A Domain Specific Language for kinematic models and fast implementations of robot dynamics algorithms

Marco Frigerio, Jonas Buchli and Darwin G. Caldwell

***Abstract*— Rigid body dynamics algorithms play a crucial role in several components of a robot controller and simulations. Real time constraints in high frequency control loops and time requirements of specific applications demand these functions to be very efficient. Despite the availability of established algorithms, their efficient implementation for a specific robot still is a tedious and error-prone task. However, these components are simply necessary to get high performance controllers.**
To achieve efficient yet well maintainable implementations of dynamics algorithms we propose to use a domain specific language to describe the kinematics/dynamics model of a robot. Since the algorithms are parameterized on this model, executable code tailored for a specific robot can be generated, thanks to the facilities available for DSLs. This approach allows the users to deal only with the high level description of their robot and relieves them from problematic hand-crafted development; resources and efforts can then be focused on open research questions.
Preliminary results about the generation of efficient code for inverse dynamics will be presented as a proof of concept of this approach.

I. INTRODUCTION

According to the presentation of the joint research project BRICS, aiming at identifying best practices in the development of robotics systems, such development process often lacks of a rigorous structure and principles [4], even after decades of research in the field. A typical example is software development for robotics, where the lack of design and identification of effective abstractions lead to the development of code–driven systems as opposed to model–based ones. In this regard, in [24] the authors point out the gap between the experience available in robotics and the exploitation of such knowledge for a proper software development process.

For the robotics research community as well as for a widespread adoption of robotic technology it is central to have flexible yet reliable software: a typical academic research unit can not afford the same resources to develop reliable software as an airplane or a car manufacturer, yet requires dependable and flexible software for similar complex machinery, in order to address open research questions.
Developing software for robots is among the most demanding and complex software engineering challenges due to a list of strict and partially conflicting requirements and the sheer complexity arising from the many tasks such a software has to perform in a well orchestrated manner.

Marco Frigerio, Jonas Buchli and Darwin G. Caldwell are with the Department of Advanced Robotics, Istituto Italiano di Tecnologia, via Morego, 30, 16163 Genova. {marco.frigerio, jonas.buchli, darwin.caldwell} @iit.it

More specifically, typical requirements for robot controller software are:

- Real time capability: specific sections of the program must be able to run in a hard real time context (e.g. a 1KHz force control loop).
- Safety guarantees: a high level of robustness of the whole system is desired (e.g. if dealing with a potentially dangerous robot, for people or for itself).
- Generation and deployment of components for multiple targets (e.g. different programming languages or different hardware platforms).
- Integration of many different resources (sensors, motors, processors etc), with different physical interfaces and APIs.
- Varieties of time constants and resource requirements: robotics applications must integrate components that need to run at different frequencies, with diverse usage of computation and memory resources (e.g. the sampling of a fast analog sensor and a stereo camera).

Satisfying such requirements translates in many software engineering challenges e.g. concerning also the architectural level of design [13]:

- Domain models: finding appropriate abstractions for very common components and *recurring problems* in robotics, to establish best practices and principled, general solutions (e.g. a reference C implementation of a PD controller or a general model of virtual components for operational space control [20]).
- Clear separation between control logic and task logic: it is desirable to be able to run exactly the same task code both against a simulator and on the real robot.
- Flexible yet resource efficient and real time capable memory management.
- Automatic unit testing.
- Tools to assess memory and time complexity.
- Integration of controller code: strategies to include components designed with different tools, such as MATLAB and SIMULINK [16].
- Automatic generation of infrastructure code: e.g. common components of simulators, coordinate transform matrices. It is desirable to avoid error-prone development by hand if this can be automated according to established models.
- Logging/debugging facilities: proper diagnostic tools that also satisfy the other requirements (e.g. a real time compatible logger).
- Graphical interfaces: visualization of the robot, the state

of its controllers, the layout of reference frames and so on; this dramatically reduces the effort for debugging.

A. Contribution and motivation

In this paper we address the automatic code generation of rigid-body dynamics algorithms for simulation and control of a robot under real time constraints, based on a general domain model. We will focus on robots assembled in chains or branched chains of rigid links.
Code used for model based controllers is a typical example of software with an apparent trade-off between flexibility/maintainability and efficiency. On one hand a rigid body model is a generic description of a robot that naturally lends itself for a rather general implementation, e.g. with object oriented code. On the other hand it is critical that such code does not violate real time constraints (e.g. by system calls such as those for dynamic memory allocation or file access) and is ideally running as fast as possible (e.g. exhaustive evaluations in sampling based planner algorithms or fast control loops).

