

Andrés Kecskeméthy

M□BILE
Version 1.3
User's Guide

Alle Rechte vorbehalten

Nachdruck, auch auszugsweise, verboten

Kein Teil dieses Werkes darf ohne schriftliche Einwilligung des Autors in irgendeiner Form, auch nicht zum Zwecke der Unterrichtsgestaltung, reproduziert, oder unter Verwendung elektronischer Systeme vervielfältigt oder verbreitet werden.

©1993, 1994, 1995, 1996, 1997, 1998, 1999 Andrés Kecskeméthy, Institut für Mechanik und Getriebelehre, Technische Universität Graz

Important Notice:

BY OPENING THE SEALING OF THE SOFTWARE ENVELOPE, OR BY LOADING THE SOFTWARE ON YOUR MACHINE, WHICHEVER APPLICABLE, YOU ACKNOWLEDGE THE LIMITED WARRANTY AND DISCLAIMER SPECIFIED BELOW AND CONSENT TO ALL THE CONDITIONS MADE THEREIN. IF YOU DO NOT FULLY APPROVE THE TERMS UNDER WHICH THIS SOFTWARE IS LICENSED, DO NOT OPEN THE SOFTWARE ENVELOPE AND RETURN IT INTACT TOGETHER WITH THE WRITTEN MATERIAL HANDED OVER TO YOU, OR DO NOT LOAD THE SOFTWARE ON YOUR MACHINE, AND DELETE ALL ITEMS RELATED TO THIS SOFTWARE FROM YOUR MACHINE, FOR A FULL REFUND OF THE PURCHASE PRICE.

Limited Warranty and Disclaimer

YOU ACKNOWLEDGE THAT THE SOFTWARE MAY NOT SATISFY ALL YOUR REQUIREMENTS OR BE FREE FROM DEFECTS. BY THE PRESENT LIMITED WARRANTY IT IS WARRANTED THAT THE MAGNETIC MEDIA ON WHICH THE SOFTWARE IS RECORDED IS FREE FROM DEFECTS IN MATERIALS AND WORKMANSHIP UNDER NORMAL USE FOR 90 DAYS FROM PURCHASE. HOWEVER, THE SOFTWARE AND THE ACCOMPANYING WRITTEN MATERIALS ARE LICENSED AS IS. ALL IMPLIED WARRANTIES AND CONDITIONS (INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE) ARE DISCLAIMED AS TO THE SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS AND LIMITED TO 90 DAYS AS TO THE MAGNETIC MEDIA. YOUR EXCLUSIVE REMEDY FOR BREACH OF WARRANTY WILL BE THE REPLACEMENT OF THE MAGNETIC MEDIA OR REFUND OF THE PURCHASE PRICE. IN NO EVENT WILL ANY OFFICER OR EMPLOYEE OF THE TECHNICAL UNIVERSITY OF GRAZ OR THE DEVELOPER OF MOBILE BE LIABLE TO YOU FOR CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE), WHETHER FORESEEABLE OR UNFORESEEABLE, ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE OR ACCOMPANYING WRITTEN MATERIALS, REGARDLESS OF THE BASIS OF THE CLAIM AND EVEN IF THE DEVELOPER OF MOBILE OR AN OFFICIAL OR EMPLOYEE OF THE TECHNICAL UNIVERSITY OF GRAZ HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Contents

1	Preface	1
1.1	What is M□BILE?	1
1.2	Intended Audience	1
1.3	Scope of M□BILE 1.3	2
1.4	Scope and Organization of this Manual	2
1.5	File Hierarchy of the M□BILE Package	3
1.6	Compiler Issues	4
1.7	Style and Symbol Conventions	5
1.8	Acknowledgements	7
2	Overview	10
2.1	Structure of M□BILE	10
2.2	Example: Analysis of a Simple Pendulum	12
2.2.1	Dissection and Re-Assembly of the System	12
2.2.2	Calculating Dynamic Properties	14
2.2.3	Automatic Integration of Dynamical Equations	16
2.3	Summary	18
3	Basic Mathematical Objects	19
3.1	The Universal Neutral Element <code>MoNullState</code>	21
3.2	The Basic Scalar Types <code>MoReal</code> and <code>MoAngle</code>	22
3.3	Vectors and Matrices	23
3.3.1	Vectors	23
3.3.2	Matrices	24
3.4	Kinetostatic State Objects	27
3.4.1	Scalar Kinetostatic State Objects (“ <code>MoStateVariable</code> ”)	28
3.4.2	Spatial Kinetostatic State Objects (“ <code>MoFrame</code> ”)	32
4	Basic Kinetostatic Transmission Elements	35

4.1	Overview of Supplied Kinetostatic Transmission Elements	35
4.2	Generic Properties of Kinetostatic Transmission Elements	36
4.2.1	Model of a Kinetostatic Transmission Element	37
4.2.2	Invoking Motion and Force Transmission	39
4.2.3	Selection of Motion and Force Transmission Subtasks	40
4.3	Basic Transmission Elements: Links, Joints and Chains	43
4.3.1	The Object “MoRigidLink”	44
4.3.2	The Object “MoElementaryJoint”	46
4.3.3	The Object “MoMapChain”	48
4.3.4	A simple example	49
4.4	Force and Mass Elements	51
4.4.1	Force Elements	51
4.4.2	Mass Elements	53
4.5	Example: Modeling of the Inverse Dynamics of a SCARA robot	54
5	Objects for Closure of Loops	57
5.1	Basic Methods for Formulating Loop Closure Conditions	58
5.1.1	Example: Inverse Dynamics of a Spatial Shaker Mechanism	60
5.2	Measurement Objects	64
5.2.1	Basic Properties of Measurements	64
5.2.2	Self-Reconfiguring Measurements	66
5.2.3	Lists of Measurements	68
5.2.4	Spatial Measurements	68
5.2.5	Scalar Measurements	71
5.2.6	Constructing Measurements with Different Numbers of Frames	74
5.2.7	Optimizing Performance by Specification of Active Branches	77
5.2.8	Interlinking Measurements	79
5.3	Objects for Solving Constraints	81
5.3.1	Implicit Solvers	82
5.3.2	Explicit Solvers	82

5.4	Examples	83
5.4.1	Body Assembly of a Spatial Four-bar Mechanism	83
5.4.2	Joint Assembly of a Shaker Mechanism	87
5.4.3	Segment Assembly of a Shaker Mechanism	90
6	Generating and Solving Dynamic Equations	94
6.1	The class <code>MoEqmBuilder</code>	94
6.2	Generating Ordinary Differential Equations in State-Space Form	97
6.2.1	The Class <code>MoMechanicalSystem</code>	98
6.3	Solving the Differential Equations	99
6.4	Example: Dynamics of a Triple Pendulum	100
7	Graphic Rendering and Animation	103
7.1	Creating a Graphics Interface	103
7.2	Importing Inventor Files	107
7.3	Prescribing Motion by Sliders	107
7.4	Realizing Autonomous Animations	110
7.5	Further Animation Capabilities	111
8	M□BILE for PC	112
8.1	Installation	112
8.2	M□BILE for PC with Open Inventor Graphic Interface	113
8.3	M□BILE for PC with OpenGL Graphic Interface	117
8.4	M□BILE for PC with Graphic User Interface	120

List of Figures

2.1	Objects in multibody systems	11
2.2	Modeling of a simple pendulum	12
2.3	Graphic representation of the pendulum example	18
3.1	The two types of algebraic scalar objects in M \square BILE	22
3.2	Class hierarchy for the different types of matrices in M \square BILE	25
3.3	State subentries of state objects	28
3.4	State objects as connectors between transmission elements	29
3.5	Structure of a scalar variable (generalized coordinate).	29
3.6	Components of a moving frame	33
4.1	Hierarchy of kinetostatic transmission elements of M \square BILE (excerpt) . . .	36
4.2	Model of a kinetostatic transmission element	38
4.3	Terms and computational steps involved in the transmission of motion and forces	40
4.4	Model of a rigid link.	44
4.5	Model of an elementary joint.	47
4.6	A Simple Manipulator	50
4.7	Elementary force element attached to a scalar measurement object	52
4.8	Model of a mass element.	53
4.9	Modeling of the inverse dynamics of a SCARA robot	55
5.1	Comparision of tree-type and closed-loop systems	57
5.2	Three basic methods for modeling loops in M \square BILE	59
5.3	Analysis of a shaker mechanism	61
5.4	Example of a “chord”	65
5.5	Example of a measurement for a moving object	67
5.6	A spatial measurement	69
5.7	Basic form of a scalar measurement	72
5.8	Geometric entities involved in the measurements between points and planes	73

5.9	Entities of interest for the topological types of measurement	74
5.10	Types of measurements based on number of frames	76
5.11	Measurement Object for Complementary Variable	80
5.12	Modelling of the spatial Four-bar mechanism	84
5.13	Modeling of the Dynamics of a Shaker Mechanism	87
5.14	Modelling of the Shaker (explicit solution)	90
6.1	Model of the inverse dynamics of a multibody system.	95
6.2	Modelling of the TriplePendulum	101
7.1	Basic structure of the M□BILE-Inventor interface	103
7.2	Overview of the Inventor interface for M□BILE	106
7.3	An example of the use of slider widgets	108
8.1	Setting a environment variable and a path	113
8.2	Generate a new Project	114
8.3	Setting the path to header- and library files	117
8.4	Start of a model with Open Inventor	118
8.5	Start of a model with OpenGL	120
8.6	Interactive models	121

List of Tables

1.1	Header files for the basic mathematical objects	8
1.2	Header files for the basic kinetostatic transmission elements	8
1.3	Header files for constraint generation and solution	9
1.4	Header files for generation and solution of equations of motion	9
1.5	Header files for animation	9
1.6	Container header files	9
2.1	Iconic representation of the elementary objects of M□BILE	13
3.1	Overview of the basic mathematical objects of M□BILE	19
3.2	Precedence of operators in M□BILE	20
3.3	Objects that are not automatically initialized in M□BILE	21
3.4	Overview of operations for objects of type MoAngle	23
3.5	Overview of operations for objects of type MoVector	24
3.6	Properties of the different types of three-dimensional matrices	25
3.7	General operations for objects of type MoMatrix	26
3.8	Special operations for objects of type MoInertiaTensor	27
3.9	Special operations for objects of type MoRotationMatrix	27
3.10	Special operations for objects of type MoXRotationMatrix, MoYRotation- Matrix and MoZRotationMatrix	27
3.11	Subentries for scalar kinetostatic state objects of type “MoStateVariable”	30
3.12	Subentries for kinetostatic state objects of type “MoFrame”	33
4.1	Selection of kinetostatic transmission elements in M□BILE	37
4.2	Types of motion and force invocation	40
4.3	Meaning of the terms in Fig. 4.3	41
4.4	Possible values for the motion subtask selection parameter	42
4.5	Possible values for the force subtask selection parameter	42
5.1	Geometric types of measurement objects	66

5.2	Basic formulas for spatial measurements (kinematics)	70
5.3	Basic formulas for spatial measurements (statics)	71
5.4	Elements of the array ‘state’ for spatial measurements	71
5.5	Basic geometric types of scalar measurements	73
5.6	Types of measurements involving different numbers of frames	75
5.7	Optional parameters for performance optimization of measurement objects	78
7.1	Default rendering geometry for the basic M□BILE objects	105
7.2	Importing Inventor files into a M□BILE model	107

1 Preface

1.1 What is MOBILE?

MOBILE is an object-oriented programming package designed for the modeling of multi-body systems. Its main features are

- Intuitive representation of mechanical entities as objects capable of transmitting motion and force across the system.
- Direct modeling of mechanical systems as executable programs, allowing the user to imbed the resulting modules in exisisting libraries.
- Open, building-block system design, making it possible to extend the provided library in any direction.
- Scalable approach, treating all mechanical systems in a unified manner.
- Responsibility-driven client-server implementation, simplifying the task of invoking the required functions and of implementing own costumized modules.
- Portable and efficient implementation, based on the object-oriented programming language C++.
- Built-in interfaces for three-dimensional graphic libraries for animation with direct user feed-back. User interaction includes click-and-drag features for on-line kinematics, statics and dynamics (this last feature may depend on system complexity and computer resources).

