A Versatile C++ Toolbox for Model Based, Real Time Control Systems of Robotic Manipulators

Robert Höpler and Martin Otter
Institut für Robotik und Mechatronik
DLR Oberpfaffenhofen
D-82234 Wessling, Germany
email: Robert.Hoepler@dlr.de, Martin.Otter@dlr.de

Abstract

Model based technologies form the core of advanced robotic applications such as model predictive control and feedback linearization. More sophisticated models result in higher quality but the use in embedded real-time control systems imposes strict requirements on timing, memory allocation, and robustness. To satisfy these constraints, the model implementation is often optimized by manual coding, an unwieldy and error prone process. This paper presents an approach that exploits code synthesis from high level intuitive and convenient multi-body system (MBS) model descriptions. It relies on an object-oriented C++ library of MBS components tailored to the computations required in robot control such as forward and inverse kinematics, inverse dynamics, and Jacobians. Efficient model evaluation algorithms are developed that apply to multi-body tree structures as well as kinematic loops that are solved analytically for a certain class of loop structures.

1 Introduction

Modern, embedded control systems of robotic manipulators internally use robot models to address several tasks, mainly motion planning, trajectory generation, loop control, parameter identifcation. Trajectory planning algorithms rely on the kinematic model of a robot and its environment to prescribe the geometrical path of the motion [1]. To compute the actuator power according to optimal control laws, that are applied to achieve, e.g., time-optimal movement [2], the model captures the relevant dynamic behaviour of the manipulator. The demand for high precision in industrial applications or the use of autonomous robots result in the need to periodically update and develop the internal model for best performance. In order to achieve a model as realistic as possible, its parameters must be identified from measurements in the real system and corrected [3].

Tasks like feedback linearization often have to be performed on-line. This requires time-consuming model evaluations to be done at the sampling-rate of the control-loop. Efficient algorithms for the computation of the non-linear dynamics of chain-structured robotic manipulators are well-known, e.g. [4, 5]. To optimize performance in embedded systems such as robot control, implementation is prevalently done by manual coding. However, this is error-prone, time-consuming, and tends to result in fast, but inflexible code.

In certain domains such as robotics research or manufacturing robots, manual coding conflicts with the necessity for flexibility and generality. Configurations, that are complex from a modeling perspective, such as kinematic loops, are used to improve performance and stability. Rapid changes in the robot manufacturer's product palette, multi-tooling robots, and operation in different environments, require quick model adaption to the different applications and configurations. Robot modeling, control algorithms, and the code synthesis process are part of this demanding process that mandates flexibility, while facilitating efficient embedded code to satisfy the harsh constraints imposed by the real-time environment.

Several approaches exist to automatically synthesize embedded code for a dynamics model of a robot. One way is to create the MBS equations of motion symbolically from a model description [6, 7]. These are converted to code and finally linked to the system. The resulting code is very efficient due to consideration of the algebraic structure of the equations. One drawback of this approach is the causal input-output behaviour of the generated models. This requires more than one model to be present at run-time, which conflicts with the limited resources available in embedded systems. In addition, when dynamic reconfiguration of the model occurs, recompilation is required which may take prohibitively long and exceed resources available. Alternatively, the model may be implemented in a high-level programming language. Implementations in an interpreter language such as MATLAB [8] are flexible, but lack performance and are difficult to integrate in an embedded system [9]. For portability reasons one is *de facto* restricted to Fortran, C, or C++.

The objective of this paper is to describe the design of a multi-purpose C++ class library for object-oriented robot modeling fitted for kinematics and dynamics computations. The main goal was to fulfil the demands for fast computations and robust evaluations needed in a real-time environment. Furthermore it is shown that a hierarchical class concept can help to reduce modeling efforts and enables non-domain engineers to synthesize code from high-level model descriptions without sacrificing performance. The resulting C++ class library, which is partially based on ideas from [10, 11, 12, 13], will be discussed in the following chapters. In Section 2 the composition of models is described, including a method to treat the dynamics of a class of kinematic loops. In Section 3 the different functions provided by the library are shown. Some examples demonstrating the high performance of this approach are presented in Section 4. Section 5 summarizes the results.