To address and resolve this apparent trade-off, we propose a simple meta–model for the generalization of kinematics/dynamics models of robots, a Domain Specific Language (DSL) for specifying conveniently such models, and a transformation step built around the DSL for the generation of optimized rigid body dynamics algorithms.
The basic idea comes simply from the observation that dynamics algorithms are general and parametrized on the kinematic description of a robot – often called the robot *model* [9] – which is relatively compact and based on a common schema, but fully specifies the physics of the system. Thus it is sensible to look for a high level representation of the robot models, which can be easily constructed by hand, while exploiting automated procedures to turn such information into executable procedures tailored for the specific robots. As a an example, we will address a real time capable C++ implementation of the recursive Newton-Euler algorithm.

This approach is not new and with some variations it is adopted by simulators and software packages commercially available. For instance, SD/FAST [28] is a complex and rich simulator of mechanical systems that produces C or Fortran implementation of the equations of motions for the given system. Similarly, Robotran [3] targets multi-body dynamics applications; after reading a user model defined with a graphical editor, it can output symbolic equations of motion and perform simulations interacting with MATLAB.
SL is a rigid body dynamics simulator and robot controller package – we are currently using it for our research –, which uses its own description of kinematics and can produce highly optimized C code [23]. The performance of such code is definitely high, but quite some improvement can be introduced with regard to the flexibility and usability of the generation process.

Our aim is to collect existing strategies and provide a coherent approach based on a sound model and dependent solely on open source technologies. We aim at tools that possibly target different robotics platforms, which can be easily adopted by users in this field and help improving the quality of their work.
As a matter of fact, even though the modeling of kinematic trees and the related algorithms for kinematics and dynamics are well known in the robotics community and have been extensively studied in the past decades, they still represent an obstacle for the development of new robotics applications: a lot of initial development that targets such issues is required to make any robot operational, it is critical for the control and simulation but it is often not the focus of the research per se (for example if one wants to test his own learning algorithms on a new manipulator). Thus, researchers would benefit from an automatic implementation in terms of robustness of the code and time spent during the start-up of the project.

A general dynamics library (such as ODE [30]) – which would necessarily require the robot model as a *parameter* – could solve this problem. But the point of being able to generate a specific *instance* of the algorithm is efficiency without loosing flexibility. With our approach, one can target different platforms and apply custom optimizations. This aspect is further discussed in Section II.

B. Domain Specific Languages

A Domain Specific Language (DSL) is a formal language suitable to represent some sort of specification related to a precise class of problems only [19]. The syntax and the semantic of the language are explicitly designed to have a limited expressiveness in general, which is paid off by the ease with which the elements of the target domain can be represented.
Moreover, a DSL itself can be implemented – by specifying the custom syntax, providing a parser and further required facilities – with a technology independent from the final platform, i.e. the target for eventual executable artifacts. So for instance one can have multiple code generators that take an instance document of the DSL (e.g. a plain text file) and output code in different languages.

Although quite general, this description already suggests how the DSL technology nicely fits our problem and requirements, and therefore it is sensible to adopt it for our purposes: first, robot models can be described by easy-to-read text files which follow a custom syntax tailored to the specific domain. In the Robot Operating System ROS [21], for instance, model files need to be provided as XML which is harder to read and maintain; in the OpenHRP simulator [2], the language for the models comes from the 3D modeling field, and mixes graphical aspects and sensors with kinematics parameters.
Then, these documents can be parsed, checked and subject to custom transformations like generating code. The DSL allows to nicely decouple the simple model that needs to be built by the user, and the coding, which is more complex and can be partially automated.

The rest of the paper is organized as follows: Section II describes a code generator which exploits a priori knowledge of the robot; Section III presents the general structure of kinematic models while Section IV describes a DSL to provide robot specific descriptions. Finally, section VI presents some related work and section VII discusses improvements and future developments.

II. EFFICIENT CODE GENERATION

Rigid body dynamics algorithms can be used in a number of components of the software system of a robot: model based control (e.g. impedance control, inverse dynamics), simulation (e.g. physics based simulation), planning (e.g. kino-dynamic planning). In some applications (e.g. simulation) minimum time of execution is desired, while in other applications (real time planning and control) a certain maximum time of execution of the code is a strict requirement. On the other hand, manual coding of these routines is a non-trivial and error-prone task, and the demand for optimizing the execution time only makes the task harder. Therefore, leaving it to a computer and concentrate on higher level aspects of the research question, whenever possible, is an effective approach.

Other arguments for efficient implementations include the persistence in robotics of constraints due to space or power availability. Often one must adopt embedded computers, less powerful than regular desktop machines.
A user might also be simply interested in having *full* control on the software of the robot, and would therefore appreciate to develop (once for all) his own code generator without external dependencies. This requirement might arise when dealing with low level, hard real time code for a machine that requires strict control to guarantee safe operation (such as the hydraulic quadruped robot we are developing at our lab, HyQ [26]).