1.2 Intended Audience

This manual is addressed to users with a certain amount of experience with the modeling and simulation of mechanical systems.

Some familiarity with the C++ programming language is needed to understand and to apply the concepts described below. However, it is not necessary to master all of the many possibilities of the programming language C++ just to generate a MOBILE model. This is necessary only for developers planning to extend the MOBILE package.

From the theoretical point of view, some acquaintance is required with the basic concepts of kinematic and dynamic analysis of spatial mechanical systems. This knowledge is only necessary at a very abstract level, such as for deciding in which sequence a set of mechanical components needs to be traversed, which kind of closure conditions arise in a chain forming a closed loop, which set of variables to use as independent generalized coordinates of a subsystem etc.

1.3 Scope of M□BILE 1.3

M□BILE 1.3 represents the entry-level library for the modeling of multibody systems. It covers the following topics:

- Basic mathematical objects and related operators for calculations in spatial dynamics: scalars, vectors, matrices, orthogonal transformations, elementary transformations, inertia tensors.
- Elementary building blocks for multibody systems: reference frames, angular and linear variables, elementary joints (prismatic and revolute), rigid links, elementary measurements mapping spatial motion to scalar quantities and tuples thereof, objects for creating composite chains of transmission elements.
- Elementary force elements (spring/damper, gravitation).
- Objects for the resolution of constraint equations, either in closed-form or iteratively.
- Objects for the generation of the equations of motion.
- Objects for the numerical integration of the dynamical equations.

More sophisticated (and also more efficient) modeling techniques for multibody systems, as for example sparse-matrix modeling of Jacobians, efficient transmission of inertia properties, etc., will be included in the additional package M□BILE 2.x, which is currently under development. Further extensions, such as elasticity effects, hydraulics and control theory, are also under development and will be included in M□BILE 3.x.

1.4 Scope and Organization of this Manual

This manual describes the basic software implemented in M□BILE 1.3 and its application to multibody systems. Specifically, the definition, use, and application of the objects listed above are described at a syntactical level and illustrated by several examples at a tutorial level. The manual does not cover details of the language C++ and of the implementation of the package M□BILE. Readers interested in these topics are recommended to consult the related literature and/or the program listings.

The manual is organized in two parts

- Part I of the manual gives an introduction to the objects of M□BILE, describing their functionality and illustrating their use by several examples. Moreover, some theoretical background information has been inserted for readers interested in the underlying computations of M□BILE. These insertions are not essential for the use of M□BILE and can be skipped by the casual reader.

- Part II of the manual comprises the so-called “**M□BILE Reference Sheets**”, a collection of detailed syntax description pages for each entity introduced by the M□BILE package.

Part I is structured as follows:

- **Chapter 2** gives a short overview of the capabilities of M□BILE 1.3
- **Chapter 3** describes the basic mathematical objects used in conjunction with the modeling of multibody systems
- **Chapter 4** is concerned with the basic mechanical modeling elements of M□BILE 1.3, which are termed “*kinetostatical transmission elements*” and which constitute the basis for all objects described later on
- **Chapter 5** is devoted to the problem of formulating and solving closed loops
- **Chapter 6** describes the objects for generating and solving the dynamical equations of multibody systems
- **Chapter 7** gives an overview of the interface of the M□BILE package for graphic animation

In general, the material presented in each chapter builds upon the material contained in the previous ones. The reader is encouraged to first browse the chapters in the provided order and then to return to individual chapters to work on the details.

1.5 File Hierarchy of the M□BILE Package

The software of M□BILE is organized into several modules, each module representing a particular group of modeling elements. For example, there are modules for joints, links, generators of closed-form solutions, etc.

Each module in M□BILE consists of two parts: a file defining the interface of the module, the so-called *header file*, and a file defining the executable portion of the module, the so-called *implementation file*. Header files have the suffix “.h”, while implementation files have the suffix “.C”. Depending on which type of license you have purchased, you may or may not possess the implementation files. The header files are shipped with every license of M□BILE. The header files currently supplied with the M□BILE software are summarized in tables (1.1), (1.2), (1.3), (1.4), and (1.5). For ease of use, header files are also summarized by groups in the container header files displayed in Table 1.6. By including one of these container header files, the user includes automatically all of the header files of the corresponding groups. This saves some typing, leading to slightly longer compilation times, although program size is not affected.

In order to access the objects of a module, one must include the corresponding header file in the program. Including a header file is accomplished by the directive

```
#include <Mobile/module-name.h>
```

Failure in including the correct header file will result in a large number of compiler errors, such as

```
CC: "Example.C", line 62: error: MoElementaryJoint R1 : MoElementaryJoint is not a type name (1314)
CC: "Example.C", line 63: error: MoElementaryJoint R2 : MoElementaryJoint is not a type name (1314)
CC: "Example.C", line 64: error: MoElementaryJoint R3 : MoElementaryJoint is not a type name (1314)
```

As shipped from factory, the M□BILE package is organized in the following directories

directory name	description
<code>\$MOBILE_HOME_DIR/Mobile</code>	header files
<code>\$MOBILE_HOME_DIR/src</code>	implementation files (not always available)
<code>\$MOBILE_HOME_DIR/lib</code>	run-time libraries (e.g., <code>libmobile.a</code>)
<code>\$MOBILE_HOME_DIR/bin</code>	utility programs
<code>\$MOBILE_HOME_DIR/examples</code>	examples and test-files
<code>\$MOBILE_HOME_DIR/Inventor</code>	Inventor graphics library

The environment variable `$MOBILE_HOME_DIR` should point to the home directory of the M□BILE package. This directory is set by the system administrator during installation of M□BILE. A typical value of `$MOBILE_HOME_DIR` is `"/usr/people/mobile"`. However, there might be a different setting on your system. Please consult your system manager for obtaining information about the location of the M□BILE home directory. Depending on which shell you are using, setting the value of the environment variable `$MOBILE_HOME_DIR` takes on the form (assuming the home directory for M□BILE is `/usr/people/mobile`)

```
Korn shell (ksh):  export MOBILE_HOME_DIR=/usr/people/mobile
C shell (csh):    setenv MOBILE_HOME_DIR /usr/people/mobile
```

You can check the value of the environment variable `$MOBILE_HOME_DIR` by typing

```
echo $MOBILE_HOME_DIR
```

1.6 Compiler Issues

M□BILE is written in standard C++ Version 2.0. In order to obtain a running program of a M□BILE model, one must compile it using the C++ compiler installed in the system. A typical compiler invocation on a UNIX system has the following appearance:

```
CC filename.C -I$MOBILE_HOME_DIR -L$MOBILE_HOME_DIR/lib/ -lmobile -ofilename
```

Here, `filename.C` is the program containing the model and `CC` is the command for invoking the C++ compiler. The character strings following `-I` and `-L` instruct the compiler

where to look for the header files and the libraries of the MOBILE package. Note that we are using here the environment variable `$MOBILE_HOME_DIR` defined above. The argument `-lmobile` instructs the compiler to load the MOBILE library `libmobile.a`. The argument `-ofilename` instructs the compiler to create an executable program with the name “filename” containing the model. Executing this file will run the model.

Another technique for compiling MOBILE models is to use the UNIX tool `make`. Instead of defining compiler invocation parameters anew for each model, one can also lay down the compilation rules in a file named `Makefile`, and locate this file in the directory in which the MOBILE model is placed. Compilation is then automatically accomplished by the command

```
make filename
```

Examples of appropriate `Makefile` settings for MOBILE models can be found in

```
$MOBILE_HOME_DIR/examples/Makefile
$MOBILE_HOME_DIR/examples/Inventor/Makefile
```

A `Makefile` template that is suitable for full MOBILE models including Inventor and NAG capabilities is included under

```
$MOBILE_HOME_DIR/examples/Makefile-Inventor-NAG-Template
```

Instructions for generating, compiling and executing Mobile models on Windows PC (98 or NT) are more elaborate. The reader is referred to Chapter 8 for corresponding details.

1.7 Style and Symbol Conventions

Throughout this manual, the following syntactical and lexicographic conventions are used:

- **Program listings**, examples and outputs are rendered in `small courier font`. For example, a program fragment describing a simple pendulum, and calculating and printing the position of its center of mass is:

```
#include <Mobile/MoMapChain.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoElementaryJoint.h>
main() {
  MoFrame K1, K2, K3 ;
  MoVector v(1,0,1) ; MoAngularVariable beta;
  MoElementaryJoint R1(K1,K2,beta) ; MoRigidLink L1(K2,K3,v) ;
  MoMapChain Pendulum ; Pendulum << R1 << L1 ;
  beta.q = PI/2 ; Pendulum.doMotion();
  cout << "Position = " << K3.R * K3.r << "\n";
}
```

After compiling the program, one can execute the code and obtain the result like this

```
$ Pendulum
$ Position = ( 0.0 , 1.0 , 1.0 )
$ _
```

- **Class names and keywords** in syntax descriptions are typed in `courier font`, as for example in

```
MoVector name ;
```

Class names and keywords, must be typed exactly as shown.

- **Identifiers**, i. e. variable names, are printed in *slanted courier font*. You can replace the names by any character string allowed as an identifier.
- **Types of arguments** passed to functions are typeset in *<italic courier>* enclosed by angle brackets. Replace these entries by a permissible identifier or variable value of the type indicated.
- **Optional parameters** are enclosed in square brackets, as in [ArgumentType]. Optional parameters can be left out, in which case they are given previously defined default values. In MOBILE, optional parameters are not used very frequently. Instead, one will find several definitions of function calls which differ in the type of the arguments. This mechanism is known as “*polymorphism*”.
- **Alternative choices** are characterized by a vertical bar ‘|’ separating the corresponding items. An example is

```
MoElementaryJoint name ( <MoFrame>, <MoFrame>,
                          <MoLinearVariable>,
                          xAxis | yAxis | zAxis )
```

Here, one of the three choices `xAxis`, `yAxis`, `zAxis` should be typed as the fourth argument of this constructor.

- **Variable number of arguments** are indicated by ellipses “...”. These indicate that the last pattern can be repeated an arbitrary number of times. For example, a chain of transmission elements might be defined like this:

```
MoMapChain name ;
name << map1 [ << map2 ... ] ;
```

- **Overloaded operators, functions, constructors and member data** are indicated explicitly in the reference sheets using the C++ syntax for class definition. This assumes some familiarity of the reader with C++. However, this knowledge is limited to recognizing the type and number of arguments passed to the functions or included in the data definition.

Note that `typewriter font` is always to be used verbatim.

