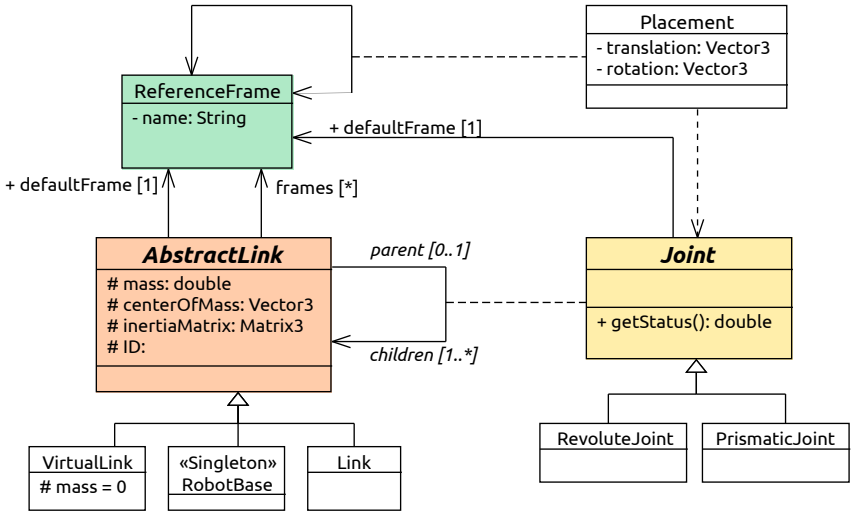


Fig. 3. The kinematic tree meta–model as an UML class diagram. It can represent multibody systems, capturing the role of a joint as the connection between a parent-child pair of links, and emphasizing their association with reference frames. The convention about the placement of these frames (that cannot be modeled in the diagram) and the parameters about the relative pose of two successive ones (the `Placement` class) provide the full geometry information about the robot.

physics of the system. A few more details about the class diagram follow.

The abstract class `AbstractLink` models any rigid body with inertia properties; we used a few subclasses to represent special cases: `Link` for the regular links of the robot. `RobotBase` for the special link identified as the root of the kinematic tree; it can be *floating* if the robot has a mobile, underactuated base, such as a humanoid. `VirtualLink` for dimensionless bodies that allow the composition of primitive joints to model more complex mechanisms; this class explicitly forces the inertia properties of its instances to be zero.

Any link can have multiple children (but only one parent), which accounts for kinematic branches such as multiple legs attached to the torso of a legged robot. The parent–child relationship is induced by joints, whose type determines the kinematic constraint between the two bodies; for this reason we chose to make `Joint` an association-class connected to the self–association for `Link`. Also the `Joint` class is subclassed to distinguish between prismatic and revolute joints. The common operation `getStatus()` shows that it must be possible in some way to inspect the joint status variable (a scalar, since we are dealing with 1-DoF only joints).

Finally, the class `ReferenceFrame` represents a coordinate system. Each joint and link have a default frame, whose physical location in the robot is defined by a convention. A convention is necessary in order to give a meaning to the geometrical parameters in a robot model. In short, link and joint frames have the z axis aligned with the joint motion axis; any joint frame is defined with respect to the frame of the link that supports the joint; any link frame coincides with the frame of the previous joint that supports the link. More information about this convention can be found in Chapter 4 of [6] and in the manual of our Kinematics-DSL (see Section 4). The `Placement` class wraps the parameters of a relative pose, and it is used for example to specify the location of a joint frame with respect to the link frame, as mentioned above. Although the same information could be represented with the classes in Figure 2, we picked a less general representation to keep the model simple; this point is further discussed in Section 6 and Figure 9. Finally, links can optionally have any number of additional frames, defined in turn with respect to the default one.

4 DSLs

This section describes the Domain Specific Languages (DSLs) that we developed based on the analysis described in the previous section. Our DSLs are specification languages, to conveniently specify coordinate transforms and articulated robot models. They are "external" DSLs, that is, they are defined by their own, custom grammar. This paper describes three DSLs: the Kinematics-DSL to specify articulated robot models; the Motion-DSL and the Transforms-DSL to specify respectively rigid motions and coordinate transforms.

An external DSL is defined by its grammar, which formally specifies the syntax of the language: allowed keywords, statements, and so on. The grammar constrains the content of any possible document, and its structure. A domain model such as those shown in Section 3, instead, defines the structure of the *information* to be conveyed by the documents. Grammar and domain model are therefore tightly related. Roughly speaking, the grammar of every DSL of RobCoGen was created by mapping each class of the UML diagram to one or more rules of the

grammar, while adding additional syntax elements to improve the readability (e.g. parenthesis). Thus, the development of the grammars was relatively straightforward, and subject to a sound understanding of the domain.

4.1 Coordinate transforms

Figure 4 shows an excerpt from the grammar and a sample document of the Motion-DSL and the Transforms-DSL. The sample documents refer to a possible use-case in which the user wants to generate the coordinate transforms for each triplet of Euler angles. To this end, all the possible sequences of three successive rotations are listed within a document of the Motion-DSL. The list alone effectively conveys all the necessary input information and therefore is the only thing the user has to write. Indeed, the document of the Transforms-DSL can itself be automatically generated given the specification of the motions (this step is covered in Section 5.1); therefore the document of the Motion-DSL is shown first in the figure. The examples do not include any translational motion, but obviously those are also possible.

The keywords `Convention` and `MovedFramePos` refer to the corresponding properties described in the meta–models in Section 3, Figure 2 and 1. They make the documents content unambiguous. Although in principle the convention is a property of each single primitive transform/motion (cf. the meta–model), we decided to have a document–wide property that applies to all the objects, to keep the syntax simpler. So, for example, in Figure 4b, all the primitive rotations have to be interpreted as successive rotations, happening with respect to the current (i.e. `local`) frame.

4.2 Articulated robots

Figure 5 refers to the Kinematics-DSL[4], which defines the format of the robot description files that RobCoGen takes as input.A manual of the language, describing the syntax and the conventions more in detail, is available online at the wiki of RobCoGen [5]. The example in the figure describes the HyL robot (a Hydraulic Leg of HyQ [32]) that we developed at our lab. It is a three DoF, planar mechanism consisting of a 2R leg attached to a vertical slider. A Kinematics-DSL document is neat and simple, yet specifies completely the dynamics of a robot. The syntax is mostly in the form of key-value pairs, with paragraphs delimited by curly braces; it would look familiar to programmers, but it is simple enough also for other users. The Kinematics-DSL does not include any property unrelated to the geometry/dynamics of the system, like graphical attributes, sensors/actuators, etc.; that is, it does not mix separate concerns.

4. Although a robot model includes both kinematics *and* dynamics properties, to keep the name short we chose "Kinematics-DSL" because of the standard term "kinematic tree".

4.3 Constants, parameters and variables

In all our DSLs, the numerical properties that appear in a document belong to one of three categories: constants, parameters, variables. Conceptually, the distinction between the three groups is based on the relative *frequency* scale at which the values are expected to vary. Given a system in regular operation, constants do not change, parameters change from time to time (possibly because of external intervention), variables change in general continuously (i.e. state variables). The absolute time scales depend on the actual system. The possibility to distinguish between constants, parameters and variables was introduced in the grammars but it is not shown in the domain models described in Section 3, as it is not specific of a particular domain.

A robot model, that is, a document of the Kinematics-DSL, contains only constants or parameters, represented respectively by constant literals such as `0.456` and identifiers such as `link_mass`. Depending on the location in the document (i.e. the *context*), a parameter is automatically classified as a "dynamics parameter" – if it represents an inertia property – or as a "kinematics parameter" – in the case of a geometrical property. The distinction allows the code generation logic to treat the two groups of parameters differently and gives more flexibility.[5] Robot models do not contain any variable definition. Conceptually, the variables in a multibody system such as an articulated robot are the joint status variables, but those do not need to be represented in a static description of the system.