In this paper we focus on the Newton-Euler inverse dynamics algorithm as the reference example (see [9], [10], [11] for detailed explanation of the algorithm). The purpose of inverse dynamics is computing the following function for multi body systems:

$$\boldsymbol{\tau} = f(\ddot{\boldsymbol{q}},\ \boldsymbol{q}, \dot{\boldsymbol{q}}) \tag{1}$$

where $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ are the *actual* joint position and velocity vectors, $\ddot{\boldsymbol{q}}$ is the *desired* joint acceleration vector, and $\boldsymbol{\tau}$ contains the forces required to achieve such accelerations.
As pointed out in [9], an additional, implicit input is the system model for which forces have to be computed. Exploiting prior knowledge about the structure and the parameters of the robot, we can resolve that dependency but also generate optimized code, by avoiding any logic that deals with a generic case (e.g. loops) and especially by exploiting numerical properties (e.g. avoiding multiplication with zero to simplify matrix operations). For instance, in the assumption of having only plain prismatic or revolute joints, the matrix S describing the motion subspace of a joint is a single column vector with only one non-zero element, thus operations involving this matrix can be greatly simplified.

Another well known advantage of code generation based on a DSL is the possibility to target different languages and platforms. For instance, even if the purpose of MATLAB is certainly not to achieve top speed, one would still benefit from optimized (if automatically generated) algorithms for simulations and rapid prototyping of algorithms.

III. MODELING KINEMATIC TREES

A. Introduction

In this paper we deal with robot *models* – descriptions implying a certain degree of abstraction – related to kinematics and dynamics. The main assumption underlying these models is that all the bodies comprising the system are perfectly rigid. From the dynamics point of view (i.e. rigid body dynamics), the basic model also assumes idealized sources of *generalized* force (i.e. force/torque) that move the bodies; the information required to compute the effect of forces is given by the inertia parameters of the bodies, i.e. mass, position of the center of mass, inertia matrix.
Concerning kinematics, we shall give here a brief description of the structure of the models and the amount of information they embed, to provide the background for the rest of the paper. For an extensive and authoritative treatise on these topics, see [29], [9].

In kinematic models, a robot is an assembly of links and joints: a link is a rigid body with inertia properties while a joint represent a *constraint* between exactly two bodies (the predecessor and successor), which would otherwise be fully free to move relatively to each other. Such a constraint is not purely a rigid junction since the joint guarantees certain *degrees of freedom* (DoF) to the attached link. A specification of the nature of each joint is obviously required.
The description of the whole structure of a robot is topological, that is, it can be simply represented by a graph where joints are arcs and bodies are nodes (quite the contrary of what graphical intuition might suggest). For simplicity, we will focus only on kinematic trees (i.e. no loops in the structure), which represent a wide class of the robots used in industry and research; the full generalization of the model is one of the natural topics for future development, and can be done by integrating the methods described in [9] in our DSL framework.

Reference frames: The geometry of the bodies and their connections is required to dynamically compute the *pose* of the bodies, the dynamical effects of the movements, such as Coriolis and centrifugal forces, and so on. To this end, various reference frames must be placed in known points of every body and every joint of the tree. The parameters for a set of *transformations* among different frames *plus* a convention about the placement of them (e.g. the z axis of a joint reference is always aligned with the rotation axis) basically encode all the required information.

Figure 1 shows the layout of frames in a general case. For more information about the convention please refer to [9] and [11]; in the following we state only some observations relevant for the development of our DSL.
We emphasize that the transform ${}_JX_p$ for the joint frame is a *constant*, since it describes the placement of the joint expressed in the reference of the predecessor link (i.e. ${}_JX_p$ depends on static, geometrical parameters of the robot).
Furthermore, we note that for each joint there are two frames,

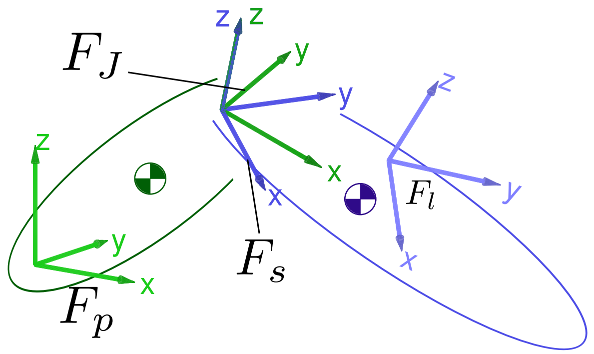

Fig. 1. Layout of reference frames for a generic section of a kinematic chain. F_p and F_s are the frames respectively of the predecessor and successor link of joint J, whose frame is F_J. F_p and F_J do not move with respect to each other, while F_J and F_s do, according to the joint behavior. F_l shows a possible additional frame located on the link.

which coincide when the joint status (i.e. the actual angle or displacement) is zero. Only the second frame moves as the joint moves, since it is attached with the successor link. As in [9] this frame (F_s) is chosen to be the reference for this link. Among the other things, this implies:

- no transformation parameters have to be associated with the link, since its frame is completely determined by the convention and the joint status.
- The generic transform ${}_sX_J$ between the two frames on the joint (F_J and F_s) is the only one which depends on the joint status. Note that ${}_sX_J$ captures the *type* of the joint as well (i.e. rotational or prismatic).