1.8 Acknowledgements

This work was supported by the Laboratory of Mechatronics (Fachgebiet Mechatronik) at the Gerhard-Mercator-University of Duisburg. The author thanks the head of the Department, Prof. Dr.-Ing. habil. M. Hiller, for his support during the development of this software. Credits are also due to Mr. Thorsten Krupp for much of the coding of the package, to Mr. Christian Schuster for the porting of M□BILE to PCs, as well as to Mr. Martin Schneider for many valuable suggestions and bug reports.

module	functionality
MoConfig.h	hardware platform and operating system configuration file
MoVersion.h	version number of current M <small>OB</small> ILE installation
MoReal.h	floating point numbers
MoRealStack.h	stacks of floating point numbers
MoAngle.h	basic properties of angles
MoNullState.h	generic zero state
MoAxis.h	objects for selection of axes
MoStateVariable.h	linear and angular scalar variables
MoVariableList.h	lists of variables
MoVector.h	three-dimensional vectors
MoVectorStack.h	stacks of three-dimensional vectors
MoFrame.h	kinetostatic state of a spatial reference frame
MoFrameList.h	lists of frames
MoMatrix.h	generic three-dimensional matrices
MoInertiaTensor.h	rigid-body inertia matrices
MoRotationMatrix.h	orthogonal matrices representing general rotations
MoXYZRotationMatrix	orthogonal matrices representing elementary rotations
MoOutputGenerator.h	printing of intermediate values

Table 1.1: Header files for the basic mathematical objects

module	functionality
MoRigidLink.h	rigid connections between reference frames
MoElementaryJoint.h	revolute or prismatic joint aligned with coordinate axis
MoElementaryScrewJoint.h	screw joint aligned with coordinate axis
MoCylindricalJoint.h	cylindrical joint aligned with coordinate axis
Mo3DTranslationalJoint.h	joint realizing general spatial translation
MoSphericalJoint.h	joint realizing spherical motion
MoFloatingBodyJoint.h	joint realizing general spatial motion
MoInstantaneousScrew.h	computation of instantaneous screws
MoMapChain.h	chains of kinetostatic transmission elements
MoConstantWrench.h	force element applying constant force and/or moment
MoLinearSpringDamper.h	linear spring-damper force element
MoMassElement.h	application of inertia properties of rigid body
MoConstantStepDriver.h	generation of constant velocity motion

Table 1.2: Header files for the basic kinetostatic transmission elements

module	functionality
<code>MoChord3DOrientation.h</code>	relative orientation between two frames
<code>MoChord3DPosition.h</code>	relative vector between two frame origins
<code>MoChord3DPose.h</code>	relative pose between two frames (union of both above)
<code>MoChordPlanePlane.h</code>	cosine of angle between two planes
<code>MoChordPlanePoint.h</code>	distance from a plane to a point
<code>MoChordPointPlane.h</code>	distance from a point to a plane
<code>MoChordPointPointLinear.h</code>	linear distance between two points
<code>MoChordPointPointQuadratic.h</code>	quadratic distance between two points
<code>MoChordList.h</code>	lists of elementary measurements
<code>MoConstraintSolver.h</code>	generic objects for the resolution of constraint equations
<code>MoExplicitConstraintSolver.h</code>	generator of closed-form solutions for scalar constraint
<code>MoImplicitConstraintSolver.h</code>	generator of iterative solutions for general constraints

Table 1.3: Header files for constraint generation and solution

module	functionality
<code>MoEqmBuilder.h</code>	generation of mechanical equations of motion
<code>MoDynamicSystem.h</code>	generation of space-state form of dynamical equations
<code>MoAdamsIntegrator.h</code>	numerical integration based on Adams-Bashford-Moulton method
<code>MoBDFIntegrator.h</code>	numerical integration based on Gear's BDF method
<code>MoExplicitEulerIntegrator.h</code>	integration based on Euler method
<code>MoRungeKuttaIntegrator.h</code>	numerical integration based on 4th order Runge-Kutta method
<code>MoStaticEquilibriumFinder.h</code>	computation of stationary point

Table 1.4: Header files for generation and solution of equations of motion

module	functionality
<code>Inventor/MoScene.h</code>	viewer and editor window for 3D motion animation
<code>Inventor/MoWidget.h</code>	slider and push button widgets

Table 1.5: Header files for animation

module	functionality
<code>MoBase.h</code>	all of Table 1.1
<code>MoBasicKTE.h</code>	all of Table 1.2
<code>MoConstraints.h</code>	all of Table 1.3
<code>MoDynamics.h</code>	all of Table 1.4
<code>Inventor/MoGraphics.h</code>	all of Table 1.5

Table 1.6: Container header files

2 Overview

This chapter is devised as an introduction to the capabilities of M□BILE. The reader will be guided through the process of modeling the dynamics of a simple example, starting from the basic topological structure and ending with the generation of an animation. The intention is to display the fundamental ideas underlying the M□BILE philosophy together with a description of the basic modeling steps. It is thus not necessary to understand all the underlying mechanisms at this point. The details are discussed in the subsequent chapters.

2.1 Structure of M□BILE

One of the main features of M□BILE is that it allows the user to model mechanical systems as executable programs that can be used as building blocks for other environments. This is achieved by representing each real-world component by a dedicated object that is capable of performing some well-defined set of actions upon request. The objects of M□BILE are roughly organized in three categories:

- (a) **basic mathematical objects**, which provide the algebraic resources for performing the typical multibody calculations,
- (b) **kinetostatic state objects**, which are used to store and retrieve kinematic or load-related information at specific locations of the multibody system
- (c) **kinetostatic transmission elements**, which transmit the information stored with the kinetostatic state objects from one location of the system to the other

Each transmission element supplies, in analogy to its real-world counterpart, two basic operations:

- (I) the **transmission of motion** and
- (II) the **transmission of forces**.

In M□BILE, these two operations are realized as virtual functions, “doMotion()” and “doForce()”, respectively, that are shared by all kinetostatic transmission elements.

Kinetostatic state objects serve as input and output variables for the various types of kinetostatic transmission elements. There exist two basic types of kinetostatic state objects:

- (a) **spatial kinetostatic state objects**, or reference frames, which can be imagined as interconnection junctures between pairs of kinetostatical transmission elements, and

- (b) **scalar kinetostatic state objects**, which represent actuator or sensor data used to drive the motors of the joints or to store scalar data extracted from the system by measurements.

The overall picture of the approach is illustrated in Fig. 2.1. Prior to system assembly, reference systems are “floating” in space and possess no mutual relationship. Scalar variables resemble “wires” waiting to be plugged into appropriate places of the kinetostatic transmission elements in order to generate the desired motion. After assembly, the reference systems become attached at specific points of the transmission elements, interconnecting them by pairs, while the scalar variables accomplish the task of inducing motion at selected joints of the system. The assembly of a mechanical system thus consists in connecting the inputs and outputs of the kinetostatic transmission elements in appropriate order such that the resulting chains resemble the original system.

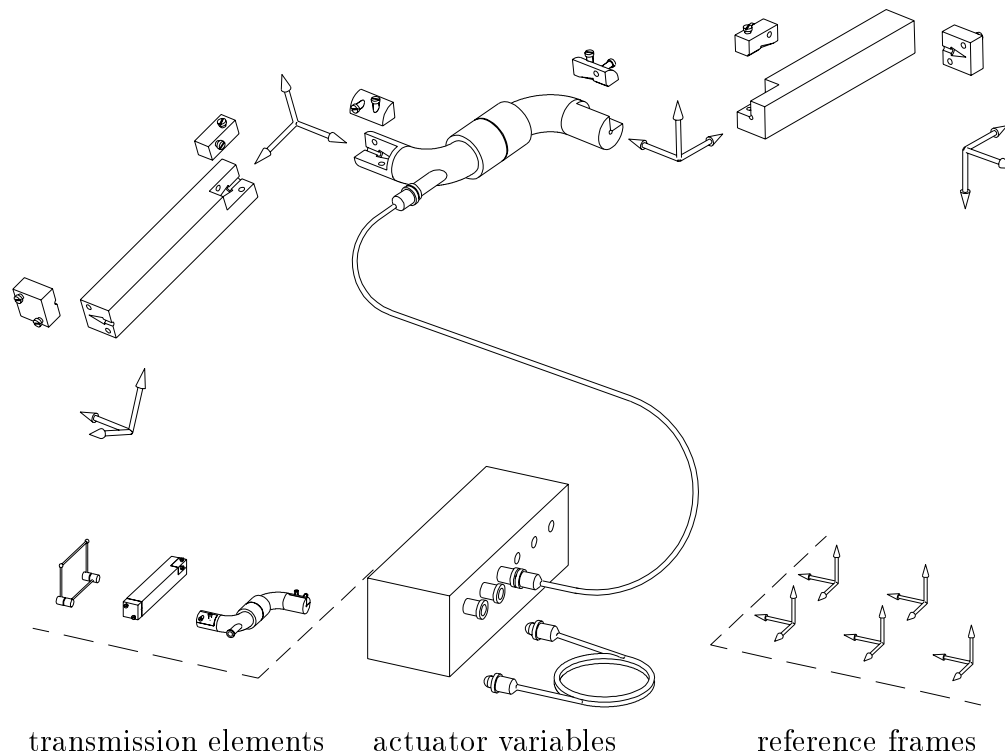


Figure 2.1: Objects in multibody systems

The modeling of mechanical systems by kinetostatic transmission elements mirrors the client-server paradigm of object-oriented programming. In this setting, objects represent individuals that are endowed with specific “responsibilities”. These responsibilities are chosen in such a way that the correct functioning of the overall society is warranted. However, the particular manner in which each object fulfills its responsibility is left as a matter of taste. In M□BILE, the responsibilities of the mechanical elements are to provide the aforementioned virtual transmission functions. For this functions, it does not matter *how* an object realizes its task. What matters is only *that* it does it.

2.2 Example: Analysis of a Simple Pendulum

The following analysis of a simple mathematical pendulum shall illustrate the basic steps involved in the modeling of a mechanical system with MOBILE. The objective is to generate the dynamical equations, then solve these, and finally animated the ensuing motion. All of this shall be accomplished by building a hierarchy of objects that provide more and more complex services by delegating sub-responsibilities to other, already existing objects.

2.2.1 Dissection and Re-Assembly of the System

The regarded system can be interpreted to consist of a massless link which can rotate about a fixed hinge at one end and to which a point mass is attached to the other end (see Fig. 2.2).

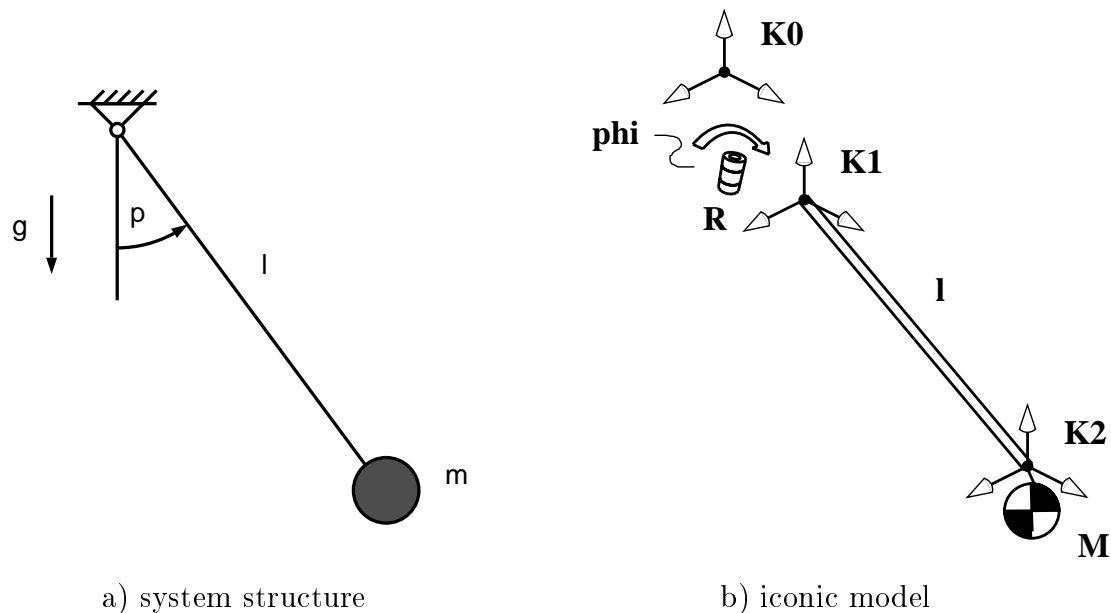


Figure 2.2: Modeling of a simple pendulum

In order to model the system, it is first necessary to dissect it into simple pieces. Such pieces can be those shipped with the MOBILE library or any other object defined by the user. The elementary objects of the MOBILE library are shown in Table 2.1

In the present example, the modeling is based on the following building blocks

- an elementary rotation about an axis,
- a translation within a rigid link, and
- a mass element attached to a particular location of the system.