2 Model Composition

In the field of robotics a popular modeling approach is considering the manipulator as a chain of homogenuous transformations parametrized by Denavit-Hartenberg parameters [14]. This representation is not easy to apply when modeling more general structures, such as kinematic trees and loops, especially when using an object-oriented component library. For this reasons in this paper the mechanical structure of a robotic manipulator is approximated by a rigid multi-body-system (MBS) representation, i.e. a set of bodies, connected by joints and springs. This approach is described in detail in this section.

2.1 Components

A robot model is composed of components, which represent physical entities such as (corresponding class identifiers given in brackets)

- elementary joints (`RevoluteJoint`, `PrismaticJoint`, `ScrewJoint`, ...),
- rigid bodies (`Link`, `Body`, ...),
- springs of different characteristics (`Spring`, ...) and external forces (`ExtForce`, `ExtTorque`, ...),
- drives and gears (`Gear`, ...), and
- components to process kinematic loops (`ConnectingRod`).

All objects are descendants of class `Transmission` and are therefore called *transmission objects* [10]. If a transmission object contains other transmission objects, it is called an *assembly*, otherwise a *primitive*. The class declaration of a primitive consists of three main blocks: (i) connector attributes, (ii) physical properties, and (iii) transformations defining the kinematic and dynamic behaviour of the object. To illustrate, an example of a simplified C++ declaration of a revolute joint primitive is shown in Fig. 1.

```
class RevoluteJoint : public Transmission
  // Connectors
  ConnectorFrameA frame_a;
  ConnectorFrameB frame_b;
  ConnectorStateB axis;
  // Physical properties
  StateVariable q;     // joint angle+derivatives
  ParameterVector3D n; // axis of rotation vector
  // Set of transformations
  virtual void do_position(); // forw. kinematics
  virtual void do_velocity();
  virtual void do_acceleration();
  virtual void do_force();    // inverse dynamics
;
```

Figure 1: Simplified `RevoluteJoint` class declaration.

This declaration reflects the way of operation of an object as sketched in Fig. 2, i.e., calling a transformation member takes data provided by connectors, transforms the data according to local (physical) properties and supplies it to other connectors.

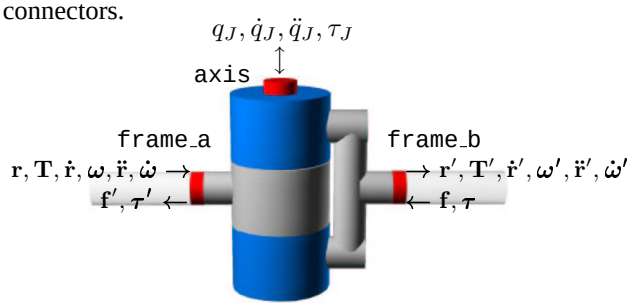

Figure 2: Symbolized way of operation of a transmission object. The `RevoluteJoint` object transforms the kinematical entities position vector $\mathbf{r}, \ldots, \dot{\boldsymbol{\omega}}$ from connector `frame_a` to `frame_b` and force and torque vectors $\mathbf{f}, \boldsymbol{\tau}$ the reverse direction. The joint variable q_J is available on the axis.

2.2 Connections

A port-based formulation is used well suited to MBS, i.e., a connection between two transmission objects can only be established between two ports, called *connector*. Such a connection has different levels of abstraction:

- From a MBS perspective it defines a unique position or frame where two entities of the MBS model interact. Presently, there are two modeled interactions, `Frame` and `State`, that correspond to different levels of detail. To allow the use of o(n) recursive Newton-Euler algorithm (RNE) [4] a tree-structured MBS is required. The formation of a spanning tree is achieved by supplying complementary ('A' and 'B' type) connectors.
- From a modeling perspective it is a binary relation between two complementary connectors implemented by the function `connect`, e.g. `connect(Revolute1.frame_b,Link2.frame_a)`.

- The implementation resolves all kinematic and dynamic data, e.g., force and torque vectors, in local coordinate frames defined in the connectors. Inside a transmission object these vectors are transformed between the connectors by executing the basic transformations. For example, calling a `RevolueJoint do_position()` member rotates the coordinate frame present in `frame_a` into `frame_b` about the rotational axis `n` according to `q`.