Even if adopting this convention, for further flexibility another frame might be added anywhere on the link, according to any user preference or requirement (e.g. to express more conveniently the position of certain sensors placed on the link).

B. UML *model*

Figure 2 shows an UML class diagram representing the key elements described before. The diagram is simple but general, and can be applied to almost any robot made by rigid links.
The central classes, quite intuitively, are `Link` and `Joint`. Joints induce a parent–child relationship among links, which is characterized by the type of joint. We chose to model this relationship by making `Joint` an association class connected to the self–association for `Link`. To keep the model simple we consider only 1-DoF joints, since actual composite joints (as a three DoFs ball joint) can be represented by primitive ones connected by virtual dimensionless links (irrelevant for kinematics and dynamics computations – see below).
The association between `Joint` and the class `DoF` basically models the intrinsic property that tells which relative movement is allowed by the joint. Links, on the other hand, have certain degrees of freedom as a consequence of the kinematic configuration, and they have a similar association as well.

Any link can have multiple children, which corresponds to a branched structure of the robot. On the other hand, as mentioned before we do not consider loops, which would require another type of joint (i.e. a loop joint) which does not determine any new child link but rather connects two existing links. The abstract class `Link` actually models any rigid body, and has a few subclasses to differentiate particular cases:

- `ChainLink`: a generic piece of the kinematic chain, what is usually referred to as link;
- `RobotBase`: a special link which represents the “root” of the kinematic tree. Can be floating if the robot is a mobile one. Note the stereotype `Singleton`, since there is only one base for each robot;
- `VirtualLink`: a dimensionless body to allow the representation of complex joints (see above); this class explicitly forces the inertia parameters of its instances to be zero. Floating base robots can be thought as connected via a virtual six-DoFs joint (i.e. no constraint), to an arbitrary point in the world, which is a virtual link as well: `WorldBase` (cf. [9]).

Finally, the conceptual model of Figure 2 describes reference frames through the class `RefFrame` and the associations with `Link` and `Joint` (see section III); however, since a frame per se does not really have any property (we assume only right–handed coordinate systems) or behavior, we observe that the relevant information is instead in `Transform`, which provides the transformation parameters for a given couple of frames.

IV. THE DSL

DSLs can be roughly divided into two categories, internal and external, the former being built through a particular usage of an existing language, while the latter is independent and usually has a custom syntax [12]. We shall choose an external DSL, whose model documents can be plain text files, with a clear aspect (syntax) and intuitive semantics.

As argued in [12], a proper DSL design would not be complete without an underlying domain model, for which DSL documents are just a specification of its instances[1]. Our domain model is described in Section III and its instances are specific robot models (so we can refer to the former as the *meta*–model); each DSL document has to carry the information to populate one of such models i.e. telling the number of links/joints, their type, their attributes, and so on. The grammar, which must specify the structure of such documents, is naturally inspired by the meta–model, which defines the structure of the *information* carried in the documents. Therefore, after the model had been established reasonably, the design of the grammar was quite straightforward. The required effort was limited and subject to a confident understanding of the domain.

Obviously the grammar of the DSL also provides additional syntax elements to improve readability. See Figure 3 for an excerpt.

[1] Actually Fowler uses the term “semantic model”, to mean a part of the whole domain model, and identifies each DSL document with *a* semantic model, rather than talking about instances.