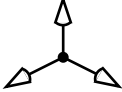








icon	M \square BILE-object
	MoFrame
	MoStateVariable
	MoRigidLink
	MoElementaryJoint (revolute)
	MoElementaryJoint (prismatic)
	MoSphericalJoint
	MoChord
	MoMassElement
	MoLinearSpringDamper

Table 2.1: Iconic representation of the elementary objects of M \square BILE

The M \square BILE modeling for the system consists in defining and assembling these pieces

```

MoFrame K0 , K1 , K2 ; // frames at the endpoints of transformations
MoAngularVariable phi ; // angular variable describing rotation
MoVector l ; // vector for displacement within the link
MoElementaryJoint R ( K0, K1, phi ) ; // object modeling the revolute
joint MoRigidLink rod ( K1, K2, l ) ; // object modeling the rigid
link MoReal m ; // scalar mass value MoMassElement
Tip ( K2, m ) ; // generates a point mass attached to K2
MoMapChain Pendulum ; // this object holds the concatenation ...
Pendulum << R << rod << Tip ; // ... of the previously defined elements

```

The basic constituents of this program are the objects “R”, “rod”, and “Tip” of type MoElementaryJoint, MoRigidLink and MoMassElement, respectively. The arguments passed to these objects correspond to their inputs and outputs. For example, K0 and K1 are the input and output frames of the revolute joint R, respectively, and phi is the corresponding rotation variable (among others the angle, as explained below). Accordingly, K1 and K2 are the input and output frames of the rigid connection rod, while l is the corresponding vector separating them. The mass element is modeled by a scalar value representing the mass attached to reference frame K2. The three pieces are assembled as a composite system termed “Pendulum” by making use of the shift operator “<<”.

Note that the name of the objects in this program is immaterial. Also, the sequence of definition of the objects is of no importance. It is only important to put them in correct

sequence into the composite chain. Moreover, the values of the components vectors and variables are not defined at this point. Only the topological structure is memorized during the definition of the objects. This is due to the fact that variables are passed “*by reference*” in the constructors of M□BILE. Thus, only *addresses* of the arguments are stored, in contrast to the “*pass by value*” technique, in which the actual value of the variables is employed. In M□BILE, values are re-read each time a motion or force traversal of the system is carried out during simulation. This gives a certain degree of symbolic capabilities to models established with M□BILE.

The mass property is defined as an additional transmission element (named “Tip”). At first sight, this seems redundant: why aren’t mass properties defined directly for the link? The background is that many mechanical systems can be modeled as massless skeletons for which mass-endowed parts occur only at discrete locations. In this case, a lot of redundant calculations would be carried out if these masses are set numerically to zero. For this reason, properties of motion and force transmission have been separated in M□BILE from inertia features. The user first models a massless scaffolding representing the overall interconnection structure of the system and attaches to it subsequently the mass elements at desired places.

2.2.2 Calculating Dynamic Properties

The code discussed above represents only a basic skeleton describing the kinematics and statics of the system. Based on this model, further computations can be performed. One example is the generation of the equation of motion, which is discussed next.

Theoretical background: Equation of motion for a one-degree-of-freedom system

The system at hand has only one degree of freedom, and no damping effects occur. Thus, the dynamics of the system are governed by the scalar equation of motion

$$m(\phi) \ddot{\phi} + b(\phi, \dot{\phi}) = Q(\phi) \quad .$$

Here, $m(\phi)$ is the *generalized mass*, $b(\phi, \dot{\phi})$ is the *generalized Coriolis and centrifugal force* and $Q(\phi)$ is the *generalized applied force* of the system.

Equations of motion are generated in M□BILE by objects of type `MoEqmBuilder`. These “builder” objects take a mechanical model represented by a kinetostatic transmission element and a set of variables acting as generalized coordinates, and compute the corresponding mass matrix and vectors of generalized Coriolis and applied forces from this information. Further arguments can be passed to the builder of equations of motion that describe the reference frame acting as the inertial frame and the upwards direction, i. e., the direction opposite to gravitation.

For the pendulum example derived above, the corresponding code has the following appearing:


```

#include <Mobile/MoBase.h>           // these are the header files ...
#include <Mobile/MoMapChain.h>       // ... containing definitions ...
#include <Mobile/MoElementaryJoint.h> // ... for the objects ...
#include <Mobile/MoRigidLink.h>      // ... used below ...
#include <Mobile/MoMassElement.h>    // ...
#include <Mobile/MoEqmBuilder.h>
void main () {

// define the mechanical system (see previous section)
MoFrame K0 , K1 , K2 ;
MoAngularVariable phi ;
MoVector l ;
MoElementaryJoint R ( K0, K1, phi ) ;
MoRigidLink rod ( K1, K2, l ) ;
MoReal m ;
MoMassElement Tip ( K2, m ) ;
MoMapChain Pendulum ;
Pendulum << R << rod << Tip ;

// create a list of generalized coordinates
MoVariableList vars ;
vars << phi ;

// create an object for generation of the equation of motion
MoEqmBuilder Dynamics ( vars , Pendulum , K0 , yAxis ) ;

// set the numerical values for the configuration to solve
l = MoVector ( 0 , -1 , 0 ) ;
m = 1 ;
phi.q = 0 ;

// carry out the analysis
for ( int i = 0 ; i++ < 18 ; phi.q += PI/18.0 ) {
    Dynamics.buildEquations() ;
    Dynamics.printMass() ;
    Dynamics.printForce() ;
}
}

```

Note that this is now an executable program. The object “Dynamics” is now capable of generating the equations of motion upon request. This occurs in the program by appending “.buildEquations()”, “.printMass()” and “.printForce()” to the object’s name. The invocations can be repeated as many times as required, without having to look again into the details of the once modeled object. Moreover, parameters, as well as variables of the transmission elements, such as *l* and *m*, and *phi*, can be basically treated as *symbols*, i.e., they can be assigned actual numerical values at arbitrary locations in the program.

The variable *phi* is a representative of a special set of objects in MOBILE termed “scalar kinetostatic state objects”. These objects comprise information about position, velocity, acceleration and force of a relative displacement. The subentries can be addressed by

appending “.q”, “.qd”, “.qdd” and “.Q” to the name of the scalar kinetostatic state object. Thus, the entry `phi.q` above addresses the *position* of the variable `phi`. This entry is for rotational variables an *angle*, which differs from the other scalar quantity, the *real number*, in that it contains also the sine and the cosine of the angle. This angle is first initialized to the value zero and then incremented within the loop in steps of in 18 steps of 10°.

Note that the object “Pendulum” does not come into play within the loop anymore. This object is now controlled by the object “Dynamics”. The object `Dynamics`, in turn, generates the equations of motion of its subordinate objects (here: `Pendulum`). In doing this, it acts as a (responsible, but lazy) master that invokes the corresponding transmission functions of its subordinates in order to accomplish the overall transmission behaviour for the complete system. Such a “delegation” of responsibility is typical for the object-oriented approach. M□BILE makes heavy use of responsibility delegation in the modeling of mechanical systems.

2.2.3 Automatic Integration of Dynamical Equations

Once generated, models of mechanical systems can be passed to further objects capable of performing numerical analysis. Examples hereof are eigenvalue analysis or the determination of the equilibrium configuration of the system. Below we reproduce a program in which the equation of motion of the pendulum is numerically integrated and the ensuing motion is animated via a realistic three-dimensional graphic model.

```
#include <Mobile/MoBase.h>
#include <Mobile/MoMapChain.h>
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoAdamsIntegrator.h>
#include <Mobile/Inventor/MoScene.h>

void main () {

// definition of mechanical system (see previous section)
MoFrame K0 , K1 , K2 ;
MoAngularVariable phi ;
MoVector l ;
MoElementaryJoint R ( K0, K1, phi ) ;
MoRigidLink      rod ( K1, K2, l ) ;
MoReal m ;
MoMassElement Tip ( K2, m ) ;
MoMapChain Pendulum ;
Pendulum << R << rod << Tip ;

// dynamic equation
MoVariableList vars ;
vars << phi ;
```

```

MoMechanicalSystem Dynamics ( vars , Pendulum , KO , yAxis ) ;

l      = MoVector ( 0 , -1 , 0 ) ;
m      = 1 ;
phi.q = phi.qd = 0 ;

// numerical integrator
MoAdamsIntegrator dynamicMotion ( Dynamics ) ;
MoReal dT = 0.1 ;
MoReal tol = 0.01 ;
dynamicMotion.setTimeInterval(dT) ;
dynamicMotion.setRelativeTolerance(tol) ;

// animation
MoScene Scene ( Pendulum ) ; // interface for 3D-rendering
Scene.makeManipulator ( R ) ; // create shape for revolute joint
Scene.makeShape ( R, rod ) ; // create shape for rigid link

Scene.addAnimationObject ( dynamicMotion ) ;
Scene.setAnimationIncrement ( 0.0 ) ; // animate as fast as possible

Scene.show() ;
MoScene::mainLoop() ; // move the scene
}

```

Note that the object “Dynamics” is now of type “MoMechanicalSystem”. This type is related to the type `MoEqmBuilder` described above, only that it maps the underlying dynamical equations to a system of first order differential equations suitable for numerical integration.

The definition of the graphical objects above employs the building blocks already employed in the previous mechanic modeling. The object “Scene”, of type `MoScene`, takes over the responsibility of rendering the animation for the user. It is instructed about which parts to render through the member function “makeShape”. In the example above, the geometry is supplied automatically by the scene object. However, it is also possible to supply user-defined geometries by specifying a corresponding data file as a second argument in `makeShape()`. The member function “makeManipulator” produces in addition to the graphical rendering a “manipulator” for the object (in this case the joint), through which the user can directly move the joint. On a Silicon Graphics workstation, this manipulation consists in dragging a cage around the joint. Fig. 2.3 shows the resulting graphics for the example above. The graphical rendering is hardware dependent. In `MOBILE 1.3`, the hardware supported are HP 9000 Series 700 Workstations and Silicon Graphics Indigo and Indy Workstations. The display in Fig. 2.3 stems from an SGI workstation.

The actual numerical simulation is carried out by an integrator of type Adams-Bashfort-Moulton. After setting step size and error tolerance, the integrator object can be treated as a kinetostatic transmission element that travels along the solution trajectory. Each time the motion transmission function “doMotion” is invoked, the system moves one small step along this trajectory. After each such step, the rendering function of the corresponding



Figure 2.3: Graphic representation of the pendulum example

graphical model of the pendulum is invoked, bringing eventually the motion of the system to the screen.