2.3 Example Robot

To illustrate, the model definition is given for a 4 degrees of freedom (dof) palletizing robot, as depicted in Fig. 3. The primary kinematic chain is a 5 axis manipulator, augmented by two coupled closed-chain mechanisms to keep the tool flange permanently parallel to the ground without actuation. Note, that two of the joints (`R4` and `R6`) are not actuated.

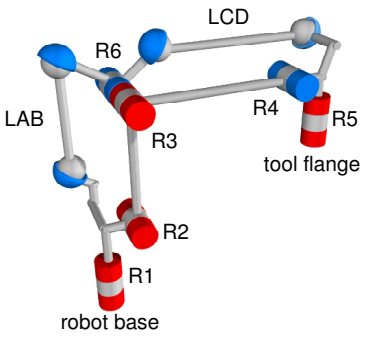

Figure 3: Model of a generic 4 dof palletizing robot with two coupled kinematic loops.

The corresponding model definition is shown in Fig. 4. In this realistic C++ code fragment components are defined and connected to form an MBS model conveniently. Arguments that furnish the objects with physical parameters such as mass, inertias, etc., have been omitted for brevity and are symbolized by '. . .'.

2.4 Automatic Processing and Solution of Kinematic Loops

Simple connection semantics are used to ensure that a model has a tree structure, which applies directly to the RNE algorithm [4]. A transmission object is restricted to have a single `A` connector and multiple `B` connectors, and one `B` may be connected to several `A` connectors. To facilitate specific loop structures a new type of object is introduced that has two `A` connectors, and therefore is able to tie two branches of a tree.

The solution of MBS that include kinematic loops is significantly more complicated than solving a tree-structured system [15]. In mathematical terms a closed loop imposes algebraic constraints on the equations of motion, which requires the solution of a non-linear system of equations, called *closure conditions*. Many techniques exist for their

```
int main() {
  // define components of primary 5 axis chain
  InertialSystem Base(Vector3D(0,0,-9.81));
  RevoluteJoint R1(ZAXIS);
  RevoluteJoint R2(YAXIS);
  RevoluteJoint R3(YAXIS);
  RevoluteJoint R4(YAXIS,DEPENDENT);
  RevoluteJoint R5(ZAXIS);
  Link L12(...);   Link L23(...);
  Link L34(...);   Link L45(...);
  // concatenate the primary kinematic chain
  connect(Base.frame_b,R1.frame_a);
  connect(R1.frame_b,L12.frame_a);
  ...  // more connects
  connect(L45.frame_b,R5.frame_a);
  // define components of kinematic loops
  RevoluteJoint R6(YAXIS,DEPENDENT);
  Link L1A(...);   Link L36(...);
  Link L6B(...);   Link L6C(...);
  Link L5D(...);
  ConnectingRod LAB(...);
  ConnectingRod LCD(...);
  // 'close' kinematic loops
  connect(R1.frame_b,L1A.frame_a);
  connect(L1A.frame_b,LAB.frame_a1);
  connect(R3.frame_b,L36.frame_a);
  connect(L36.frame_b,R6.frame_a);
  connect(R6.frame_b,LAB.frame_a2);
  ...  // more connects
  // define some load
  Body Payload(...);
  connect(R5.frame_b,Payload.frame_a);
  ... // to be continued
```

Figure 4: C++ code fragment describing an MBS model of the example robot shown in Fig. 3.

general solution, e.g., Newton-Raphson methods. However, these are all iterative and have a number of inherent drawbacks: (i) the supply of initial values, (ii) a varying number of iterations, (iii) no guarantee of convergence, and (iv) nonunique solutions. As a consequence, it is difficult to apply iterative methods in on-line tasks with fixed timing, especially in a manufacturing system, where robustness and safety are paramount.

These problems can be mitigated by enforcing a restriction to a special class of loops, which allows an *analytic* and hence a fast and reliable solution of closure conditions. This paper follows the idea of Möller [11] to close the loop by a special object, a *connecting rod*, which is a rigid link with a spherical joint on each end, as illustrated in Fig. 5.