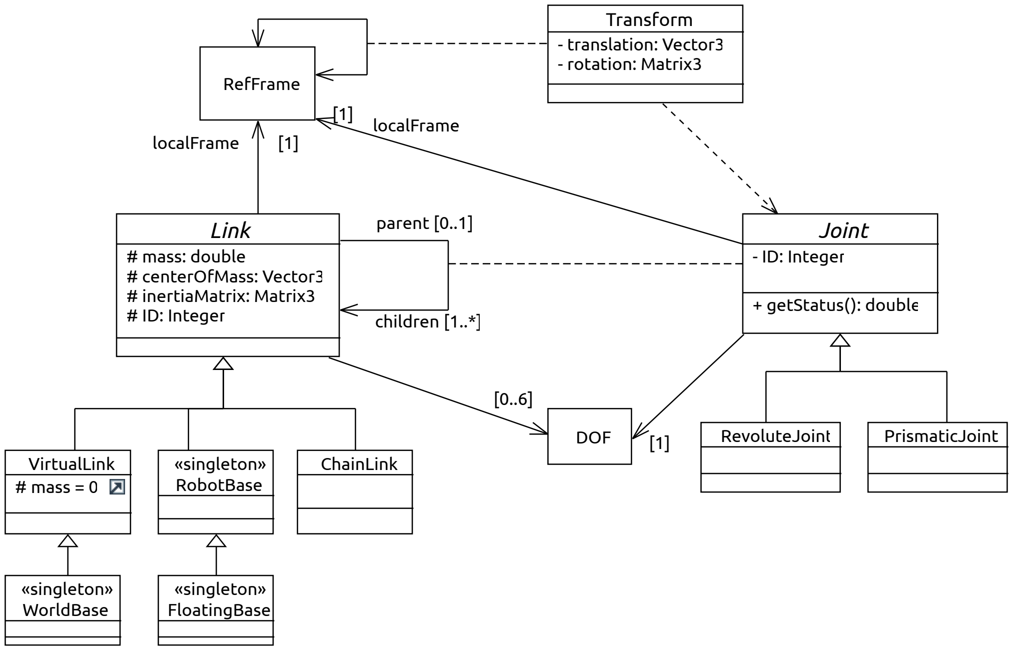

Fig. 2. The kinematic tree (meta)model as an UML class diagram.

```
generate kinDsl "dsl.iit/KinDsl"

Robot:    'Robot' name = ID '{'
           base = RobotBase
           links  += Link+
           joints += Joint+
          '}';

AbstractLink:  Link | RobotBase;
RobotBase:     FixedRobotBase | FloatingRobotBase;

FixedRobotBase:    'RobotBase' name=ID '{'
                      inertiaParams = InertiaParams
                      childrenList = ChildrenList
                      refFrame = RefFrame
                   '}';
FloatingRobotBase: 'RobotBase' name=ID 'floating' '{'
                      inertiaParams = InertiaParams
                      childrenList = ChildrenList
                    '}';
Link:    'link' name=ID '{'
           'id' '=' num = MY_ID
           inertiaParams = InertiaParams
           childrenList = ChildrenList
           refFrame = RefFrame
         '}';
```

Fig. 3. An excerpt of the DSL grammar designed with Xtext. `InertiaParams` defines how to write in the document the inertia parameters of the bodies. `ChildrenList` allows to insert a list of links, while `RefFrame` takes care of the rotation and translation parameters of a coordinate transform. See also Figure 4.

A. Tools

The cost for the syntactic freedom associated with an external DSL is the need to develop a custom grammar and the associated parser, but luckily there are effective tools to support these activities. We have adopted Xtext, a framework based on the Eclipse platform that supports the creation of a complete language infrastructure [7]; both software packages are open source tools. In particular, Eclipse is a rich development environment widely used in different domains, equipped with large support for model–driven development [14] and adopted in the robotics community as well [4]. Xpand/Xtend are the related languages to specify the templates for text generation.

However, it is important to note that Xtext/Eclipse can output a stand-alone package containing the main tools related to the DSL (i.e. the parser and the code generator), which can then be distributed and used independently of it. The only requirement is a Java interpreter, which is a widely adopted technology. Eclipse/Xtext provides also rich text editing features to write DSL documents, but any plain text editor can be used.

B. Example: a quadruped robot

To provide further insight on the structure of our DSL, Figure 4 shows a section of the description of our mobile, four legged robot HyQ [26] as an example. HyQ has a trunk (the floating base) and four identical legs – left front, right front, left hind and right hind – each composed of three links: hip, upper leg and lower leg. As you can see from the listings, the floating base does not specify any reference frame transform, since there are no constraints between the world and the body and all the parameters of such transform are free.

V. EXPERIMENTS AND RESULTS

Once the DSL is completed, creating new robot descriptions is a matter of minutes, since the DSL is simple and intuitive (most of the time is spent looking in the robot documentation for the inertia parameters and the frame transformations). If the code generator is properly verified, then it is impossible to introduce low level bugs such as memory leaks in this step.

```
Robot HyQ {
RobotBase trunk floating {
    inertia_params {
        mass =  53.0
        CoM = (-0.0002, 0.0001, 0.0011)
        moments =  1.26,6.56,6.69,0.03,-0.06,-0.03
    }
    children {
        LF_hip via LF_jHAA
        RF_hip via RF_jHAA
        LH_hip via LH_jHAA
        RH_hip via RH_jHAA
    }
}

link LF_hip {
    id = 1
    inertia_params {...}
    children {  LF_leg via jHFE }
    ref_frame {...}
}
link LF_leg {
    id = 2
    inertia_params {...}
    children {  LF_lowerleg via jKFE }
    ref_frame { ...}
}
link LF_lowerleg {
    id = 3
    inertia_params {...}
    children {}
    ref_frame {...}
}

r_joint jHAA {
    id = 1
    ref_frame {
        translation = (0.0, 0.0, 0.0)
        rotation = (0.0, -1.57079632, -3.141592)
    }
}
r_joint jHFE {
    id = 2
    ref_frame {
        translation = (0.08, 0.0, 0.0)
        rotation = (1.57079632, 0.0, 0.0)
    }
}
r_joint jKFE {
    id = 3
    ref_frame {
        translation = (0.35, 0.0, 0.0)
        rotation = (0.0, 0.0, 0.0)
    }
}
}
```