There are also other routines for integration installed in M \square BILE. Examples hereof are the explicit Euler method (`MoExplicitEulerIntegrator`) or the Runge-Kutta method (`MoRungeKuttaIntegrator`). In M \square BILE, numerical integration, as well as other computationally expensive numerical tasks, are solved using modules of standard numerical libraries. Currently, there exist interfaces for the numerical libraries SLATEC, NAG and IMSL. However, only the SLATEC routines are actually shipped with the M \square BILE software. The libraries NAG and IMSL are liable to licenses which have to be purchased by the user directly from the corresponding dealers. The user can freely choose between these methods and attach them to the mechanical models without having to regard the details of numerical algorithms.

2.3 Summary

The example discussed above displays some of the capabilities of the object-oriented multibody modeling library M \square BILE. It can be appreciated that objects of M \square BILE provide an intuitive and natural language for rapid prototyping of mechanical systems that is also well-suited for devising hand-tailored programs for simulating systems. Hand-tailored programs have the advantage of being open and easy to extend. Thus, once created, models can be extended as the demands grow. This is the key for efficient and integrated approaches featuring code reusal and interdisciplinary procedures, as pursued by this package.

3 Basic Mathematical Objects

This chapter describes the basic mathematical objects currently shipped the M□BILE package. The objects introduced here cover the typical mathematical entities encountered in the treatment of spatial kinematics and dynamics. These are

- **linear and angular scalars,**
- **three dimensional euclidean vectors,**
- **transformation matrices,**
- **inertia tensors,**
- **linear and angular variables, and**
- **spatial reference frames.**

Table 3.1 gives an overview of the classes for basic mathematical objects supplied with the M□BILE 1.3 software.

class	Functionality
MoAngle	Angles representing elements $\theta \in T^1$ (the one-dimensional Torus).
MoAngularVariable	Scalar state object storing motion and load state of a cyclic variable.
MoAngularVariableList	List of linear variables.
MoFrame	Spatial state object storing motion and load state of a reference frame.
MoFrameList	List of spatial reference frames.
MoInertiaTensor	Three dimensional inertia tensor.
MoLinearVariable	Scalar state object storing motion and load state of a linear variable.
MoLinearVariableList	List of angular variables.
MoMatrix	Generic base class for three dimensional matrix.
MoNullState	Object representing the universal neutral element.
MoReal	Floating point numbers $x \in \mathbb{R}$.
MoRealStack	Stack of floating point numbers.
MoRotationMatrix	Base class for three-dimensional orthogonal matrices.
MoStateVariable	Base class for scalar state objects.
MoVariableList	Heterogeneous list of scalar state objects.
MoVector	Three dimensional euclidean vector.
MoVectorStack	Stack of three dimensional euclidean vector.
MoXRotationMatrix	Elementary transformation matrix for rotation about the x -axis.
MoYRotationMatrix	Elementary transformation matrix for rotation about the y -axis.
MoZRotationMatrix	Elementary transformation matrix for rotation about the z -axis.
MoXYZRotationMatrix	Base class for elementary rotation transformation matrices.

Table 3.1: Overview of the basic mathematical objects of M□BILE

The above mentioned entities have been endowed with certain functions, operators and data structures that make it possible to use them in an intuitive and mathematically familiar way. When using the operators, care must be taken to regard the correct precedence of the latter. This precedence of operators is fixed by the C++ language. For convenience, the precedence of operators used in M□BILE is recollected in Table 3.2. Each block of

operators listed within two horizontal lines constitutes a group. The precedence of groups of operators is from top to bottom, and within each group operators are applied from left to right.

Most of the operators defined in MIBILE are in conformance to common usage in C++. However, some of these are applied quite differently. For example, the operator for the cross product of two vectors was chosen as the “modulus” operator “%”, which is the available operator most closely resembling the original mathematical symbol \times . Moreover, this operator has precedence over the + and - operators, so it conforms to common usage. However, the operator for dyadic product of two vectors, chosen as “^”, has a lower precedence than the additive operators. Thus, expressions such as $a \circ b + c \circ d$ have to be programmed with additional levels of parenthesis, i. e., as $(a \wedge b) + (c \wedge d)$ for correct expression evaluation.

[]	subscripting	<i>pointer[expr]</i>
()	function call	<i>expr(expr_list)</i>
~	complement	<i>~expr</i>
-	unary minus	<i>-expr</i>
+	unary plus	<i>+expr</i>
*	multiply	<i>expr*expr</i>
/	divide	<i>expr/expr</i>
%	modulo (remainder)	<i>expr%expr</i>
+	add (plus)	<i>expr+expr</i>
-	subtract	<i>expr-expr</i>
<<	shift left	<i>expr<<expr</i>
>>	shift right	<i>expr>>expr</i>
<	less than	<i>expr<expr</i>
<=	less than or equal	<i>expr<=expr</i>
>	greater than	<i>expr>expr</i>
>=	greater than or equal	<i>expr>=expr</i>
==	equal	<i>expr==expr</i>
!=	not equal	<i>expr!=expr</i>
&	bitwise AND	<i>expr&expr</i>
^	bitwise exclusive OR	<i>expr^expr</i>
	bitwise inclusive OR	<i>expr expr</i>
&&	logical AND	<i>expr&&expr</i>
	logical inclusive OR	<i>expr expr</i>
=	simple assignment	<i>lvalue=expr</i>
=	multiply and assign	<i>lvalue=expr</i>
/=	divide and assign	<i>lvalue/=expr</i>
%=	modulo and assign	<i>lvalue%=expr</i>
+=	add and assign	<i>lvalue+=expr</i>
-=	subtract and assign	<i>lvalue-=expr</i>
&=	AND and assign	<i>lvalue&=expr</i>
=	inclusive OR and assign	<i>lvalue =expr</i>
^=	exclusive OR and assign	<i>lvalue^=expr</i>

Table 3.2: Precedence of operators in MIBILE

3.1 The Universal Neutral Element MoNullState

The mathematical entities used in M□BILE stem from quite different algebraic spaces. In each of these spaces, there exists a unique “point” which represents some kind of “initial” or neutral state, which is called the *neutral point*. For example, the neutral point of a vector space is the origin or zero vector 0, while for transformation matrices it is the identity matrix.

In M□BILE, it is possible to reset any mathematical object to its value at the neutral element by assigning to it the universal neutral element “MoNullState”. The assignment operator returns again a reference to the object MoNullState, so it is possible to concatenate this resetting operation even when the objects at both sides of the equal signs are of different types. This allows one to reset a whole bunch of objects in only one line of code, as in:

```
#include <Mobile/MoBase.h>
main() {
    MoAngle beta ;
    MoZRotationMatrix Rot_z ;
    beta = Rot_z = MoNullState ;
}
```

Here, the angle `beta` and the matrix `Rot_z` are simultaneously reset to zero and the identity, respectively.

Due to efficiency issues, not all objects are automatically initialized in M□BILE when they are defined. The most volatile of them come into being with arbitrary values. Table 3.3 gives an overview of the objects for which no automatic initialization is performed at definition time. The table also exhibits the values that the objects will take upon as-

class	value generated by MoNullState
MoAngle	zero degrees
MoInertiaTensor	MoZeroMatrix
MoRotationMatrix	MoIdentityMatrix
MoVector	MoNullvector
MoXRotationMatrix	MoIdentityMatrix
MoYRotationMatrix	MoIdentityMatrix
MoZRotationMatrix	MoIdentityMatrix

Table 3.3: Objects that are not automatically initialized in M□BILE

signment of the universal neutral element. All other objects of M□BILE are initialized to a definite value at definition time. The corresponding initialization values are described in the M□BILE reference sheets.

3.2 The Basic Scalar Types MoReal and MoAngle

Scalar algebraic objects constitute the basis for the construction of mathematical expressions and of more involved composite elements. In the analysis of mechanical systems, there are two types of scalar numbers (Fig. 3.1):

- (a) **linear coordinates**, which are elements of the real line $x \in \mathbb{R}$, and
- (b) **cyclic coordinates**, which are elements on the real circle $\theta \in \mathbf{T}^1$.

In M□BILE, these two kinds of scalar algebraic objects are termed **MoReal** and **MoAngle**. The distinction between linear and cyclic scalar coordinates is appropriate for two reasons

- by introducing cyclic coordinates, one can avoid repeated evaluation of trigonometric expressions, thus reducing computational overhead,
- by introducing linear variables, unnecessary evaluation of trigonometric expressions is avoided and the compiler can also make case selections based on the type of motion (for example, recognition of prismatic and revolute joints based on the type of the actuation variable)

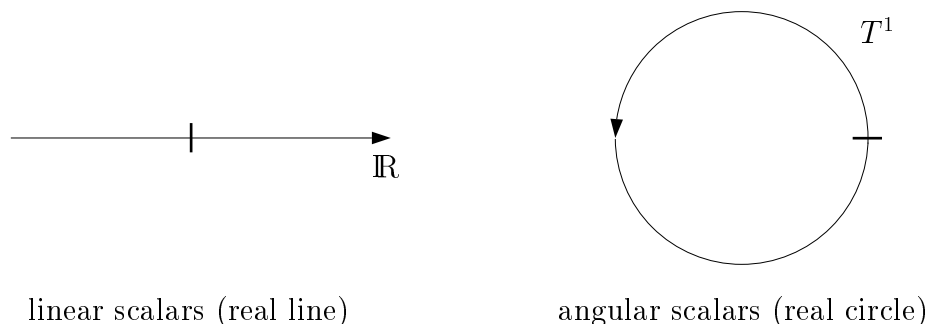


Figure 3.1: The two types of algebraic scalar objects in M□BILE

The type **MoReal** is just an alias of the native type **double** of C++. Thus, all operations defined for the double precision variables are also defined for the type **MoReal**.

The type **MoAngle** groups together the value of an angle, measured in *radians*, and its sine and cosine. The operations defined for this type are displayed in Table 3.4.

An example of the use of angles is shown below.

```
#include <Mobile/MoAngle.h>
main(){
MoAngle beta1=0 , beta2 , beta3 ; // beta1 is defined, beta2 and beta3 not!
beta2 = 30 * DEG_TO_RAD ;      // angle, sine and cosine of beta2 are set
```


Operator usage	Action
<code>angle = angle</code>	\mapsto <code>angle</code> assign an angle
<code>angle = real</code>	\mapsto <code>angle</code> assign a scalar (the value of the angle is in <i>radians</i>)
<code>angle + angle</code>	\mapsto <code>angle</code> add two angles
<code>angle - angle</code>	\mapsto <code>angle</code> subtract two angles
<code>- angle</code>	\mapsto <code>angle</code> change sign of an angle
<code>angle += angle</code>	\mapsto <code>angle</code> add and assign an angle
<code>angle += real</code>	\mapsto <code>angle</code> add and assign an angle (in <i>radians</i>)
<code>angle -= angle</code>	\mapsto <code>angle</code> subtract and assign an angle
<code>angle -= real</code>	\mapsto <code>angle</code> subtract and assign an angle (in <i>radians</i>)
<code>angle = MoNullState</code>	\mapsto <code>angle</code> assign zero angle
<code>angle . degrees()</code>	\mapsto <code>real</code> return value of angle in degrees
<code>angle . radians()</code>	\mapsto <code>real</code> return value of angle in radians
<code>angle . sine()</code>	\mapsto <code>real</code> return sine of angle (no computation)
<code>angle . cosine()</code>	\mapsto <code>real</code> return cosine of angle (no computation)

Table 3.4: Overview of operations for objects of type `MoAngle`

```

beta3 = beta1 - beta2 ;           // carry out an algebraic operation
beta3 += 10 * DEG_TO_RAD ;       // this also works
cout << beta3.degrees() << "\n"  // print the value of the angle, ...
    << beta3.sine() << "\n"     // its sine, ...
    << beta3.cosine() << "\n" ; // and its cosine to standard output.
cout << beta1 << "\n" ;         // prints [0,0,1] to standard output.
}

```

3.3 Vectors and Matrices

The methodologies for the analysis and synthesis of spatial kinematics and dynamics build substantially upon the notions of three-dimensional vectors and matrices. In `MOBILE`, three-dimensional vectors and matrices are thus given a particular attention. These objects are represented in `MOBILE` by the classes `MoVector` and `MoMatrix`. Operations concerning these classes have been designed such as to provide the user with an intuitive interface and with optimized code that take best advantage of the three-dimensional case.