In kinematic terms the connecting rod imposes one additional constraint, i.e., a constant distance between two coordinate frames, which in turn reduces the number of dof in the loop by one. Therefore, one joint variable is *not* a dof. When creating a joint object, the user has to provide this information via an extra argument `DEPENDENT`. The example robot contains two loops, which in turn requires two joints, `R4` and `R6`, see Fig. 4 to be not driven.

2.4.1 Solving Loop Kinematics and Dynamics

The algorithm used in this work is based on a general method described in [11, 10]. It allows the efficient and reliable solution of non-linear closure conditions of loops permitting an explicit solution. It is illustrated by a closed-chain mechanism, built from several rigid bodies, arbitrary driven joints, one free revolute joint J and one connecting rod CR, see Fig. 5.

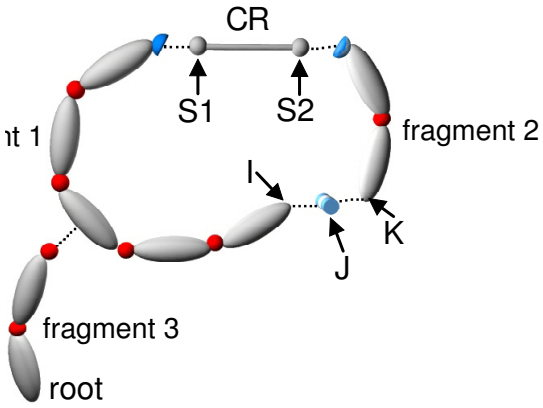

Figure 5: A kinematic loop partitioned for explicit solution. The generic loop consists of rigid bodies (ellipses), arbitrary joints (balls), one connecting rod (CR), and one free revolute joint (J).

In the first step of the *forward kinematics* the loop is automatically partitioned in three fragments, as depicted in Fig. 5. Next, the *relative* kinematics in $fragment1$ and $fragment2$ computes the vector pointing from frame I to $S1$ resolved in frame I $\mathbf{x} := {}^{I}\mathbf{r}^{I,S1}$ and $\mathbf{y} := {}^{K}\mathbf{r}^{K,S2}$. Considering the type of constraint imposed by the connecting rod, it can be solved explicitly for joint variable q_J when J is a revolute or a prismatic joint [11]:

$$g(q_J) := |T^{ki}\mathbf{x} - \mathbf{y}|^2 - |\mathbf{r}^{S1,S2}|^2 = 0\,, \qquad (1)$$

where $T^{ki}(q_J)$ is the transformation matrix from frame I to frame K. When $T^{ki}(q_J)$ is a rotation about the z-axis q_J can be determined by the solution of

$$A \cdot \cos q_{J_k} + B \cdot \sin q_{J_k} + C = 0\ , \qquad (2)$$

whereas

$$\begin{aligned} A &:= -2 \cdot (x_1 y_1 + x_2 y_2) \\ B &:= 2 \cdot (x_2 y_1 - x_1 y_2) \\ C &:= \mathbf{x}^T\mathbf{x} + \mathbf{y}^T\mathbf{y} - 2x_3 y_3 - \mathbf{r}^{S1,S2^T} \cdot \mathbf{r}^{S1,S2}\,. \end{aligned}$$

In case of a prismatic joint the expressions can be derived in an analogous manner. Note, that the solution is not unique. To choose one automatically, the initial configuration of the kinematic loop in the model is analyzed. After all joint variables are known, one can employ *global* kinematics to calculate the position of all parts of the MBS.

The algorithm avoids repeated evaluation of the kinematics in parts of the model as follows:

1. Do global kinematics in fragments 1 and 3 $\Rightarrow \mathbf{r}^{I,S1}$.
2. Compute *relative* kinematics in $fragment2 \Rightarrow \mathbf{r}^{K,S2}$.
3. Solve closure condition (1) $\Rightarrow q_J$.
4. Compute global kinematics in $fragment\,2$ including joint J.

$\dot{q}_J$ and $\ddot{q}_J$ can be computed analogously from $\dot{g}$ and $\ddot{g}$.