In the Motion-DSL and Transforms-DSL, instead, identifiers are treated by default as variables (like `a1 a2 a3` in the examples in Figure 4), unless they are declared explicitly as parameters, with a specific syntax. These DSLs do not impose any policy to determine which property belongs to which group, but, simply, the grammar gives a mechanism to make the distinction, if necessary. Furthermore, note that these DSLs have no concept specific to robotics like the joint status, as their purpose is to do code generation of coordinate transforms, which is an independent issue. However they are flexible enough to be *used* by RobCoGen to generate the coordinate transforms of a robot. See Section 6.

4.4 Implementation

We implemented all our DSLs with the Xtext application for Eclipse [10]. We chose Xtext as it seemed convenient, powerful, and because of the integration with Eclipse, which was already part of our development tools. The automatic generation of the DSL infrastructure given the grammar, a feature of Xtext, was also a strong motivating point. We did not review other available tools possibly comparable with Xtext as

5. The literature commonly refers to the mass and the moments of inertia as the "inertia parameters". To avoid any ambiguity, in this work we refer to them as the "inertia properties" since we use the term "parameter" for a more specific concept.

```
Motion:
    start=[Frame] '->' end=[Frame]
    ':' primitiveMotions += PrimitiveMotion*
    ('['userName=ID']')?   ;

PrimitiveMotion:  Rotation | Translation;
Rotation:    Rotx | Roty | Rotz ;
Translation: Trx  | Try  | Trz ;

Rotx:  'rotx' '('arg=ArgSpec')';
// [...]
```

(a) Excerpt from the grammar of the Motion-DSL.

```
// [...]
Convention = local // motion is wrt current frame
S -> Exyz : rotx(a11) roty(a12) rotz(a13)
S -> Exzy : rotx(a21) rotz(a22) roty(a23)
S -> Eyxz : roty(a31) rotx(a32) rotz(a33)
// [...]
```

(b) A sample document of the Motion-DSL specifying sequences of three successive rotations.

```
Transform:
'{'leftFrame=[Frame]'}''_X_''{'rightFrame=[Frame]'}'
    '=' matrices += AbsMatrix*
      ('['userName=ID']')?  ;
AbsMatrix:   Rotation | Translation ;
Rotation:    Rx | Ry | Rz ;
Translation: Tx | Ty | Tz ;

Rx: 'Rx' '('arg=ArgSpec')';
// [...]
```

(c) Excerpt from the grammar of the Transforms-DSL.

```
// [...]
MovedFramePos = right

{S}_X_{Exyz} = Rx(a11) Ry(a12) Rz(a13)
{S}_X_{Exzy} = Rx(a21) Rz(a22) Ry(a23)
{S}_X_{Eyxz} = Ry(a31) Rx(a32) Rz(a33)
// ...
{Exyz}_X_{S} = Rz(-a13) Ry(-a12) Rx(-a11)
{Exzy}_X_{S} = Ry(-a23) Rz(-a22) Rx(-a21)
{Eyxz}_X_{S} = Rz(-a33) Rx(-a32) Ry(-a31)
// ...
{Exzy}_X_{Exyz} = Ry(-a23) Rz(-a22) Rx(-a21)
                  Rx(a11) Ry(a12) Rz(a13)
```

(d) A sample document of the Transforms-DSL defining a set of coordinate transforms, *generated* from the example in (b).

Fig. 4. The DSLs for the specification of coordinate transforms. Excerpts from the Xtext grammars (left) and from example documents (right). Note that the grammar is a single, fixed specification, whereas the document on the right is only one of infinitely possible examples, conforming to the grammar. Only part of the text is shown due to space limitations. `S` and `E···` are just user defined names for hypothetical reference frames. Also the variable names `a··` are arbitrary; of course numerical constants would also be possible.

that was beyond the scope of our research, and Xtext seemed to satisfy our requirements.

For each DSL, Xtext also generates a dedicated editor, as an Eclipse plugin. The editor provides syntax coloring and auto completion. An interesting feature, although we did not focus on the DSLs editors, is the extensibility of the static analysis and validation of the code. For example, by programming an appropriate extension, the editor would be able to display a warning in case the inertia tensor specified in a link description is not positive definite.

5 Code Generation

The ultimate purpose of our DSLs is to generate executable code, which can be effectively used in custom applications. So far we developed code generators for MATLAB and C++, which are suitable for applications ranging from quick prototypes and simulations to hard real time control. Any other programming language can in principle be supported, by developing new generators. Code generators are textual templates that are instantiated for each input model (any document of a DSL is a model). These templates are written with the Xtend language [33], which is natively supported by Xtext [10]. Xtext is in turn the workbench we use to develop the DSLs.

5.1 Coordinate transforms

The code generators of the Motion-DSL and the Transforms-DSL form a tool for the automatic implementation of coordinate transformation matrices. The output is standalone source code with the definition of the matrices requested by the user (e.g. by means of a configuration file). The main user input is typically a document of the Motion-DSL, which lists the relative pose of some reference frames of interest. Currently, our generators supports homogeneous coordinate transforms as well as transforms for spatial motion/force vectors [6], [34].

Code generation basically relieves the user from the following tasks: (1) finding the correct primitive transforms and the composition sequence that would result in the desired composite transform; (2) concretely calculating the result and implementing it in a programming language, ideally with a dependency on the variables/parameters of the problem (e.g. the status of a joint). These tasks are error prone because identifying the correct sequence and the correct form of each primitive transform is confusing (cf. Section 3.1). Although not fundamentally complicated, these tasks are tedious, time consuming and error prone, and thus preferably solved by a computer program.

These two tasks are addressed separately, to keep the

```
Robot HyL {

RobotBase base {
    inertia_params { /* ... */ }
    children { slider via SLIDE }
}

link slider {
    id = 1
    inertia_params {
        mass = 3.982
        CoM = (-0.08315, 0.06849, -0.00654)
        Ix =0.037131  Iy = 0.087473  Iz =0.112496
        Ixy=0.028543  Ixz=-0.00097   Iyz=0.003392
    }
    children { upperleg via HFE }
}
link upperleg {
    id = 2
    inertia_params { /* ... */ }
    children { lowerleg via KFE }
}
link lowerleg {
    id = 3
    inertia_params { /* ... */ }
    children {}
    frames {
        fr_foot {
            translation = (0.33, 0.0, 0.0)
            rotation = (0.0, 0.0, 0.0)
        }
    }
}

p_joint SLIDE {
    ref_frame {
        translation = (0.0, 0.0, 0.0)
        rotation = (0.0, 0.0, 0.0)
    }
}
r_joint HFE {
    ref_frame {
        translation = (0.0, 0.13, -0.08)
        rotation = (-PI/2.0, 0.0, PI/2.0)
    }
}
r_joint KFE {
    ref_frame {
        translation = (0.35, 0.0, 0.0)
        rotation = (0.0, 0.0, 0.0)
    }
}
}
```

Fig. 5. A robot model in the Kinematics-DSL format. The grammar of the language is not shown for brevity. In bold purple are the keywords of the language: for example `via` states which joint connects a child link to the current link; {p|r}_joint stand respectively for prismatic and revolute joint. Some of the inertia properties are hidden for brevity.

software simpler and more modular. Specifically, the code generator of the Motion-DSL only performs the first task, as it turns the geometric information of its documents (user input) into an abstract specification of the transforms of interest, in the form of sequences of primitive transforms. Concretely, the output of the Motion-DSL generator is in turn a document of Transforms-DSL, as the latter language defines a format that is independent of the transform type (homogeneous, spatial, etc.) and of programming languages. For example, the document in Figure 4d is generated given the one in Figure 4b. An intuitive description of relative poses is turned into a more specific, yet code independent, definition of coordinate transforms. Note that, for example, given the pose of A and B with respect to S, the generator can determine (if requested) the transforms from S to A or B, as well as the one from A to B and also all the inverses.

On the other hand, the generator of the Transforms-DSL is only responsible for the generation of actual source code, as detailed in the next subsections. Note, however, that the Transforms-DSL was not designed for the sole purpose of being an intermediate by–product of the overall code generation process. The user might very well use it explicitly, should he need to write directly a specification of transforms rather than motions; for example, in the case of debugging.