3.3.1 Vectors

Table 3.5 shows the operations defined for vectors. By making use of these operators, not only one can write programs resembling common vectorial expressions, but it is also guaranteed that the operations are carried out in the most efficient way.

The components of each vector are represented by the member elements `x`, `y`, `z` of type `MoReal`. One accesses these components by appending ‘`.x`’, ‘`.y`’ or ‘`.z`’ to the name of the vector.

Operator usage	Action
<code>vector = vector</code>	\mapsto <code>vector</code> assign a vector
<code>-vector</code>	\mapsto <code>vector</code> change sign of vector
<code>vector + vector</code>	\mapsto <code>vector</code> add two vectors
<code>vector - vector</code>	\mapsto <code>vector</code> subtract two vectors
<code>vector * vector</code>	\mapsto <code>real</code> create inner product of two vectors
<code>scalar * vector</code>	\mapsto <code>vector</code> multiply vector by scalar
<code>vector % vector</code>	\mapsto <code>vector</code> create vector product (left times right)
<code>vector+=vector</code>	\mapsto <code>vector</code> add and assign a vector
<code>vector-=vector</code>	\mapsto <code>vector</code> subtract and assign a vector
<code>vector%=vector</code>	\mapsto <code>vector</code> vector product and assign (left times right)
<code>vector*=XYZrotation</code>	\mapsto <code>vector</code> transform by elementary rotation and assign
<code>vector^=XYZrotation</code>	\mapsto <code>vector</code> transform by transpose of elementary rotation and assign
<code>vector = MoNullState</code>	\mapsto <code>vector</code> assign zero vector

Table 3.5: Overview of operations for objects of type `MoVector`

The following program fragment shows some examples of the use of vectors.

```
#include <iostream.h>
#include <Mobile/MoVector.h>
main() {
MoVector v , u(0,0,1) , w = MoVector ( 1 , 1 , 0 ) ;
v = u + w ;
MoVector a = 3 * v ;
MoVector b = v % ( v % w ) + a % w + u ;
cout << b << " , " // prints to standard output: "(-4,2,3), "
    << a.y << " , " // prints to standard output: "3, "
    << u*w << "\n" ; // prints to standard output: "0<Newline>"
}
```

Note the different ways in which a vector can be defined. Note also the way operator precedence can be employed in order to reduce the number of parenthesis.

3.3.2 Matrices

Three-dimensional matrices come in different “flavors” in `M□BILE`: there are matrices representing rotations and matrices representing inertia properties. Similarly to the definition of angular and linear scalars, these two categories of objects, although they look alike, differ considerable in their inner structure: while rotational matrices are orthogonal, inertia matrices are always positive definite. Thus, in `M□BILE` special types of matrices are introduced that take account of matrix structure and allowed operations for each type.

Fig. 3.2 displays the functional hierarchy for the matrices defined in the `M□BILE` package. In this hierarchy, types to the right support all operations that the types to the left do. This hierarchy is only conceptual. It is not actually implemented in this manner. In particular, the data of the three elementary rotation matrices is structured in a different

manner than that of the other classes. Note that not all operations allowed for one type apply also to the other.

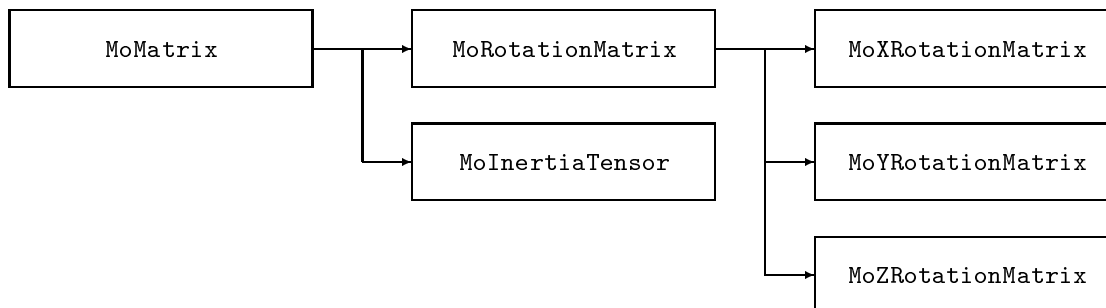


Figure 3.2: Class hierarchy for the different types of matrices in M□BILE

Table 3.6 displays the basic data structure of the different types of matrices supported by the M□BILE package. Furthermore, the 3×3 zero matrix `MoNullMatrix` and the 3×3 identity matrix `MoIdentityMatrix` are defined.

class	type of space	structure of matrix
<code>MoMatrix</code>	$A \in \mathbb{R}^3 \times \mathbb{R}^3$	$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$
<code>MoRotationMatrix</code>	$A \in \text{SO}(3)$	$A^T A = \mathbf{I}_3$
<code>MoXRotationMatrix</code>	$A \in \text{Rot}[x, \theta]$	$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}$
<code>MoYRotationMatrix</code>	$A \in \text{Rot}[y, \theta]$	$A = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$
<code>MoZRotationMatrix</code>	$A \in \text{Rot}[z, \theta]$	$A = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$
<code>MoInertiaTensor</code>	$A \in \{A: A \geq 0\}$	$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{22} & a_{33} \end{pmatrix}$

Table 3.6: Properties of the different types of three-dimensional matrices

The generic set of operations for matrices is summarized in Table 3.7. These operations are defined for all types of matrices. Specialized operations, which only make sense for a particular type of matrix, are discussed further below.

Operator usage	Action
$matrix = matrix \mapsto matrix$	assign a matrix.
$-matrix \mapsto matrix$	change of sign of a matrix.
$real * matrix \mapsto matrix$	scale a matrix.
$matrix * vector \mapsto vector$	multiply matrix times vector
$vector * matrix \mapsto vector$	multiply vector by transpose of matrix
$\sim matrix \mapsto matrix$	transpose a matrix.
$\sim vector \mapsto matrix$	generate skew-symmetric matrix from vector
$vector \hat{\ } vector \mapsto matrix$	generate dyadic product of two vectors

Table 3.7: General operations for objects of type `MoMatrix`

The two operators in Table 3.7 generating matrices from vectors are defined as

$$\begin{aligned} \sim vector : \quad & \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \mapsto \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \\ vector \hat{\ } vector : \quad & \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \otimes \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \mapsto \begin{pmatrix} a_x b_x & a_x b_y & a_x b_z \\ a_y b_x & a_y b_y & a_y b_z \\ a_z b_x & a_z b_y & a_z b_z \end{pmatrix} \end{aligned}$$

Note that the dyadic product operator used here corresponds to the bitwise exclusive OR operator in native C++. Thus, it has a very low precedence over the additive operators, and the user must enforce, by additional levels of parentheses, the correct order of evaluation.

The columns of matrices are accessible for types `MoMatrix`, `MoRotationMatrix` and `MoInertiaTensor` through three members of type `vector`. These members are denoted by `e1`, `e2` and `e3`. One can access the column vectors of the aforementioned matrices by appending `e1`, `e2` and `e3` to their name.

Another way of accessing the columns of a matrix for the types listed above is to use the parenthetical expression `matrix(index)`. Here, `index` is the `index` of the column to be returned. Allowed index values are 1, 2, 3.

Columns are not defined for objects of type `MoXRotationMatrix`, `MoYRotationMatrix` and `MoZRotationMatrix`. These represent elementary rotations about a coordinate axis in a compact and efficient manner. In fact, while behaving similarly to objects of class `MoRotationMatrix` at the global level, these objects store internally just the sine and the cosine of the corresponding rotation angle.

Table 3.8 shows the additional operations supported for inertia tensors. Note that the generic operations defined in Table 3.7 also apply to inertia tensors. Also, note that the product of two inertia tensors is not defined. This would not make any sense, as the resulting units would not be compatible with any mechanical quantity.

The special operations supported for rotation matrices are listed in Table 3.9. Note again

Operator usage		Action
$inertia + inertia$	$\mapsto inertia$	add two inertia matrices
$inertia += inertia$	$\mapsto inertia$	add and assign an inertia matrix
$inertia - inertia$	$\mapsto inertia$	subtract two inertia matrices
$inertia -= inertia$	$\mapsto inertia$	subtract and assign an inertia matrix
$inertia = MoNullState$	$\mapsto inertia$	assign zero matrix

Table 3.8: Special operations for objects of type `MoInertiaTensor`

that these are in addition to those of Table 3.7. Now the addition of rotation matrices is not supported, since it would destroy the property of orthogonality.

Operator usage		Action
$rotation * rotation$	$\mapsto rotation$	multiply two rotation matrices
$rotation *= rotation$	$\mapsto rotation$	multiply (left times right) and assign
$rotation = MoNullState$	$\mapsto rotation$	assign identity matrix
$rotation . normalize()$	$\mapsto rotation$	normalize non-orthogonal matrix

Table 3.9: Special operations for objects of type `MoRotationMatrix`

The matrix types `MoXRotationMatrix`, `MoYRotationMatrix` and `MoZRotationMatrix` support the same operations, plus the additional special operations listed in Table 3.10. Note that it is possible to assign a linear or angular scalar to an elementary rotation matrix. This sets the value of the rotation value, keeping the structure of the matrix fixed. Note also that it is only possible to assign elementary matrices *of the same type* to each other. For example, one can not “copy” an x -rotation to a y -rotation.

Operator usage		Action
$XYZrotation = angle$	$\mapsto XYZrotation$	assign angle of rotation
$XYZrotation = real$	$\mapsto XYZrotation$	assign angle of rotation in <i>radians</i>
$XYZrotation *= XYZrotation$	$\mapsto XYZrotation$	multiply and assign same type matrix
$XYZrotation = MoNullState$	$\mapsto XYZrotation$	assign identity matrix

Table 3.10: Special operations for objects of type `MoXRotationMatrix`, `MoYRotationMatrix` and `MoZRotationMatrix`

3.4 Kinetostatic State Objects

Kinetostatic state objects represent entities for storing motion and force information at different places in a mechanism. For the modeling of mechanical systems, two types of kinetostatic state objects are required: *scalar* kinetostatic state objects and *spatial* kinetostatic state objects. Spatial kinetostatic state objects embody the junctures between the mechanical components, while scalar kinetostatic state objects represent the “wires” that pass information about desired motion or forces to the joints of the mechanism.

Kinetostatic state objects enclose information about the motion and load state at a particular location of the mechanical system. The kinetostatic state consists of four basic items, termed “*kinetostatic state subentries*” (see also Fig. 3.3): position, velocity, acceleration and force.