Inverse dynamics determines all generalized forces $\boldsymbol{\tau}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}})$, i.e., the joint forces and torques, from a given state of motion of an MBS. On the one hand, joint J is free which in turn requires the joint driving torque τ_J to be zero. On the other hand, the connecting rod CR introduces an unknown constraint force λ_{CR} acting between the spherical joints $S1$ and $S2$. The method relies on the idea to apply the constraint force such that it compensates for a torque $\tau 0_J$ introduced merely by the bodies in $fragment\,2$ and comprises 3 steps:

1. Compute inverse dynamics in $fragment\,2$ without taking into account for the constraint imposed by CR. The result is the torque $\tau 0_J$ acting in joint J.
2. Determine the constraint force $\lambda_{CR} = f(\tau 0_J, \mathbf{r}^{K,S2})$.
3. Compute inverse dynamics in the complete mechanism, including $CR \Rightarrow \tau 0_J \equiv 0$.

The computation of $q_J, \dot{q}_J, \ddot{q}_J$, and λ_{CR} involves some homogenuous transformations, and a second evaluation of kinematics and dynamics in $fragment\,2$. A numerical effort even tolerable in a real-time system.

2.4.2 Handling Coupled Loops

Analysis of the robot in the example in Section 2.3 reveals two coupled kinematic loops. Figure 6 shows the mechanism configuration.

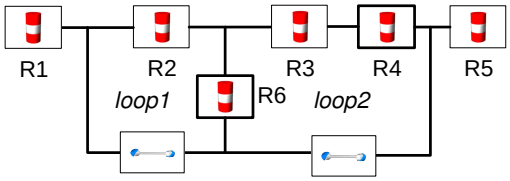

Figure 6: Iconic view of the example robot model, showing joints, connecting rods, and their interconnection. Bold boxes correspond to free joints, fine boxes to driven joints.

Loop 1 contains revolute joint R6 as a free joint and loop 2 R4 and R6. After short inspection the sequence of solution is obvious: First solve loop 1 for angle of joint R6, then solve loop 2 for the remaining angle of joint R4.

When dealing with more complex configurations, the model topology and, therefore, the structure of the equations may be obscure. To alleviate the user and to avoid errors, the analysis is automatically performed by the library during model setup. First a directed a-cyclic graph is created from the model description, with transmission objects

as nodes and connections as edges. When kinematic loops are present, the dedicated Assignment-algorithm [16] is employed to determine the association of free joint variables and loops. The final step identifies strongly connected parts of the graph in order to find the correct sequence of solutions [17].

3 Functionality

The class `Manipulator` is the library core. Its tasks are:

- Analysis of the model description to avoid non-valid or non-physical models, process closed-loop systems in the way shown above, and extract valuable information, e.g., dof and physical parameters.
- Setup objects to synthesize optimized code on-line in the form of special solver objects for algorithms such as the computation of a Jacobian. These solvers make use of the local transformations incorporated in every transmission object.
- Interface functions to provide user-friendly access to the solvers. These are briefly described in the following sections.

The following code fragment is a continuation of Fig. 4 and creates a manipulator object and sets up the robot model:

```
Manipulator ExampleRobot;
connect(ExampleRobot,base);
ExampleRobot.init();
```

3.1 Kinematics

3.1.1 Forward Kinematics

In the robotics domain forward kinematics refers to the computation of position and orientation, usually of the end-effector of the manipulator, from given joint variables. The following code fragment sets the joint angles of the example robot from Fig. 4, performs forward kinematics of the complete model, and retrieves the position of a certain connector:

```
double new_q[4]={PI,0,1,PI/2};
ExampleRobot.dof().set(&new_q[0]);
ExampleRobot.do_position();
cout << Payload.frame_a.get_position();
```

To save run-time partial kinematics can be performed by supplying a certain frame in the `do_position` call:

```
ExampleRobot.do_position(R3.frame_a);
cout << R3.frame_a.get_position();
```

3.1.2 Inverse Kinematics

Other work has shown that it is difficult to obtain joint variables from a given posititon $\mathbf{r}^{0,A}$ and orientation $\mathbf{T}^{0,A}$ of the end-effector A, since the relation $\mathbf{q} = G(\mathbf{r}^{0,A}, \mathbf{T}^{0,A})$ in general requires to solve a system of non-linear equations. This is avoided by the restriction to explicit solutions, which, however, exist for many manipulators of practical interest. In order to reap the benefits of fast solutions in a real-time environment, a user provides an implementation of G that is dedicated to a certain kinematic structure. The integration of this additional code is accomplished elegantly via an assembly. This assembly is a class definition containing (at least) three blocks: (i) a definition of the manipulator model, comprising components and their interconnections, (ii) connectors to connect this (sub-)structure to other parts, such as tools, and (iii) a dedicated set of functions implementing the associated inverse kinematics.