Fig. 4. An excerpt of the DSL instance document modeling the quadruped robot HyQ. It shows the trunk and the parts of one leg; the other legs are almost identical. LF stands for left-front, RH for right-hind, and so on; jHAA is joint-Hip-Abduction-Adduction, jHFE is for Hip-Flexion-Extension, jKFE for Knee-Flexion-Extension. Inertia parameters and reference frames for the leg links are hidden to keep the image small.

For a proof of concept of the proposed approach we chose the C++ language and the Eigen library for linear algebra [15]; this allowed to have more compact and readable code, so that it is easier to debug during the first experiments. Eigen is a modern, carefully designed and quite well documented library for efficient computations with matrices, adopted for instance in ROS [21].

However, whether using an external library is appropriate given the discussed requirements, is something we have to establish with experimentation: on one hand these libraries provide very efficient optimization (e.g. avoiding temporaries and exploiting sparsity), and also allow to have clearer code.

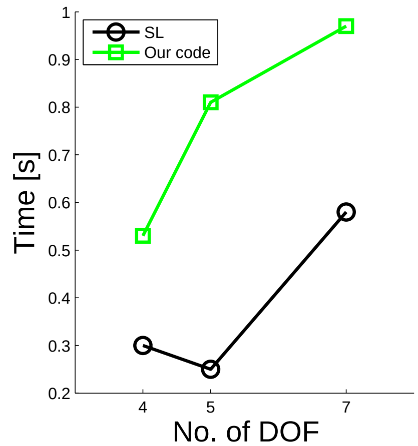

Fig. 5. Performance comparison of two implementations of the Newton–Euler algorithm for inverse dynamics. The plot shows the cumulative execution time for 10^5 calls of the function $\boldsymbol{\tau} = f(\ddot{\boldsymbol{q}},\ \boldsymbol{q}, \dot{\boldsymbol{q}})$ as a function of the number of degrees of freedom of the model (tests executed on a Intel(R) Core(TM)2 Duo CPU, P8700 @ 2.53GHz).

The other solution (mandatory if similar facilities are not available but speed is still of concern) is to manually address each single operation of the algorithm producing low level, basic instructions only when necessary; this is the most inconvenient approach, results in not so clear templates and code but it is the most efficient. In addition, adopting a library injects an external dependency and might not be trivial to use it properly (in our opinion, an effective usage of Eigen requires some experience, due for instance to the complexity of expression templates).

For numerical correctness, we have tested our implementation of the inverse dynamics algorithm against the MATLAB code available on Featherstone's web page [8], comparing the numerical output for different robot models and different inputs ($\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$).
As far as performance is concerned, instead, we made some comparisons with the SL simulator; as mentioned in Section I, this software generates a highly optimized, low level C code implementation, whose performance can very well be considered as a reference. SL adopts exactly the inconvenient-but-fast approach described before.

The graph in Figure 5 shows how the algorithms scale as a function of the number of DoFs, and shows at the same time a speed comparison. We used a four-DoF robot (a leg of HyQ attached to a vertical slider), a five-DoF robot with revolute and prismatic joints and finally a seven-DoF model obtained adding a two link branch to the previous robot. As expected, SL provides the fastest function, but our implementation based on Eigen is not that much slower. For some reasons, probably related to the different optimization applicable to different models, SL is slightly faster with five DoFs compared to four DoFs.
It is important to note that we did not apply so much hand-crafted optimization, leaving this job to the library and the compiler, but we got already good results. Our generator

basically unrolls loops, uses a sparse vector for the motion subspace matrix, and then quite literally maps the steps of the algorithm into the appropriate algebra operations. Thus there is room for further optimization, like precomputing some operations (e.g. algebra which involves constants such as ${}_J X_p$). Thanks to the overloaded operators and clear identifiers built from the names provided in the DSL, the resulting code is human-readable, unlike the code generated by SL.

Even though this code does not have to be directly maintained – as opposed to what is behind the generator, i.e. the model and the template – having it readable is quite desirable. The user can more easily inspect it, and spot errors in the generation process.