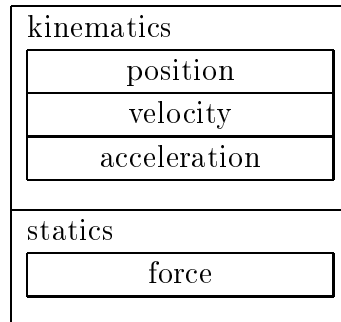


Figure 3.3: State subentries of state objects

Note thus that the notion of “kinetostatic state object” differs entirely from the term “state variables” known from system dynamics. In system dynamics, state variables are those appearing as first time derivatives in the state space form of the dynamical equations. Here, we denote as kinetostatic state objects the collection of position, velocity, acceleration and load information for a (scalar or spatial) variable in a mechanism.

State objects are designed as connectors that are placed between the kinetostatic transmission elements (Fig. 3.4). By the connector paradigm, it is made possible to access the information regarding kinematics and statics at any intermediate place of the multibody system. In this setting, state objects can be viewed as standardized, dual-ported RAMs allowing access both from the transmission elements and from the user. Hence, they allow the user to easily exchange or assemble transmission elements by making changes at one place independent of changes at another place. Note that the notion of kinetostatic state objects plays a similar role as the concept of nodes in the finite element method. There, the latter are introduced to define finite elements independently of one another.

The representation of state subentries differs from one type of kinetostatic state object to the other. In the following, the characteristics of the state subentries is discussed separately for scalar and spatial kinetostatic state objects.

3.4.1 Scalar Kinetostatic State Objects (“MoStateVariable”)

A scalar kinetostatic state object comprises the information about position, velocity, acceleration, and force related to one generalized coordinate (Fig. 3.5). This information is “pre-wired” within the various kinds of transmission elements, so the user does not need to bother about how to pass the different types of information correctly.

As discussed in Section 3.2, two types of displacements may arise in kinematics, namely angular and linear displacements. Accordingly, in MOBILE there exist two types of scalar kinetostatic state objects:

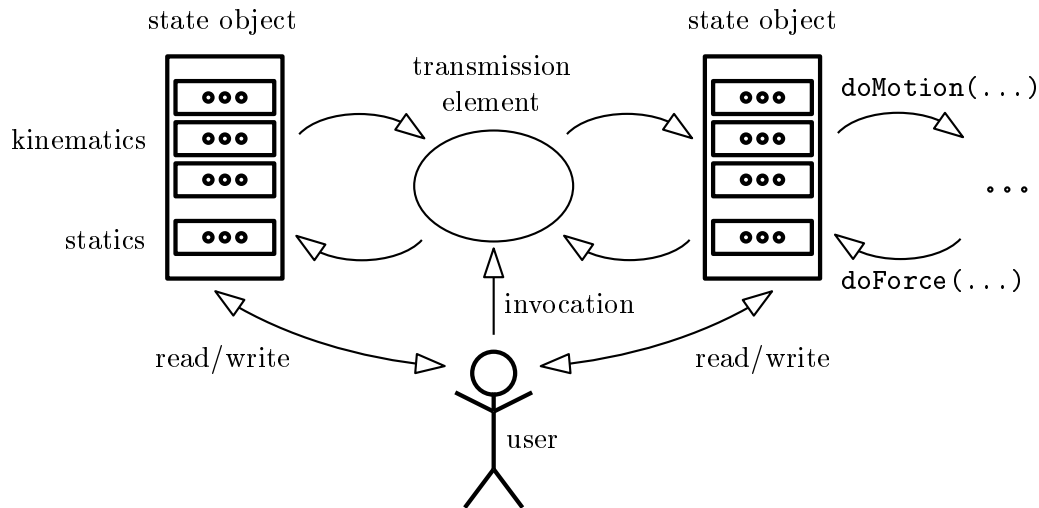


Figure 3.4: State objects as connectors between transmission elements

$$\{\beta\} = \begin{cases} \beta & \text{position} \\ \dot{\beta} & \text{velocity} \\ \ddot{\beta} & \text{acceleration} \\ Q_{\beta} & \text{generalized force} \end{cases}$$

Figure 3.5: Structure of a scalar variable (generalized coordinate).

- **angular variables** (class `MoAngularVariable`)
- **linear variables** (class `MoLinearVariable`)

These classes are derived from the generic class `MoStateVariable`, which collects their common properties. The class `MoStateVariable` is *abstract*, i.e., it does not allow for direct instantiation of objects. The only objects that can actually exist are those of type `MoLinearVariable` or `MoAngularVariable`. However, *pointers* to objects of type `MoStateVariable` are allowed. The use of these pointers is explained further below.

The position, velocity, acceleration and force subentries of scalar kinetostatic state objects are accessed by appending “.q”, “.qd”, “.qdd” or “.Q” to the variable name, respectively. The type of the corresponding state subentry is summarized in Table 3.11.

In some applications, a discerning between the two types of scalar kinetostatic state objects is of no importance, and one may wish to collect both types of kinetostatic state objects into one common list. For example, when generating the equations of motion or resolving constraint equations of complex mechanisms, it does not matter whether the input coordinates are linear or angular. In `MOBILE`, this can be done by introducing an array of pointers to their base class, `MoStateVariable`, and assigning to these pointers the values of the addresses of existing scalar kinetostatic state objects.

An example is the following code fragment in which the addresses of two scalar kinetostatic

state subentry	access of component	type of entry	
		angular case	linear case
position	<i>variable.q</i>	MoAngle	MoReal
velocity	<i>variable.qd</i>	MoReal	MoReal
acceleration	<i>variable.qdd</i>	MoReal	MoReal
force	<i>variable.Q</i>	MoReal	MoReal

Table 3.11: Subentries for scalar kinetostatic state objects of type “MoStateVariable”

state objects of different types, namely `beta` and `s`, are placed in the common array `vars`:

```
MoAngularVariable beta ;
MoLinearVariable s ;
MoStateVariable *vars[2] ;
vars[0] = &beta ;
vars[1] = &s ;
```

After declaring this array of pointers, one can access the state subentries of the concrete kinetostatic state objects through the pointer dereferencing mechanism. For example, the velocity, acceleration, and force components of both objects declared above can be accessed as follows:

```
vars[0]->qd = vars[1]->qd = 1.0 ; // velocity subentries
vars[0]->qdd = vars[1]->qdd = 2.0 ; // acceleration subentries
vars[0]->Q = vars[1]->Q = 3.0 ; // force subentries
```

For accessing the position, it is necessary to know the exact type of kinetostatic state object addressed by the pointer, as the type of this subentry depends on the type of the state variable. This information is supplied by the member function “`getType()`”. It returns an object of type `MoVariableType` which can take on two values:

- `PRISMATIC`, for linear variables, and
- `REVOLUTE`, for angular variables.

The use of this function is illustrated in the following code fragment

```
#include <Mobile/MoVariableList.h>

main() {

MoAngularVariable beta ;
MoLinearVariable s ;
MoStateVariable *vars[2] ;
vars[0] = &beta ;
```



```

vars[1] = &s      ;

beta.q = 90.0 * DEG_TO_RAD ;
s.q = 1.0 ;
for ( int i = 0 ; i < 2 ; i++ )
    switch ( vars[i]->getType() ) {
        case PRISMATIC:
            cout << ((MoLinearVariable*)vars[i])->q << "\n" ;
            break;
        case REVOLUTE:
            cout << ((MoAngularVariable*)vars[i])->q << "\n" ;
            break;
    }
}

```

The abstract pointer to `MoStateVariable` has been cast here to the correct type of concrete pointer before accessing the position subentry of the state variable.

Note that direct creation of objects of type `MoStateVariable` has been made impossible in `MOBILE` by declaring the constructor as “protected”. Thus, if the user mistakenly types something like this

```
MoStateVariable anyvar ; // error, constructor is private
```

a compiler error will be issued, because the constructor of the class `MoStateVariable` is not defined publicly.

A more elegant way of creating sets of heterogeneous scalar state variables is the use of *variable lists*. Variable lists are created in `MOBILE` as instances of class `MoVariableList`. Variable lists support operations for appending additional items to the list, moving forwards and backwards within the list, and jumping to the beginning or the end of the list. This functionality is realized through the operator “<<” and the functions “`getNext()`”, “`getPrevious()`”, “`rewind()`” and “`jumpToEnd()`”, respectively. Hereby, the lists of variables adapt their storage requirements automatically, so the user does not need to be concerned about implementation issues.

An example of the use of variable lists is the following code fragment. Here, the two scalar variables `beta` and `s` are collected in the variable list “`varlist`”, from where they are retrieved further below in the program in order to print the value of their position subentry.

```

MoAngularVariable beta ;
MoLinearVariable s ;
MoVariableList varlist ;
varlist << beta << s ;
beta.q = 90.0 * RAD_TO_DEG ;
s.q = 1.0 ;
MoStateVariable *p ;
varlist.rewind() ;
while ( p=varlist.getNext() ) // returns '0' at end of list

```

```

switch ( p->getType() ) {
  case PRISMATIC:
    cout << ((MoLinearVariable *)p)->q ;
    break;
  case REVOLUTE:
    cout << ((MoAngularVariable *)p)->q ;
    break;
}

```

It is also possible to access individual entries of variable lists directly through indexing. For convenience, two styles of indexing are supplied with the M_{OB}ILE software:

- **FORTRAN-style** indexing, using parenthesis “()”, and
- **C-style** indexing, using brackets “[].

With FORTRAN-style indexing, indices run from ‘1’ to the number of entries in the list. With C-style indexing, indices run from ‘0’ to the number of entries in the list minus one. For example, in the following program fragment, each line accesses exactly the same element of `varlist`:

```

varlist(2)->qd = varlist[1]->qd = 1.0 ; // velocity subentries
varlist(2)->qdd = varlist[1]->qdd = 2.0 ; // acceleration subentries
varlist(2)->Q = varlist[1]->Q = 3.0 ; // force subentries

```

3.4.2 Spatial Kinetostatic State Objects (“MoFrame”)

Spatial kinetostatic state objects store the motion and the load of a moving orthogonal frame. Spatial kinetostatic state objects are represented in M_{OB}ILE by objects of type “MoFrame”.

In M_{OB}ILE, motions of frames are defined with respect to an implicitly assumed inertial reference frame \mathcal{K}_0 , which is at rest (see Fig. 3.6). This single moving frame is termed below the *actual* frame, while the inertial frame is denoted as the *fixed* frame.

The motion of the actual frame comprises a rotational part and a translational part. The translational part is determined by the radius vector \mathbf{r} connecting the origin of the fixed frame with the origin of the actual frame, as well as the velocity \mathbf{v} and the acceleration \mathbf{a} of the origin of the moving frame with respect to the fixed frame. The rotational part consists of the orthogonal matrix \mathbf{R} representing the transformation of coordinates from the moving frame to the fixed frame as well as the angular velocity ω and angular acceleration $\dot{\omega}$ of the moving frame with respect to the fixed frame.

The load at the frame is described by a moment τ , measured with respect to the origin of the actual frame, and a force \mathbf{f} , both of which are assumed to be acting *from* the actual frame *onto* the structure spanned between the actual frame and the fixed frame.