3.1.3 Manipulator Jacobians

The manipulator Jacobian $J(\mathbf{q})$ [18] with respect to a frame A fixed to the model and a reference frame 0 is defined by

$$\begin{pmatrix} \dot{\mathbf{r}}^{0,A} \\ \boldsymbol{\omega}^{0,A} \end{pmatrix} = J(\mathbf{q})^{0,A} \cdot \dot{\mathbf{q}}\,. \tag{3}$$

Though the library uses a Jacobian-free formulation of velocity kinematics, it can be used to compute J analytically without any discretization errors. Column i results when applying unit velocities $\dot{\mathbf{q}} = \mathbf{u}_i$ and calculating velocity kinematics. This method is not as efficient as dedicated algorithms used for kinematic chains [19], but in turn applicable to closed-chained robots. The following example code computes the Jacobian with respect to the B-frame in joint `R5`:

```
Matrix J=ExampleRobot.do_Jacobian(R5.frame_b);
```

In off-line tasks such as calibration, imprecise or uncertain physical parameters of the robot are identified. The employed numerical techniques require the sensitivity of the end-effector position of the manipulator in terms of these physical parameters. The method used for Eq. 3 is extended to compute a generalized Jacobian matrix $J(\mathbf{q}, \boldsymbol{\lambda})$ with respect to arbitrary kinematical physical parameters $\boldsymbol{\lambda} := (\lambda_1, \cdots, \lambda_m)$, e.g., the components λ_i may be dimensions of a link. This Jacobian is defined as

$$J(\mathbf{q}, \boldsymbol{\lambda})^{0,A} := \begin{pmatrix} \boldsymbol{\rho}_1 & \cdots & \boldsymbol{\rho}_m \\ \boldsymbol{\chi}_1 & \cdots & \boldsymbol{\chi}_m \end{pmatrix} \in R^{6\times m}\,,$$

where

$$\boldsymbol{\rho}_i := \frac{\partial \mathbf{r}^{0,A}}{\partial \lambda_i} \quad \text{and} \quad \frac{\partial \mathbf{T}^{0,A}}{\partial \lambda_i} = -\mathrm{skew}(\boldsymbol{\chi}_i) \cdot \mathbf{T}^{0,A}.$$

This approach applies to all *physical* kinematic parameters, as long as the transmission objects provide a pseudo-velocity $(\boldsymbol{\rho}, \boldsymbol{\chi})$ with respect to λ_i. The following example results in a 6 by 3 matrix, returning the sensitivity of the position of `frame_b` of joint `R5` in terms of the direction of axis of rotation `n` of joint `R3`.

```
Matrix J=
  ExampleRobot.do_Jacobian(R3.n,R5.frame_b);
```

3.2 Dynamics

Given a certain condition of motion $\{\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}\}$ inverse dynamics computes the joint forces and torques $\boldsymbol{\tau}$. The solution is implemented using the RNE method [4] augmented by the method presented in Section 2.4.1. The following code fragment shows one evaluation of inverse dynamics of the example robot:

```
double q_qd_qdd[12]={ ... };
ExampleRobot.dof().set_all(&q_qd_qdd[0]);
```

```
ExampleRobot.do_inverseDynamics();
cout << ExampleRobot.dof().get_force();
```

Dynamic calibration algorithms often require the sensitivity of the joint torques with respect to arbitrary inertial parameters. A modified backward iteration of the RNE was implemented to compute these derivatives analytically. The following example code shows the calculation of $\frac{\partial \boldsymbol{\tau}}{\partial m_{PayLoad}}$:

```
ExampleRobot.do_forcePrime(PayLoad.m());
cout << ExampleRobot.dof().get_forcePrime();
```

The method `do_inertiaMatrix` implements the algorithm taken from [5] to compute the joint inertia matrix M of the model, needed to solve for joint accelerations $\ddot{\mathbf{q}}$ from given $\mathbf{q}$, $\dot{\mathbf{q}}$, $\boldsymbol{\tau}$, and external forces. Using the inverse dynamics algorithm as a basis it is inherently capable of dealing with the class of closed-loop systems described in Section 2.4.1.