5.1.1 Basic strategy

The basic approach of the Transforms-DSL generator consists in translating the primitive transforms into matrices in the target language, and the composite transforms into products of such matrices. When there is a simple library in the target language that already implements the six primitive transforms `Rx()`, ..., `Tz()`, it is then enough to generate the correct sequence of products between the identifiers defined in the library. However, the code generator must know (by means of a configuration mechanism) the convention used by the library: by comparing it with the convention declared in the input document, the generator can autonomously decide whether the abstract element `Rz(`α`)`, for instance, corresponds to its counterpart in the library or its inverse. This point is explained in Section 3.1.

5.1.2 Symbolic simplification

In fact, the Transforms-DSL generator uses a symbolic engine to improve the efficiency of the generated code. The engine develops the products of the primitive transforms and simplifies the resulting matrix to a compact form. Afterwards, the generator for an actual programming language (e.g. MATLAB) can take advantage of the prior knowledge about the matrix: the analytical expression for each element, which elements

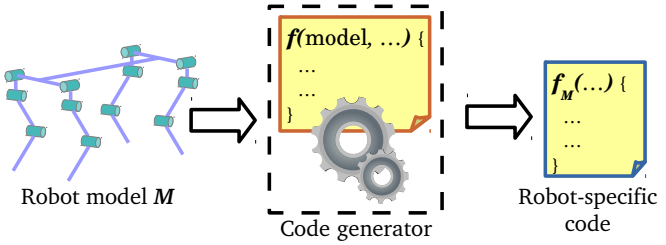


Fig. 6. Generation of an implementation of dynamics algorithms tailored for specific robots. The code generator must embed the knowledge about the abstract algorithm. A concrete implementation is achieved by turning the algorithm into code in some language, where the references to the robot model are substituted with actual values and specific instructions.

are constant, which ones are equal to others, which and how many unique trigonometric functions the matrix depends on. As a result, the generated code does not perform matrix multiplication but only evaluates the expressions for the non-constant elements. Sines and cosines and possibly other common expressions are evaluated only once. The generation of code for the symbolic engine itself, which is a prerequisite for the rest of the process, follows the basic strategy described before. In the current version of our software, the symbolic engine we use is Maxima [35].

Symbolic simplification is not intrinsically required to accomplish code generation, as the basic strategy could be used for every target language. However, it can improve the efficiency of the generated code, by reducing the amount of algebra, at the sole cost of having more complicated generators. Considering the need for efficiency we mentioned in the introduction – to address the constraints of real–time controllers or fast simulations – symbolic simplification is desirable.

5.2 Dynamics algorithms

The code generator of the Kinematics-DSL outputs implementations of rigid body dynamics algorithms: the *Composite–Rigid–Body algorithm* for the computation of the joint space inertia matrix, the *Recursive Newton–Euler algorithm* for inverse dynamics and the *Articulated–Body algorithm* for forward dynamics, all in the formulation using spatial vectors algebra [6], [34]. These algorithms are the most efficient ones for the purpose.

A robot model in the form of a document of the Kinematics-DSL, allows to resolve the dependency of the algorithms on the model itself (see Section 3.2). That is, the generated implementation of a dynamics algorithm is no longer parametrized with the robot model, but *implicitly embed* the information of a specific robot (see Figure 6).

Robot–specific implementations have the advantage of being faster since they realize a simplified and optimized version of the algorithm. In the case of C++, the code is also real–time safe, since it performs purely numerical computations with predictable execution time (no dynamic memory allocation, for example). A robot-specific, optimized implementation is ideal to run fast simulations of the robot or to deploy the code on the actual real hardware for control.

The code generator of the Kinematics-DSL does *not* use symbolic simplification, but still applies optimizations which are possible thanks to the prior knowledge of the kinematics of the mechanism. Such knowledge allows to avoid, in the generated code, all the logic that would be required to be general: the dimension of vectors and matrices is a compile time constant; loops are unrolled; all the boolean tests and the corresponding branches (i.e. `if, then, else`) are simply replaced with the implementation of the sole case which is known to be the right one. Similarly, standard optimizations, which only apply to special cases, appear in the generated code without a test for their applicability, since it is determined in advance. For example, the motion subspace matrix $\mathbf{S}$ – which describes the motion freedom allowed by a joint – for prismatic and revolute joints is a single column vector with a single non-zero element; therefore operations with this matrix can be greatly simplified [6].

Branch–induced sparsity is another significant example of optimization. The term refers to the sparsity pattern of some matrices related to the robot kinematics/dynamics, which is due to the branching of the kinematics structure of the robot. Specialized algorithms – for both the computation of the matrix itself and for the operations involving it – avoid any computations for the elements known to be zero, and might be significantly more efficient than general purpose ones [6], [36].

The code generator of the Kinematics-DSL implements the *Composite–Rigid–Body algorithm* for the calculus of the Joint–Space Inertia Matrix (JSIM) $\mathbf{H}$, which exhibits branch–induced sparsity. The generated code not only avoids to calculate the zero elements of $\mathbf{H}$, but also does not need to *identify* the sparsity at run-time. The information about the branching is exploited during code generation, and the output is a tailored algorithm that embeds implicitly knowledge about the sparsity. Efficient code is generated also for the inverse $\mathbf{H}^{-1}$; it implements the $\mathbf{L}^T\mathbf{L}$ factorization of $\mathbf{H}$, where $\mathbf{L}$ is a lower triangular matrix that preserves the same sparsity pattern as $\mathbf{H}$ [37], and a custom routine for the inversion of a lower triangular matrix that takes advantage of the sparsity as well.

6 RobCoGen

The Robotics Code Generator (RobCoGen) is composed of the code generators of the DSLs we have described in this paper. The main input for RobCoGen is a robot model in the form of a document of the Kinematics-DSL, which is the only source file the user has to develop and maintain manually. Another, secondary input, is a sort of configuration file stating which

```
Robot HyL
Frames {
  fr_base, fr_slider, fr_upperleg,
  fr_lowerleg, fr_foot
}
Transforms {
  left_frame= fr_foot  right_frame= fr_lowerleg
  fr_base  <-  fr_foot // alternative syntax
  fr_base  ->  fr_foot
}
Jacobians {
  base= fr_base   target= fr_foot
}
```

Fig. 7. A sample configuration file, stating which transforms/Jacobians to generate.

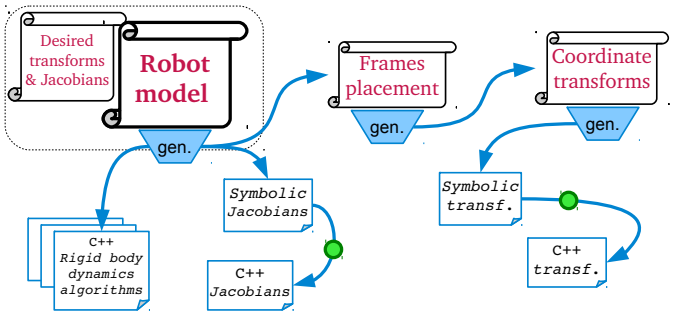


Fig. 8. Overview of the RobCoGen workflow. Scrolls represent DSLs documents (left to right: configuration DSL, Kinematics-DSL, Motion-DSL, Transforms-DSL); the trapezoids are code generators, while the sheets are generated source files. The green circle indicates a code generation step requiring the symbolic engine. The user only provides the robot model and a configuration file, as the other DSLs documents are also generated.

coordinate transforms and Jacobians the user would like to generate. The file format is another simple DSL we have not described for brevity, however Figure 7 shows an example.[6]

Figure 8 gives an overview of the components and the data flows involved in the generation of robot–specific code. The code generator of the Kinematics-DSL directly takes care of the dynamics algorithms, but also orchestrates the other generators for the coordinate transforms, as if it was a user of the other languages. We mentioned before that the generator for transforms is a standalone software not limited to the case of articulated robots, however that is the use case we focus on here.