All the code has been compiled with the same flags, and functions have been statically linked into the executable. The program measures CPU time by calling the library function `std::clock()`.

VI. RELATED WORK

Software engineering *for* robotics has only recently become an explicit research area (especially if considering the age of the two disciplines), as shown for example by the birth of a new journal [6].

In this context, the model–driven paradigm is recognized to be an effective approach for the design of software. In [31], the authors point out the importance of resource awareness in robotics applications; they describe a development process and a meta–model for robotics systems that are focused on the non-functional properties of the components.

The techniques of meta–modeling and domain specific languages are exploited in [22] to design a programming environment independent of the target robot, to facilitate the specification and reuse of control programs.

In [18], the authors present an execution environment based on the scripting language Lua, to support the implementation of internal DSLs for modeling expressive state machines for robot coordination. The work focuses particularly on dynamic memory management issues, not to violate real time constraints during the interpretation (execution) of the state machines.

An example of the use of a DSL in robotics, as a consequence of the need to find higher abstractions to drive software development, is presented in [5], which targets the specific field of modular robots. Here the authors give an extensive description of a domain specific language for modeling the kinematics of individual robot modules and their possible interconnections, which is exploited to generate code for both the Webots simulator and a custom platform for the execution of real experiments. In the same context, [25] presents a high level language built around the concepts of roles to facilitate the programming of controllers for the modular robot ATRON, independently of its physical configuration. While sharing the approach of model based generation and the focus on kinematics, our work targets the different domain of robots with linear or branched structure composed by rigid links (such as manipulators or legged machines); it focuses on the generation of efficient dynamics algorithms applicable in different components of a software framework for robots.

SIMULINK [16] is a well known tool in engineering which supports the simulation of a broad class of dynamical systems, and can also generate MATLAB or C code. However, SIMULINK is very general and thus not so convenient for very specific needs like customized code generation of particular algorithms as the ones by Featherstone [9].

Similar comments apply for instance for Modelica, a multi-domain, object-oriented modelling language used also in industry [1]. Its models basically contain the system equations, which then need to be transformed into executable code or into a form suitable for a simulation engine.

Being so general purpose, these tools are likely to incur some unnecessary overhead in terms of learning, usage and required tools, if one wants to get similar results as with the DSL; the DSL infrastructure is more lightweight and designed explicitly with the requirements of a real time controller code for a real robot in mind.

VII. CONCLUSIONS AND FUTURE WORKS

In this paper we have proposed a Domain Specific Language for the specification of kinematics and dynamics parameters of robots consisting of rigid links. The DSL is based on a domain model that captures the minimum amount of information required to specify the physics of the system. By using this information it is possible to generate executable code as for instance rigid body dynamics algorithms; such code is efficient and compatible with real time constraints at high frequencies, e.g. in low level control loops. This approach allows researchers to quickly set up new simulations or controllers, without having to deal manually with critical and delicate parts of code.

This work aims at contributing to the field of model–based development for robotics; on one side robotics research requires a lot of experimental and exploratory activities and on the other side exhibits many recurring issues and common problems that should be solved by principled, general approaches. Our work aims at addressing these recurring issues and thus freeing resources for the required exploratory sides of robotics research.

However we stress that our work is still at a preliminary stage, and many aspects could be improved. A natural development is to investigate other targets for the code generation, addressing for instance algorithms for floating base robots, and forward dynamics. As an additional example, much of the infrastructure for more advanced control schemes, e.g. operational space control [17], [27] could be generated. This includes transformation, Jacobian and projection matrices for specific points on the kinematic tree.

Other improvements of the DSL itself include extending the validation of documents with checks of semantic constraints (e.g. a link cannot be the child of more than one other link) or the usage in the documents of labels defined externally.

The model described in section III-B has been developed mainly as a reference for the design of the DSL, and could be refined and extended as well. A minor improvement would be including data about the joints range of motion, which is not relevant for dynamics algorithms but it is definitely part of a kinematic description. Kinematic loops should be addressed explicitly; additional classes like `Chain` and `Tree` might be added.

In the paper we have already referred to the class diagram of Figure 2 as a meta–model. As a final remark, we observe that it could equivalently be considered as a simple domain model, that is, a description which drives the design of software representations of the important elements for a problem. Joints and links (or legs, for example) are the subject of a variety of tasks which involve different components of the robot software, as for instance the low level position control or the planning of foot trajectory in a humanoid. Therefore, finding proper representations in computer code of these objects – and of all the other relevant aspects – is itself an important issue in the software for robotics.

VIII. ACKNOWLEDGMENTS

This research has been funded by the Fondazione Istituto Italiano di Tecnologia.