3.3 Re-Configuration of the Model

In certain applications it is desirable to re-configure an existing model, even on-line, e.g., in case of tooling of a manufacturing robot or treatment of contact dynamics, when picking a part with a dextrous robot hand. Because of the modular conception of the library it is possible to remove an existing connection between two components of the current model. As a counterpart to `connect` the `disconnect` function detaches two objects from each other. The following example again refers to the example robot. The `Payload` object is disconnected from the robot model and two more revolute joints and an arbitrary tool are attached instead, resulting in a 6 dof tooled robot with a spherical wrist.

```
disconnect(R5.frame_b,Payload.frame_a);
RevoluteJoint R6(YAXIS); RevoluteJoint R7(ZAXIS);
AnyTool tool(...);
connect(R5.frame_b,R6.frame_a);
connect(R6.frame_b,R7.frame_a);
connect(R7.frame_b,tool.frame_a);
ExampleRobot.init();
```

The `init` call is necessary to instantaneously effect the model changes to prevent non-valid models during reconfiguration.

4 Performance Aspects

The library was tested on a standard 200 MHz Pentium PC, using the Microsoft Visual C++ 6.0 Compiler. The only components of the example carrying inertial properties are the `Link` objects, with full inertia tensor taken into account. Computing times for the example robot definition from Fig. 4 are shown in Table 1.

Present day robot controllers employ a sampling rate of 1 kHz or higher, and control schemes such as computed torque [18] need one evaluation of inverse dynamics of the robot model per cycle. The results shown in Table 1 suggest that our approach is well suited for real-time on-line robot control.

Function	Comp.time
`do_position()`	17μs
`do_Jacobian()`	58μs
`do_inverseDynamics()`	75μs

Table 1: Time per one evaluation of different functions of the robot model in Fig. 4 run on a 200 MHz Pentium PC.

A comparison to other implementations reveals the performance of this work. The results for one evaluation of inverse dynamics of a chain-structured 6 dof robot are presented in Table 2, and were performed using a model that contains 6 revolute joints and 6 links with full inertia tensor, obtained on the same computer as already mentioned.

Used approach	Language	Comp.time
Manually implem. RNE	C	23μs
Dymola generated code (symbolically) [20, 12]	C	12μs
MATLAB MEX-File [8]	C	3.3ms
ROBOOP library [9]	C++	1.2ms
This work	C++	36μs

Table 2: Time per one evaluation of inverse dynamics of a 6 dof robot. Comparison of different implementations.

As to be expected the symbolically and manually derived solutions show the best performance. The manually coded inverse dynamics routine was an RNE tailored to kinematic chains, not containing any function calls. For the approach described in this paper several steps have been worked out to achieve computational performance comparable to manually optimized C-code:

- Local transformations in every object are sub-divided in separate parts for position, velocity, and acceleration kinematics and dynamics (see Fig. 1). These parts can be highly optimized without sacrificing readability, and combined flexibly to form the global algorithms, e.g., for Jacobian computation. Moreover software testing effort is greatly reduced.
- The need to use structural information of the equations of motion is evidenced by the superior performance of symbolically generated code. Structural information provided through the object-oriented modeling is exploited to reduce floating point operations, e.g., that a revolute joint represents a *plane* rotation.
- Connected objects share the same data, no data is copied during execution of transformations.
- Virtual calls are expensive, therefore void calls are automatically detected and suppressed.
- STL and a proprietary optimized vector-/matrix-class for 3D operation are used, the public domain Matrix-Template-Library (MTL) [21] for high-level matrix operations.

- Objects with rigid connections are automatically merged to result in one rigid body in order to increase performance.

One fundamental drawback when using C++ is the unavoidable cost for accessing code through interfaces. The required virtual calls are rather expensive, especially when the accessed code portion is small, but are mandatory to handle all transmission objects uniformly and transparently in the processes of model-setup and re-configuration.