In this regard, the figure shows that the Kinematics-DSL software also generates a document of the Motion-DSL, which describes the pose of the joint frames of the robot. This step is really just a change of representation of the same information already encoded in the robot model, via the six parameters in each `joint` paragraph. The motivation for this additional, intermediate step, is the reuse of the code generator of the Motion-DSL. Although it would be possible to turn the joint parameters directly into a Transforms-DSL document, that would require to replicate in the Kinematics-DSL generator some functionality anyway available in the Motion-DSL generator. Keep in mind that the Motion-DSL itself is motivated by the need for code generation for transforms, which exists independently from the case of robotics (Section 5.1). With respect to the Motion-DSL, the Kinematics-DSL has a less general way of expressing the pose of a frame, to keep the grammar simple and thus the documents more readable. However, a possible development of the Kinematics-DSL could optionally allow to specify the joint frame pose directly with the syntax of the Motion-DSL, for maximum flexibility. An example of this transformation is visible in Figure 9.

In turn, the generator of the Motion-DSL outputs a list of abstract transforms in the form of a document of the Transforms-DSL. The list also depends on the configuration file we mentioned at the beginning of the section. The actual implementation of the coordinate transforms is finally generated by the Transforms-DSL. These last steps are explained in Section 5.1. When the transforms refer to frames attached to different links of the robot (which is usually the case), they are implemented with a dependency on variables that correspond to the status of the appropriate joints.

6. Some transforms are always generated regardless of the user configuration, as they are required by the dynamics algorithms.

6.1 Geometric Jacobians

In this work, the term Jacobian refers to a matrix that maps joint space velocities to Cartesian velocities of some point of interest located on the robot, according to the well known formula:

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\ \dot{\mathbf{q}}$$

Although the code generated with RobCoGen mostly uses the spatial vector algebra [34], at the current version the generated Jacobians are not spatial. We use instead the classical definition of geometric Jacobian that would be familiar to most roboticists [31]: the generated Jacobians are $6 \times n$ matrices; the six elements of the resulting velocity vectors are coordinates of an arbitrary reference frame, which account for the 3D linear velocity of the point and the 3D angular velocity of the robot link the point is fixed to. The number of columns n is the same as the number of degrees of freedom of the kinematic chain whose joints contribute to the velocity of the point.

The user selects a set of desired Jacobians by listing pairs in the form <frame–point of interest> (e.g. a point on the foot and the torso frame, in a humanoid). Since Jacobians are intrinsically defined on a robot model it seemed sensible to implement the code generation within the Kinematics-DSL. The procedure is as follows: at first the program generates code for the symbolic engine; this code calculates the Jacobian matrix given the appropriate coordinate transforms, generated before (geometric Jacobians can be computed from direct kinematics

```
r_joint HFE {
    ref_frame {
        translation = (0.0, 0.13, -0.08)
        rotation = (-PI/2.0, 0.0, PI/2.0)
    }
}
```

(a) Specification of the pose of a joint-frame in a robot model (see the full example in Fig. 5).

```
// [...]
Convention = local
upperleg -> HFE : try(0.13) trz(-0.08)
                  rotx(-PI/2.0) rotz(PI/2.0)
```

(b) Same information in a document of the Motion-DSL. The upperleg is the robot link supporting the joint HFE. However, for the Motion-DSL that is just a name for a reference frame.

Fig. 9. The transformation of the geometric information from a robot model to a motions model (see Fig. 8)

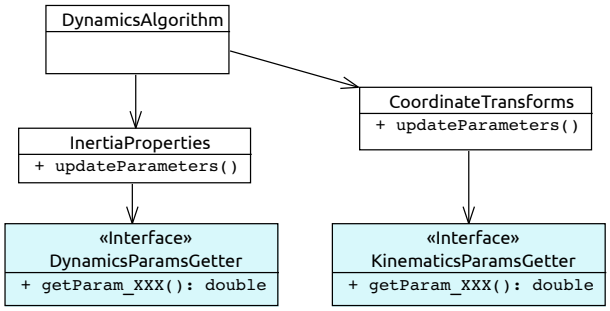


Fig. 10. Simplified layout of the C++ classes dealing with dynamically changing parameters, as generated by RobCoGen (the Jacobians class is not shown). The illustrated dependencies enable the kinematics/dynamics to consistently reflect a change of the parameters. Changes only depends on the user implementation of the interfaces (in light blue).

[31]). Afterwards the symbolic code is actually interpreted by the symbolic engine and actual code is generated in the same way as for coordinate transforms, as described in Section 5.1.2.

6.2 Parameters and robot classes

Thanks to the support for parameters in our DSLs (see Section 4.3), a robot model can be defined without specifying in advance all the values of its properties. A parametric robot model can in fact be thought of as a *class* of equivalent robots. The equivalence relation would be formally defined by the isomorphism between the kinematic tree (i.e. a graph) of the robots, plus the additional constraint that corresponding joints must be of the same type (i.e. prismatic or revolute). The efficiency of the generated code in the case of a parametric model is not affected significantly, since the optimizations applied by RobCoGen (section 5) depend mainly on the kinematic tree topology and the joints type, which are invariants of the class.

Parameters represent non-constant numerical properties of the robot, which vary at a lower average rate than variables. The joint positions are the only variables in a robot model. A parametric model can be very effective for simulations supporting the design process of new robots, as interactive tests or automatic optimization techniques can be applied to determine ideal values for the unknown robot properties. A parametric kinematics/dynamics engine may be useful also for existing real robots with some options for structural reconfiguration. In any case, the distinction parameter–variable based on the rate of change is sound, as in general the joints move continuously for a time interval while the parameters are fixed.

For the actual computation of the kinematics/dynamics, the values of the parameters must be resolved at run–time, according to some mechanism. We shall describe here such a mechanism as implemented in the C++ code generated by RobCoGen, which applies the well known principle of *programming to an interface* [38]. RobCoGen generates two interfaces (i.e. virtual C++ classes), respectively for kinematics and dynamics parameters, which just declare a getter method for each parameter defined in the robot model. Other generated classes depend on such interfaces, as they invoke the methods at run–time to get the actual value of the parameters. The end user of RobCoGen is required to provide a concrete implementation of the interfaces to be used in an actual application.

With this approach, the kinematics/dynamics engines (generated by RobCoGen) do not hard-code any policy to resolve the parameters, whereas the user has ample flexibility to develop different strategies for the purpose. For example, a simple class that parses a configuration file, or a complex component taking part of a batch of non-interactive simulations, which varies the parameters automatically. It is also possible to have a single component implementing both interfaces, possibly to impose some kind of coupling among the parameters (e.g. a mass that increases whenever the length of the link increases).

The classes that rely on the getter interfaces (if the robot model is parametric at all) are those directly affected by a change of the dynamics/kinematics parameters: the class that encapsulates the inertia properties of the links, the one containing the coordinate transforms and the class with the Jacobians. These classes expose the method `updateParameters()`, used by an external component aware of a variation in the parameters, to signal the event. The method implementation requests the interface for fresh values, and triggers the internal update logic of the class. See Figure 10. This mechanism conceptually realizes a *push*–based communication, where a component – within the application using the RobCoGen code – actively *provides* the updated parameters when they change. This mechanism is more suitable than having the kinematics/dynamics algorithms to continuously poll for fresh values, as the average rate of change of parameters is generally

much lower than the execution rate of the same algorithms (which would follow the rate of change of the joint status).

7 Experimental results

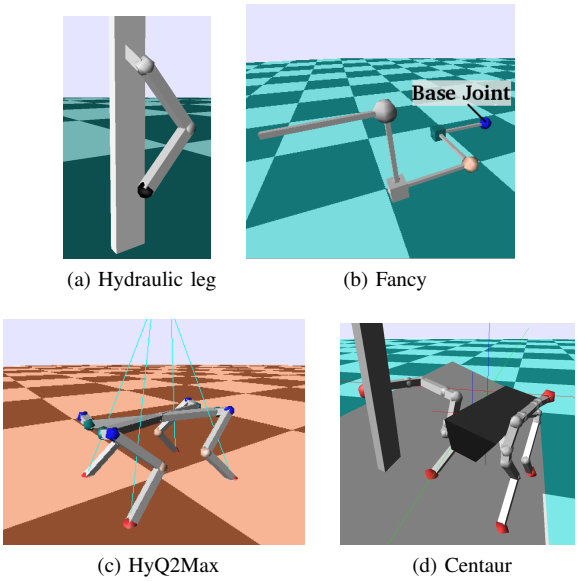
This section discusses the concrete usage of RobCoGen and the benefits for the end user. The numerical results we show refer primarily to the performance of the generated code, as it is difficult to find metrics and tests to measure usability and flexibility. For these tests we usually compare C++ implementations, but our generator can currently emit also MATLAB and MAXIMA code; at the price of developing more generator templates, potentially any language can be supported.