We would like to thank Frank Buchli at Zuehlke Engineering for helpful discussions about the use and implementation of domain specific languages.

REFERENCES

[1] Modelica and the Modelica Association [online]. Available from: `https://www.modelica.org` [accessed on 2011.07.25].

[2] OpenHRP [online]. Available from: `http://www.openrtp.jp/openhrp3/en/index.html` [accessed on 2011.07.25].

[3] Robotran [online]. Available from: `http://www.robotran.be/` [accessed on 2011.07.25].

[4] Rainer Bischoff, Tim Guhl, Erwin Prassler, Walter Nowak, Gerhard Kraetzschmar, Herman Bruyninckx, Peter Soetens, Martin Haegele, Andreas Pott, Peter Breedveld, Jan Broenink, Davide Brugali, and Nicola Tomatis. BRICS – best practice in robotics. In *IFR International Symposium on Robotics (ISR)*, 2010.

[5] Mirko Bordignon, Ulrik Pagh Schultz, and Kasper Stoy. Model-based kinematics generation for modular mechatronic toolkits. *ACM SIGPLAN Notices*, 46:157–166, October 2010.

[6] Davide Brugali. From the Editor-in-Chief: A New Research Community, a New Journal. *Software Engineering for Robotics*, 1:1–2, 2010.

[7] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.

[8] Roy Featherstone. Spatial Vectors and Rigid-Body Dynamics [online]. Available from: `http://users.cecs.anu.edu.au/~roy/spatial/index.html` [accessed on 2011.07.25].

[9] Roy Featherstone. *Rigid Body Dynamics Algorithms*. Springer, 2008.

[10] Roy Featherstone. A Beginner's Guide to 6-D Vectors (Part 1). *IEEE Robotics & Automation Magazine*, 17(3):83–94, 2010.

[11] Roy Featherstone. A Beginner's Guide to 6-D Vectors (Part 2). *IEEE Robotics & Automation Magazine*, 17(4):88–99, 2010.

[12] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[13] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–39. World Scientific Publishing Company, 1993.

[14] Richard Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Eclipse Series. Addison-Wesley Professional, 2009.

[15] Gaël Guennebaud, Benoît Jacob, et al. The Eigen library v3 [online]. Available from: `http://eigen.tuxfamily.org`.

[16] The MathWorks Inc. Matlab & Simulink software [online]. 2009. Available from: `http://www.mathworks.com/` [accessed on 2009.09.20].

[17] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[18] Markus Klotzbcher, Peter Soetens, and Herman Bruyninckx. OROCOS RTT-Lua: an execution environment for building real-time robotic domain specific languages. In *Internation Workshop on Dynamic Languages (DYROS)*, November 2010.

[19] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[20] Jerry Pratt, Chee-Meng Chew, Ann Torres, Peter Dilworth, and Gill Pratt. Virtual Model Control: An Intuitive Approach for Bipedal Locomotion. *The International Journal of Robotics Research*, 20(2):129–143, 2001.

[21] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[22] Michael Reckhaus, Nico Hochgeschwender, Paul G. Ploeger, and Gerhard K. Kraetzschmar. A platform-independent programming environment for robot control. In *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, 2010.

[23] Stefan Schaal. The SL simulation and real-time control software package. Technical report, CLMC lab, University of Southern California, 2006. Available from: `http://www-clmc.usc.edu/publications/S/schaal-TRSL.pdf`.

[24] Christian Schlegel, Thomas Haler, Alex Lotz, and Andreas Steck. Robotic software systems: From code-driven to model-driven designs. In *International Conference on Advanced Robotics (ICAR)*, pages 1–8, july 2009.

[25] Ulrik P. Schultz, David Christensen, and Kasper Sty. A domain-specific language for programming self-reconfigurable robots. In *Automatic Program Generation for Embedded Systems (APGES)*, 2007.

[26] Claudio Semini, Nikos G. Tsagarakis, Emanuele Guglielmino, Michele Focchi, Ferdinando Cannella, and Darwin G. Caldwell. Design of HyQ – a Hydraulically and Electrically Actuated Quadruped Robot. *Proc. IMechE Part I: J. Systems and Control Engineering*, 225, 2011.

[27] Luis Sentis. *Synthesis and Control of Whole-Body Behaviors in Humanoid Systems*. PhD thesis, Stanford University, July 2007.

[28] Michael Sherman and Dan Rosenthal. SD/FAST [online]. Available from: `http://www.sdfast.com/`.

[29] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics. Modelling, Planning and Control*. Springer, 2009.

[30] Russell Smith. The Open Dynamics Engine simulation library [online]. Available from: `www.ode.org` [accessed on 2011.07.25].

[31] Andreas Steck and Christian Schlegel. Towards quality of service and resource aware robotic systems through model-driven software development. In *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, September 2010.