5 Conclusions

In this work the design of a C++ library for kinematics and dynamics computations in real-time robot control was presented. The library was developed with the goals of flexibility and high computational efficiency in mind, to satisfy the constraints imposed by a real-time embedded system. Efficient standard algorithms for tree-structured models were extended for a class of closed-loop models, which allow explicit and therefore fast and reliable solution. A convenient and intuitive high-level model description is used to define the computational model, which offers the possibility to perform re-configuration of the model on-line. Applications which demand gradient information are facilitated by providing various Jacobians and force derivatives. Because a high-level programming language has been used, the integration in robot control and trajectory planning algorithms is straightforward and the porting to different computational platforms is facilitated. It is shown that the efficiency obtained is close to that of manually optimized C-Code. One evaluation of the inverse dynamics of a 6 dof robot model is computed in about 36μs on a 200MHz Pentium PC.

6 Acknowledgements

The authors would like to thank Stefan Pieters, AMATEC Robotics, Germany, who initiated and continuously supported this work. One author (R. H.) is funded in part by AMATEC Robotics.

References

[1] J.-C. Latombe. *Robot motion planning*. Kluver Academic Publishers, 1991.

[2] J. E. Bobrow, S. Dubowsky, and J. S. Gibson. Time-optimal control of robotic manipulators along specified paths. *The International Journal of Robotics Research*, 4(3):3–17, 1985.

[3] Zvi S. Roth, Benjamin W. Mooring, and Bahram Ravani. An overview of robot calibration. *IEEE Journal of Robotics and Automation*, RA-3(5):377–385, 1987.

[4] J. Y. S. Luh, M. W. Walker, and R. P. C. Paul. On-line computational scheme for mechanical manipulators. *ASME J. of Dynamic Systems Meas. and Control*, 102:69–76, 1980.

[5] M. W. Walker and D. E. Orin. Efficient dynamic computer simulation of robotic mechanisms. *ASME J. of Dynamic Systems Meas. and Control*, 104:205–211, 1982.

[6] M. C. Leu and N. Hemati. Automated symbolic derivation of dynamic equations for mechanical manipulators. *ASME J. of Dynamic Systems Meas. and Control*, 108:172–179, 1986.

[7] Harry G. Kwatny and Gilmer L. Blankenship. Symbolic construction of models for multi-body dynamics. *IEEE Transactions on Robotics and Automation*, RA-11(2):271–281, 1995.

[8] P.I. Corke. A robotics toolbox for MATLAB. *IEEE Robotics and Automation Magazine*, 3(1):24–32, March 1996.

[9] Richard Gourdeau. Object-oriented programming for robotic manipulator simulation. *IEEE Robotics and Automation Magazine*, 9:21–29, 1997.

[10] A. Kecskeméthy. *Objekt-orientierte Modellierung der Dynamik von Mehrkörpersystemen mit Hilfe von Übertragungselementen*. PhD thesis, Universität Duisburg, 1993.

[11] Manfred Möller. *Ein Verfahren zur automatischen Analyse der Kinematik mehrschleifiger räumlicher Mechanismen*. PhD thesis, Universität Stuttgart, 1992.

[12] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. Fortschrittberichte VDI, Reihe 20, Nr.147, 1995.

[13] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial, Version 1.4*. Modelica Association. http://www.Modelica.org, 2000.

[14] J. Denavit and R.S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of applied mechanics*, pages 215–221, June 1955.

[15] Robert E. Roberson and Richard Schwertassek. *Dynamics of multibody systems*. Springer-Verlag, 1988.

[16] C.C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal of scientific and statistical computing*, 9:213–231, 1988.

[17] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[18] Mark W. Spong and M.Vidyasagar. *Robot dynamics and control*. John Wiley & sons, 1989.

[19] D. E. Orin and William W. Schrader. Efficient computation of the Jacobian of robot manipulators. *The International Journal of Robotics Research*, 3(4):66–75, 1984.

[20] DynaSim AB. Dymola - Dynamic Modeling Laboratory. http://www.dynasim.se.

[21] MTL Homepage. The matrix template library. http://www.lsc.nd.edu/research/mtl/.