7.1 General remarks

Results from our experience with RobCoGen are promising. Creating new robot descriptions takes little time, since the Kinematics-DSL is simple and intuitive. Code generation is not affected by occasional mistakes, and is more reliable than manual development. Consistency between source code in different languages, generated given the same input (e.g. a robot model), is also helpful. For example, one can test Jacobian matrices in MATLAB with rapidly prototyped simulations, while using the same quantities in C++ in a real robot controller, with a strong confidence that they will behave in the same way. Correctness and consistency allow the user to focus the initial development and debugging efforts on the models, simpler than code hence easier to inspect.

To ease the process of learning RobCoGen, for end users as well as possible contributing developers, it is valuable for the toolchain to be as clear and as observable as possible. With this requirement in mind, we designed simple DSLs addressing well defined and confined problems, so that it is relatively easy to perform many trials and understand the behavior of the generators. Clarity in the generated code is also of great help for debugging and to give the user more chances to understand and thus be satisfied with the tool. Model–based optimizations (Section 5.2) as opposed to exhaustive symbolic simplification, permit to use vector arithmetic in the generated code, which is thus more compact and readable. Vector arithmetic is in fact an opportunity to exploit existing libraries *specialized* for the purpose. In the case of C++, RobCoGen uses Eigen [39], a sophisticated library for efficient linear algebra. MATLAB supports natively vector operations. A few more remarks on symbolic simplification can be found in Section 2.

On the other hand, a factor that in our experience negatively influenced the acceptance of RobCoGen among other users, is the intrinsic lack of generality in the generated code. While beneficial for efficiency (one of the primary points of this paper), it is perceived as incompatible with the development of generic code, i.e. high level algorithms addressing articulated robots in general. However, languages like C++ do have tools that allow to write generic interfaces to wrap specialized code – such as the output of RobCoGen–, namely templates and traits [40]. It is true, though, that these techniques require a deeper knowledge of the language and make coding harder and slightly more verbose.



(a) Hydraulic leg (b) Fancy

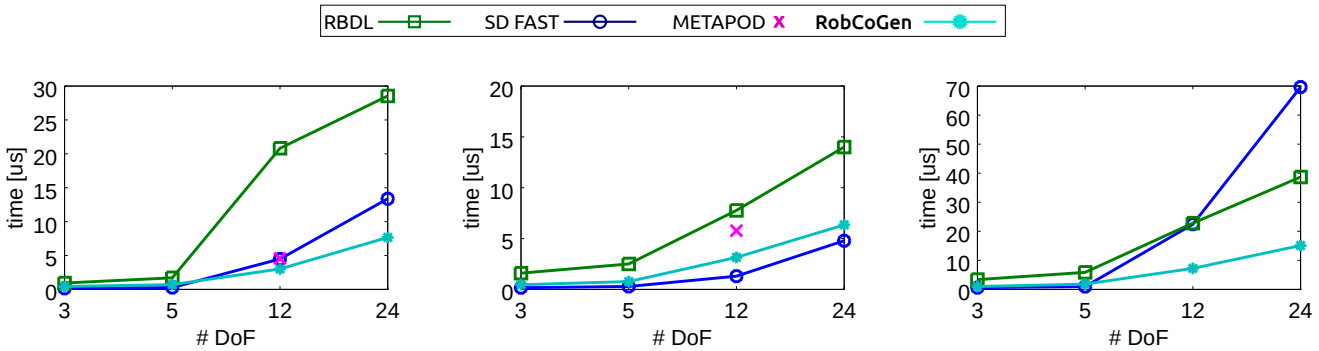
(c) HyQ2Max (d) Centaur

Fig. 11. Some of the simulations created with SL and RobCoGen.

7.2 Validation

In order to validate the numerical correctness of the generated code we have tested it against other existing and established implementations, automating the comparison of the numerical output for different robot models and different inputs (e.g. the joint status $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$). We used the MATLAB code available on Roy Featherstone's web page about spatial vector algebra and dynamics algorithms [30], and also the SL software package, which has been in development for more than fifteen years and is used in several research labs for simulations and hard real time model–based control of real robots [41]. The C++ code has also been compared with all the other tools subject to our efficiency comparison, described in the next section.

Furthermore, we are currently using RobCoGen as the dynamics engine for the simulation of all the robots developed in our lab, and also in the controllers of the real robots. Our simulations are currently based on SL, which also includes components for the low level joint control and for trajectory generation. To validate RobCoGen, we modified SL by replacing the dynamics engines and the kinematics computations with our generated code, leaving the rest unmodified. The C++ code generated by RobCoGen is self contained and with a neat interface, therefore it is easily composable with an existing application. So far, our experience with these simulations, also in comparisons with experiments with the real robots, is positive (Figure 11).



#DoF	SD-FAST	RBDL	**RobCoGen**
3	0.15	0.94	0.43
	0.36	*2.19*	*1*
5	0.27	1.71	0.70
	0.38	*2.42*	*1*
12	4.49	20.82	3.02
	1.49	*6.90*	*1*
24	13.37	28.54	7.67
	1.74	*3.72*	*1*

(a) The joint-space inertia matrix

SD-FAST	RBDL	**RobCoGen**
0.17	1.60	0.46
0.36	*3.47*	*1*
0.29	2.50	0.77
0.37	*3.24*	*1*
1.30	7.77	3.15
0.41	*2.47*	*1*
4.79	14.00	6.35
0.75	*2.21*	*1*

(b) Inverse dynamics

SD-FAST	RBDL	**RobCoGen**
0.60	3.38	1.00
0.60	*3.38*	*1*
0.98	5.87	1.81
0.54	*3.24*	*1*
22.44	22.81	7.24
3.10	*3.15*	*1*
69.65	38.69	15.10
4.61	*2.56*	*1*

(c) Forward dynamics

Fig. 12. Performance comparisons between the C++ code generated by RobCoGen and RBDL, SD-FAST, Metapod– for three dynamics routines. The plots at the top show the absolute, average execution time in μs versus the number of degrees of freedom of the robot model. The tables at the bottom refer to the same results, and show the absolute execution time (regular font) as well as the ratio with the execution time of RobCoGen (*italic* font).

7.3 Performance comparisons

We performed several speed comparisons between our generated C++ code and other implementations, to demonstrate that our approach not only provides ease of use and flexibility, but also high run time performances. The execution times shown in the graphs are more relevant as a comparison rather than in absolute terms, since they also depend on the hardware.[7]

We measured the execution time of the routines that calculate inverse dynamics, forward dynamics and the joint-space inertia matrix. We tried our best to make fair comparisons, by measuring the execution time of the minimum amount of code required to compute the desired quantity, for each of the engines we tested. Execution time was measured simply with the `high_resolution_clock` of the standard C++ library: the difference between two absolute-time samples (function `now()`), taken before and after the code snippet of interest, gives an estimate of its execution time. In fact, in our test programs the second sample is taken only after 10^3 repetitions of the same snippet, in order to make the execution time longer and cope with the inaccuracy of the clock for very short time intervals.[8] This procedure is repeated several times, and in every iteration the input variables are randomly changed. The average execution time is finally calculated for each engine.

We compared RobCoGen with SD-FAST, RBDL, and, whenever possible, with Metapod. These programs/libraries are good terms of comparison as they are representative of different approaches than RobCoGen. SD-FAST is a well know program that uses symbolic simplification, whereas RBDL is a generic library whose API requires the robot model at runtime. Metapod uses C++ metaprogramming to maximize efficiency, so it is partially similar to RobCoGen. More information about these tools can be found in Section 2. The results of our comparison tests are displayed in Figure 12.

The robot models we used in the tests are: a single leg of our HyQ robot [32] mounted on a vertical slider (3 DoFs); a fictitious robot with both prismatic and revolute joints (5 DoFs); our new hydraulic quadruped HyQ2Max (12 DoFs) [42]; a centaur robot, obtained by adding two of our new hydraulic arms [43] on the body of HyQ (24 DoFs). A screenshot from the simulator of each of these models is visible in Figure 11.

The plots show that RobCoGen performs well, and it is always comparable with SD-FAST. In fact, in the case of the inertia matrix and forward dynamics RobCoGen seems to scale better with the degrees of freedom, and it is faster than SD-FAST. As expected, RobCoGen is also faster than RBDL since both engines use the same algorithms but RobCoGen provides robot-specific optimized implementations. Metapod performs very well too; only few samples are shown in the plots because, to the best of our understanding of the library,

7. All the tests were executed on a Intel(R) Core(TM)2 Duo CPU, P8700 @ 2.53GHz, with 4GB of RAM

8. The difference between two immediately subsequent calls to `now()` is far from being zero. For this offset to be negligible, one should measure time intervals a few order of magnitude longer.

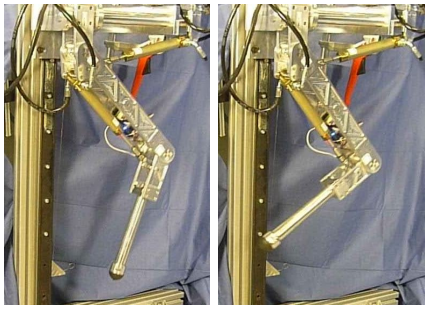


Fig. 13. The hydraulic leg of the HyQ robot.

Metapod does not support prismatic joints (which appear in the first two robot models), nor more than five children for a single link (last robot model) nor any algorithm for forward dynamics.

Additional results from older comparison tests can be found in [44], [45] and [21].

7.4 Hard real time control

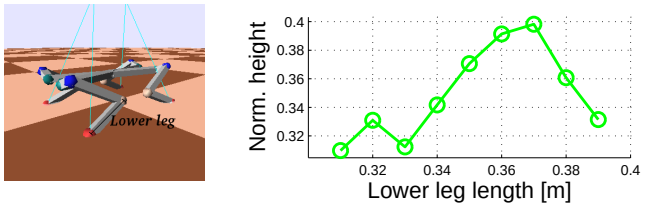
The overall execution times described above, plus the feature that only predictable-duration instructions are performed, are alone good evidence of the compatibility of the generated C++ code with hard real time constraints. In order to provide a more rigorous argument for this claim, we tested the execution of the generated code within the control processes of actual robots, in a real time environment. Figure 13 contains a few snapshots of an experiment with the hydraulic leg of HyQ that consists in tracking a very fast (5Hz) sinusoidal position trajectory at the joints.

The desired motion commanded to the robot is so fast that proper position tracking is only possible using an inverse dynamics based controller (this experiment was in fact originally designed to demonstrate the effectiveness of our force controller; proper force control is required to use inverse dynamics [46]). The RobCoGen implementation of inverse dynamics for the leg was executed at 250 Hz without issues, especially without violating the real time constraints (a violation such as a dynamic memory allocation would be signaled by the real time operating system).

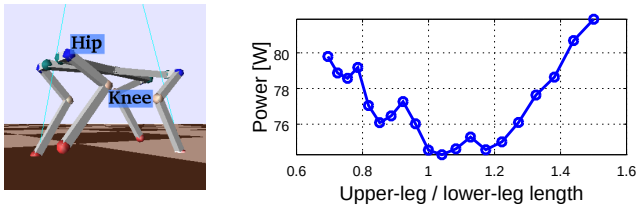
Another example of the use of RobCoGen on a real robot can be found in [47]. In this work the authors propose a planning and execution framework for quadrupedal locomotion over rough terrain. The implementation of the required kinematics/dynamics routines on the real HyQ robot is based on RobCoGen.

7.5 Robot design

The kinematics/dynamics engines generated by RobCoGen allow to investigate task–based optimizations of the robot design, thanks to the support for parametric models (see Section 4.3 and 6.2). This section describes a couple of simple



(a) Squat jump varying the length of the lower legs (the upper leg is $36.4cm$ long). The plot shows the distance traveled by the trunk, from the lift-off position to the apex of the jump, normalized by the lower leg length. The trunk is roughly $90 \times 30cm$ long.



(b) Mechanical power employed by the hip and knee of one leg of the quadruped during trotting. Each data point is the average value over a single step. The x axis shows the ratio of the length of the upper and lower leg.

Fig. 14. Two examples of simulations with a parametric quadruped model.

yet illustrative examples about this feature, which give an idea of the possibilities of the parametric engines. We ran the tests with a model of a preliminary design of our new hydraulic quadruped HyQ2Max [42]. The results for both examples are illustrated in Figure 14.

The first example stems from a study about the torque requirements for a quadruped performing a squat jump, with respect to the leg size [48]. In that work, a few assumptions on the robot kinematics simplify the problem and make an analytical analysis possible. In our case instead, we can consider a more realistic scenario by leveraging the flexibility of our numerical simulator. We performed repetitive simulations where the length of the lower leg of the robot is changed automatically before performing the squat jump. The purpose is to find a possibly optimal value with respect to the effectiveness of the motion, measured as normalized jump height (Fig. 14a). As the leg length varies in a narrow range, we assume a constant mass of the links (note that it would have been perfectly possible to include a variable mass also). The jump is achieved by imposing a vertical force on the trunk, that is then mapped into desired joint torques by a simplified version of the trunk controller described in [49].

The second example involves the simulation of a trotting gait with the same quadruped model [49]. We analyze the effect of moving the position of the knee with respect to the trunk, without altering the robot height; this effect is achieved by varying the length of the two links of the leg while keeping their sum constant. Since trotting is a periodic motion (as opposed to the squat jump), the results refer to a single simulation run for a given time interval; to avoid

injecting abrupt disturbances in the trot controller, the lengths are changed smoothly during the motion. Figure 14b shows the plot of the mechanical power requirements of the hip and knee joints, summed together and averaged over a single trot step.

In both examples the results basically tell that there seem to be an optimal configuration corresponding to the upper and lower leg having the same length. However, note that in the case of trotting changing the desired forward velocity of the robot also changes the power profile, suggesting that the relations among the various quantities are not so intuitive, and require further investigation.

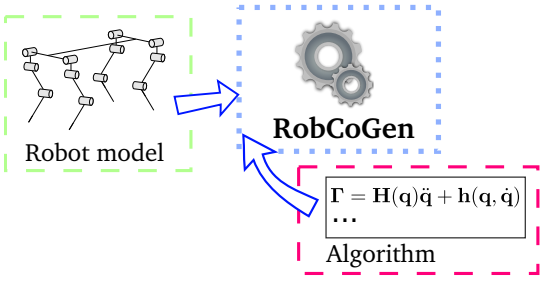


Fig. 15. Ideal inputs of RobCoGen. Having an independent model of the algorithm, in addition to the model of the robot, would make the code generator more extensible.

8 Future Developments

One of the main limitations of the current version of RobCoGen is the lack of support for mechanisms with kinematic loops. A kinematic loop is a closed chain of connected links. The topology of these mechanisms cannot be modeled with a tree, but a general graph is required. Rigid body dynamics algorithms for this class of mechanisms are also known in the literature [6], and they could be implemented within RobCoGen. However, the new feature would require some modifications starting already from the domain model presented in Section 3.2. For example, the concept of loop joint has to be included. A loop joint is a joint that does not induce the parent–child relationship between two links, it just adds a further kinematic constraint. The concept of *spanning tree* may be also useful. A spanning tree is a connected graph without loops that include all the nodes of the original graph but drops the arcs that close the loops, i.e. the loop joints. Choosing the spanning tree is really choosing which joints are to be considered loop joints. These and other improvements of the domain model would allow to robustly extend the Kinematics-DSL and the code generator. The class of mechanisms without loops include anyway a large set of articulated robots, which is why the first version of RobCoGen does not address loops; starting with a simpler case was another reason for our choice.

In our experience with DSLs, we realized that a limitation of code generation based on textual templates is the coupling between the target programming language and the target algorithm. That is, the knowledge about the algorithm is embedded implicitly in the code generation template, which forces the developer to repeat totally similar logic in templates targeting the same algorithm but in a different language. In our case these repetitions pertain not only the algorithms themselves, but also the robot–specific optimizations (Section 5.2), which obviously do not depend on the target language. For example, the fact that the velocity of the first link of a fixed base manipulator is due only to the joint velocity (the parent velocity is zero); it is a special case to be implemented with simpler code, regardless the language. Ideally, both the robot model *and* the dynamic algorithm would be real inputs to the code generation infrastructure, as illustrated in Figure 15. A possible approach to tackle this issue is a new DSL to implement rigid body dynamics algorithms in a neutral format; a sort of general–purpose pseudo–code, with additional directives specific for the case of dynamics algorithms. Adding a new algorithm to RobCoGen would then be a matter of expressing it with this DSL. A new target language would require mainly development on the generators of this DSL. Evaluating the feasibility and the effectiveness of such a solution is part of our future work.

A technological factor that in our experience slowed the development of RobCoGen was the symbolic engine – in our case Maxima. The complexity of integrating the symbolic engine was simply due to the need for programmatic interaction with an external application. In addition, such application is originally meant to be used interactively, and the textual input/output interface forced us to develop solutions for the robust interpretation of strings, adding further complexity to the generator as a whole. Future developments of RobCoGen might switch to more sound technological solutions better suitable for programmatic interaction (e.g. the Python SymPy [50]). However, as mentioned already in this paper (see Sections 2, 5.1.2, 5.2) RobCoGen uses symbolic simplification only marginally, for coordinate transforms.

9 Summary and Conclusions

This paper presented RobCoGen, a Java program that generates efficient code for articulated robot kinematics and dynamics. The program is based on three Domain Specific Languages (Motion-DSL, Transforms-DSL, Kinematics-DSL) for the description of high level models, and corresponding code generators that turn the models into executable code. Despite apparently overlapping, the languages (i.e. the underlying models) refer to different concepts that as such should be modeled separately (e.g. rotations of a rigid body and rotation matrices)[9]; this separation results in code generators that fulfill independent tasks yet can be used effectively together.

9. A limited, actual semantic overlap between the Motion-DSL and the Kinematics-DSL is mentioned in Section 6 and Figure 9.

The Kinematics-DSL defines a new, simple file format for the description of articulated robot models conforming to the kinematic tree meta–model. The language exposes only concepts pertaining the kinematics/dynamics of the robot. After reading a model, RobCoGen automatically generates a robot–specific implementation of three common rigid body dynamics algorithms, as well as Jacobians and coordinate transforms. The implementation is optimized, and it is suitable for both simulation and real control. RobCoGen takes care of a significant part of the software infrastructure typically required in robotics applications, such as model based controllers, and thus allows the roboticist to speed up the development process and focus on actual research questions.

When plugged into a simulator, the kinematics/dynamics engines generated by RobCoGen allow to investigate task–based optimizations of the robot design, thanks to full support for parametric models. That is, any number of the geometrical or inertial properties of the robot (e.g. link lengths) can be changed at run–time; this feature is ideal for multiple interactive simulations as well as non–interactive, automatic optimization and learning techniques, which can discover optimal values for the parameters.

RobCoGen compares to existing tools which use symbolic simplification to generate optimize code, but it leverages the modern technology of DSLs. The result is a tool that is easy to use and easier to extend, e.g. by supporting more programming languages. External DSLs have a custom syntax tailored to a specific purpose, such as describing the structure of an articulated robot, and thus can be clear and neat; they expose explicitly high level domain concepts and thus are effective as specification languages. In addition, with modern workbenches such as Xtext it is easy to create the software infrastructure for new languages, including code generators. Therefore DSLs are an effective technology to implement a tool like RobCoGen, whose purpose is to allow the users to focus their effort on a high level specification of the problem rather than on the low level coding.

The experimental results showed that the efficiency of the generated code (C++) is comparable with the case of symbolic simplification, and is higher than with a general purpose dynamics library. RobCoGen has been successfully used in our lab to implement the kinematics/dynamics engines of all our robot simulations, as well as in some of the controllers running on real robots. These results are a good evidence of the actual applicability of our tool for concrete use cases. In the design of RobCoGen the ease of use and the generality of the generated code were equally important criteria as performance.

In conclusion, we believe our work to be a useful contribution to the robotics community, as it constitutes a new technological opportunity for the concrete implementation of the software of a robot. It is also a notable example of exploitation of Domain Specific Languages and code generation; as far as RobCoGen is concerned, DSLs were in turn identified as a technological opportunity, which proved to be effective.

Acknowledgements

This research has been funded by the Fondazione Istituto Italiano di Tecnologia. J. Buchli is supported by a Swiss National Science Foundation Professorship.

References

[1] E. Rohmer, S. P. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) 2013*, 2013, pp. 1321–1326. 1

[2] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3, 2004, pp. 2149–2154. 1

[3] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE Journal on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987. 1

[4] N. Hogan, "Impedance control: An approach to manipulation: Part II – Implementation," *Dynamic Systems, Measurement, and Control*, vol. 107, pp. 8–16, 1985. 1

[5] M. Frigerio. The Robotics Code Generator wiki. [Online]. Available: bitbucket.org/mfrigerio17/roboticscodegenerator/wiki/Home 1, 4.2

[6] R. Featherstone, *Rigid Body Dynamics Algorithms*. Springer, 2008. 1, 2, 3.1, 3.2, 3.2, 5.1, 5.2, 5.2, 8

[7] R. Smith. (2013) The Open Dynamics Engine simulation library. [Online]. Available: www.ode.org 1.1

[8] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010. 1.2

[9] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots*, 2014. 1.2, 2

[10] S. Efftinge *et al.* (2013) Xtext. [Online]. Available: www.eclipse.org/Xtext/ 1.2, 4.4, 5

[11] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis, "BRICS – best practice in robotics," in *IFR Int. Symposium on Robotics (ISR)*, 2010. 2

[12] C. Schlegel, T. Haßler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *Int. Conf. on Advanced Robotics (ICAR)*, July 2009, pp. 1–8. 2

[13] T. D. Laet, W. Schaekers, J. de Greef, and H. Bruyninckx, "Domain specific language for geometric relations between rigid bodies targeted to robotic applications," in *3rd Int. Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, 2012. 2

[14] R. W. Toogood, "Efficient robot inverse and direct dynamics algorithms using microcomputer based symbolic generation," in *IEEE Int. Conf. on Robotics and Automation*, vol. 3, May 1989, pp. 1827–1832. 2

[15] M. Sherman and D. Rosenthal. (2013) SD/FAST. [Online]. Available: www.sdfast.com/ 2

[16] CEREM. (2013) Robotran. Center for Research in Mechatronics (CEREM), iMMC, UCL. [Online]. Available: www.robotran.be/ 2

[17] M. Felis. The Rigid Body Dynamics Library. [Online]. Available: http://rbdl.bitbucket.org/ 2

[18] The Orocos Project. [Online]. Available: http://www.orocos.org/ 2

[19] M. Naveau, J. Carpentier, S. Barthelemy, O. Stasse, and P. Souères, "Metapod - template meta-programming applied to dynamics: CoP-CoM trajectories filtering," in *Int. Conf. on Humanoid Robotics*, 2014. 2

[20] R. Philippsen, L. Sentis, and O. Khatib, "An open source extensible software package to create whole-body compliant skills in personal mobile manipulators," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, September 2011, pp. 1036 –1041. 2

[21] M. Frigerio, J. Buchli, and D. G. Caldwell, "Code generation of algebraic quantities for robot controllers," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2012. 2, 7.3

[22] T. D. Laet, S. Bellens, R. Smits, E. Aertbelien, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies: Semantics for standardization," *IEEE Robotics and Automation Magazine*, 2012, accepted for publication. 2

[23] T. D. Laet, S. Bellens, H. Bruyninckx, and J. D. Schutter, "Geometric relations between rigid bodies (part 2) - From semantics to software," *IEEE Robotics and Automation Magazine*, 2012. 2

[24] OpenHRP group. (2013) OpenHRP. [Online]. Available: http://fkanehiro.github.io/openhrp3-doc/en/index.html 2
[25] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 2
[26] T. Foote, "tf: The transform library," in *IEEE Int. Conf. on Technologies for Practical Robot Applications (TePRA)*, April 2013, pp. 1–6. 2
[27] Modelica Association. (2013) Modelica and the modelica association. [Online]. Available: www.modelica.org 2
[28] M. Fowler, *UML distilled*. Addison-Wesley, 2003. 3
[29] P. Corke, "A robotics toolbox for Matlab," *IEEE Robotics Automation Magazine*, vol. 3, no. 1, pp. 24–32, mar 1996. 3.1
[30] R. Featherstone. (2013) Spatial vectors and rigid-body dynamics. [Online]. Available: royfeatherstone.org/spatial/ 3.1, 7.2
[31] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics. Modelling, Planning and Control*, M. J. Grimble and M. A. Johnson, Eds. Springer, 2009. 3.2, 6.1
[32] C. Semini, N. G. Tsagarakis, E. Guglielmino, M. Focchi, F. Cannella, and D. G. Caldwell, "Design of HyQ – a hydraulically and electrically actuated quadruped robot," *IMechE Part I: J. of Systems and Control Engineering*, vol. 225, pp. 831–849, 2011. 4.2, 7.3
[33] S. Efftinge and S. Zarnekow. (2013) The Xtend language. [Online]. Available: www.eclipse.org/xtend/ 5
[34] R. Featherstone, "A beginner's guide to 6-d vectors," *IEEE Robotics & Automation Magazine*, vol. 17, no. 3, pp. 83–94, 2010. 5.1, 5.2, 6.1
[35] Maxima. (2011) Maxima, a Computer Algebra System. version 5.25.1. [Online]. Available: maxima.sourceforge.net/ 5.1.2
[36] R. Featherstone, "Exploiting sparsity in operational-space dynamics," *The Int. J. of Robotics Research*, vol. 29, no. 10, 2010. 5.2
[37] ——, "Efficient factorization of the joint-space inertia matrix for branched kinematic trees," *The Int. J. of Robotics Research*, vol. 24, no. 6, pp. 487–500, 2005. 5.2
[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1997. 6.2
[39] G. Guennebaud, B. Jacob *et al.* (2013) The Eigen library v3. [Online]. Available: http://eigen.tuxfamily.org 7.1
[40] N. C. Myers, "A new and useful technique: "traits"," *C++ report*, vol. 7, no. 5, pp. 32–35, June 1995. 7.1
[41] S. Schaal, "The SL simulation and real-time control software package," CLMC lab, University of Southern California, Tech. Rep., 2009. 7.2
[42] C. Semini, J. Goldsmith, B. U. Rehman, M. Frigerio, V. Barasuol, M. Focchi, and D. G. Caldwell, "Design overview of the hydraulic quadruped robots HyQ2Max and HyQ2Centaur," in *The Fourteenth Scandinavian Int. Conf. on Fluid Power (SICFP)*, 2015. 7.3, 7.5
[43] B. U. Rehman, M. Focchi, M. Frigerio, J. Goldsmith, D. G. Caldwell, and C. Semini, "Design of a hydraulically actuated arm for a quadruped robot," in *Int. Conf. on Climbing and Walking Robots (CLAWAR)*, 2015. 7.3
[44] M. Frigerio, J. Buchli, and D. G. Caldwell, "A domain specific language for kinematic models and fast implementations of robot dynamics algorithms," in *2nd Int. Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, September 2011. 7.3
[45] ——, "Model based code generation for kinematics and dynamics computations in robot controllers," in *7th workshop on Software Development and Integration in Robotics (ICRA SDIR VII)*, 2012. 7.3
[46] T. Boaventura, C. Semini, J. Buchli, and D. G. Caldwell, "Actively–compliant leg for dynamic locomotion," in *Int. Symposium on Adaptive Motion of Animals and Machines (AMAM)*, 2011. 7.4
[47] A. Winkler, C. Mastalli, I. Havoutis, M. Focchi, D. G. Caldwell, and C. Semini, "Planning and execution of dynamic whole-body locomotion for a hydraulic quadruped on challenging terrain," in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, May 2015. 7.4
[48] C. Semini, H. Khan, M. Frigerio, T. Boaventura, M. Focchi, J. Buchli, and D. G. Caldwell, "Design and scaling of versatile quadruped robots," in *Climbing and Walking Robots (CLAWAR)*, 2012. 7.5
[49] V. Barasuol, J. Buchli, C. Semini, M. Frigerio, E. R. D. Pieri, and D. G. Caldwell, "A reactive controller framework for quadrupedal locomotion on challenging terrain," in *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2013. 7.5, 7.5
[50] S. D. Team. SymPy, a python library for symbolic mathematics. [Online]. Available: http://www.sympy.org/en/index.html 8

Marco Frigerio received his B. Sc. and M. Sc. degrees in computer science from the University of Milano Bicocca, respectively in 2006 and 2008, and the Ph. D. degree in Robotics from the Istituto Italiano di Tecnologia (IIT) in 2013. He is currently a Post-Doc researcher at the Dynamic Legged Systems lab of IIT. His research interests include software for robotics, software architectures, kinematics and dynamics of mechanisms.

Jonas Buchli is a SNF Assistant Professor for Agile and Dexterous Robotics at ETH Zurich since May 2012. He holds a Diploma in Electrical Engineering from ETH Zurich (2003) and a Ph. D. from EPF Lausanne (2007). From 2007 to 2010 he was a Post-Doc at the CLMC Lab at the University of Southern California, where he led the USC Team for the DARPA Learning Locomotion challenge. In 2010-12 he was a Team Leader at the Advanced Robotics department of the Istituto Italiano di Tecnologia (IIT). He has received a Prospective and an Advanced Researcher Fellowship from the Swiss National Science Foundation (SNF). In 2012 he received a SNF Professorship Award. His research interests include model based control of legged robots and human locomotion and manipulation, machine learning and adaptive control, and dynamic and versatile service and field robots.

Darwin G. Caldwell received the B.S. and Ph.D. degrees in robotics from University of Hull, Hull, U.K., in 1986 and 1990, respectively, and the M.Sc. degree in management from University of Salford, Salford, U.K., in 1996. He is the Director of Advanced Robotics at the Istituto Italiano di Tecnologia, Genoa, Italy. He is a Visiting/Honorary/Emeritus Professor with University of Sheffield, the University of Manchester, and University of Wales, Bangor. His research interests include innovative actuators and sensors, haptic feedback, force augmentation exoskeletons, dexterous manipulators, humanoid robotics, biomimetic systems, rehabilitation robotics, and telepresence and teleoperation procedures.

Claudio Semini received his M. Sc. degree in Information Technology and Electrical Engineering from ETH Zurich, Switzerland, in 2005. For his Master thesis he visited the Hirose Robotics laboratory at the Tokyo Institute of Technology. In 2005/2006 he was a robotics researcher at the Toshiba's Corporate R+D Center in Kawasaki, Japan. Subsequently, he joined the Advanced Robotics department of the Istituto Italiano di Tecnologia (IIT), Italy, from where he received his Ph. D. degree in Robotics in 2010. Claudio Semini is the lead designer of the hydraulic quadruped robot HyQ and currently the head of the Dynamic Legged Systems lab of the IIT. His research interests lie in the field of the design and control of versatile, highly-dynamic robots with legs and arms.