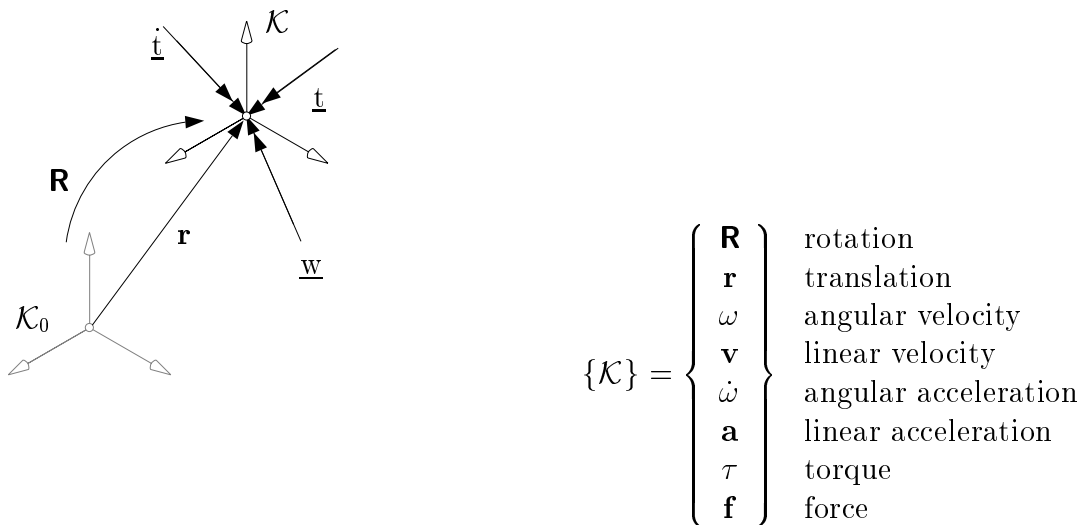


Figure 3.6: Components of a moving frame

In `MOBILE`, all vectors are usually assumed to be decomposed in the actual frame. Exceptions from this rule are stated explicitly.

The state subentries for spatial reference frames are accessed by appending the name of the correspondent member element to the name of the frame. The identifiers for the member elements are recollected in Table 3.12.

	state subentry	access of component	type of entry
position:	orientation	<code>frame.R</code>	<code>MoRotationMatrix</code>
	translation	<code>frame.r</code>	<code>MoVector</code>
velocity:	angular	<code>frame.ang_v</code>	<code>MoVector</code>
	linear	<code>frame.lin_v</code>	<code>MoVector</code>
acceleration:	angular	<code>frame.ang_a</code>	<code>MoVector</code>
	linear	<code>frame.lin_a</code>	<code>MoVector</code>
load:	moment	<code>frame.t</code>	<code>MoVector</code>
	force	<code>frame.f</code>	<code>MoVector</code>

Table 3.12: Subentries for kinetostatic state objects of type “`MoFrame`”

For example, if `K` is an object of type `MoFrame`, the corresponding location of the origin with respect to the fixed frame can be printed as

```
cout << K.R * K.r ;
```

Note that here the origin vector is premultiplied by the rotation matrix of the frame prior to being printed. Thus the printed components are measured with respect to the fixed

frame. Conversely, if one has a vector decomposed with respect to the fixed frame and wants to apply it the actual frame, one transforms it to the actual frame by

```
K.lin_v = vector * K.R ;
```

Note that, here, the vector is post-multiplied by the rotation matrix. This is equivalent to pre-multiplying the vector with the transpose of the matrix $K.R$ (see Section 3.3).

As it was the case with scalar state objects, it is possible to create logical units comprising several frames either by arrays or by lists. In MOBILE, lists of frames are created as instances of the class `MoFrameList`. Objects of this class are used in exactly the same manner as objects of type `MoVariableList` (see Section 3.4.1). Which of the two techniques the user prefers depends on the application. A code fragment in which both methods are employed is displayed below.

```
#include <Mobile/MoElementaryJoint.h>

main() {
// (a) arrays of frames
MoFrame K[10];
MoAngularVariable beta ;
MoElementaryJoint joint ( K[0] , K[1] , beta ) ; // access by indices

// (b) lists of frames
MoFrame K1 , K2 ;
MoFrameList frameList ;
frameList << K1 << K2 ;           // fill the list
MoFrame *p ;
while ( p=frameList.getNext() )   // returns '0' at the end of list
    cout << p->R * p->r << "\n" ; // global components of radius vectors
}
```

Spatial and scalar kinetostatic state objects are crucial for the concatenation of transmission elements. They can be regarded as the “glue” that holds the different parts together. Moreover, kinetostatic state objects provide the interfaces through which the user can access and manipulate the state-related data within the mechanical system. This gives to the user complete control and insight over the operations performed within the model. This will become more clear in the following chapter when the kinetostatical transmission elements are discussed.

4 Basic Kinetostatic Transmission Elements

This chapter describes the basic building blocks of M□BILE for the kinematic, static and inverse dynamics modeling of tree-type mechanical systems. Further objects, suitable for the resolution of constraint equations (i.e., for the treatment of closed loops), the generation of direct dynamics, and for the numerical solution of the equations of motion, will be discussed in subsequent chapters.

4.1 Overview of Supplied Kinetostatic Transmission Elements

The mechanical building blocks in M□BILE are particular specializations of a generic element denominated “MoMap”. This “super-ancestor” subsumes the generic properties of mechanical components in form of virtual functions that are guaranteed to be supplied by all mechanical components, whatever the details of their implementation. Two of these functions are “doMotion(...)” for the transmission of *motion* and “doForce(...)” for the transmission of *forces*.

Modules of mechanical components are obtained in M□BILE by derivation from the class MoMap. The idea behind this kind of module organization is that one can address the generic properties of a component without knowing the exact type of implementation hidden behind it. In M□BILE, one can access the generic properties of any mechanical component, e.g., a joint, a rigid link, or a complete vehicle suspension, by regarding them as instances of class MoMap.

Fig. 4.1 illustrates the class hierarchy for a part of the family of kinetostatic transmission elements. Note that some nodes act again as a base class for several descendants. Note also that some elements, like MoOutputGenerator, MoIntegrator and MoDriver, are not actually mechanical objects in the sense that they participate in the power transmission of the system. Nevertheless, they are treated in M□BILE as kinetostatic transmission elements in the sense that they carry out signal processing tasks resembling the power transmission mechanism of the kinetostatic model.

Table 4.1 gives an overview of the currently supplied kinetostatic transmission elements. Note that classes in M□BILE can be either *abstract*, or *concrete*. For example, the classes MoForceElement and MoChord are abstract, while the classes MoConstantStepDriver, MoChordPointPlane, and MoLinearSpringDamper are concrete. Abstract classes act merely as conceptual containers for the common properties of a family of elements. They can not be instantiated directly and their implementation is thus of no interest for the user. However, one can introduce *pointers* to this type of objects that can be employed for accessing the generic properties of objects derived from this class. On the other hand, concrete classes represent actual implementations of entities that can be instantiated and used as many times as necessary.

This chapter introduces the some fundamental classes needed for kinetostatic modeling of tree-type mechanical systems. Other elements will be discussed in the subsequent chapters. A detailed syntax description for these objects can be found in the appended

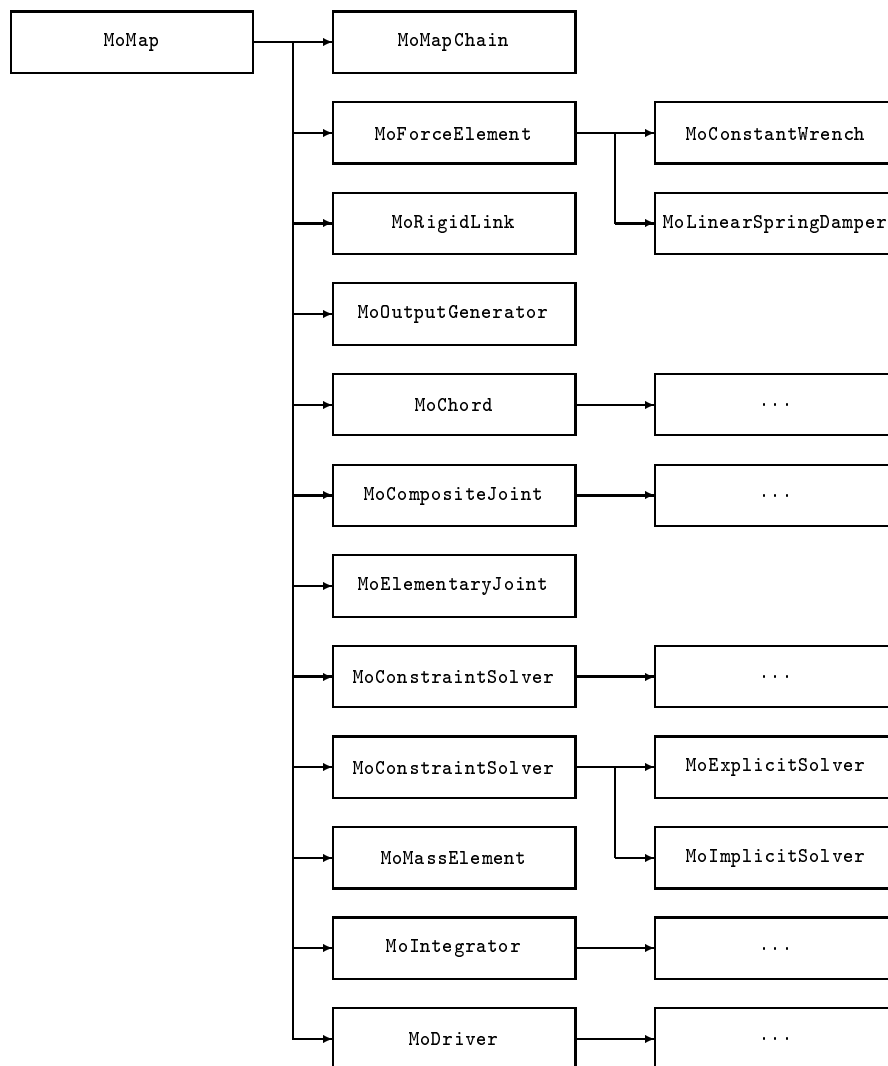


Figure 4.1: Hierarchy of kinetostatic transmission elements of M□BILE (excerpt)

Reference Sheets.

4.2 Generic Properties of Kinetostatic Transmission Elements

The notion of the kinetostatic transmission elements was already introduced in Section 2. This section focuses on the common features of the kinetostatic transmission elements, presenting their underlying generic model and shared “services”.

Class	Functionality	class type
MoMap	generic base class for kinetostatic transmission elements	abstract
MoElementaryJoint	rotational or prismatic joints	concrete
MoRigidLink	rigid link (binary or multiple)	concrete
MoElementaryScrewJoint	screw joint	concrete
MoCompositeJoint	base class for multi-degree-of-freedom joints	abstract
MoSphericalJoint	spherical joint (both Euler or Byrant angles)	concrete
Mo3DTranslationalJoint	pure three-dimensional translation	concrete
MoFloatingBodyJoint	general spatial motion	concrete
MoMapChain	concatenation of transmission elements	concrete
MoForceElement	the base class for objects modeling applied forces	abstract
MoConstantWrench	spatial force or moment at a reference frame	concrete
MoLinearSpringDamper	linear spring-damper element (for joints and chords)	concrete
MoChord	base class for geometric measurements	abstract
MoChordPointPointQuadratic	measurement of quadratic distance between two origins	concrete
MoChordPointPointLinear	measurement of linear distance between two origins	concrete
MoChordPointPlane	measurement of distance between origin and plane	concrete
MoChordList	list of chords	concrete
MoConstraintSolver	base class for solution of loop constraint equations	abstract
MoExplicitConstraintSolver	closed-form solution of scalar constraint equations	concrete
MoImplicitConstraintSolver	iterative solution of general constraint equations	concrete
MoMassElement	inertia properties of point masses and rigid bodies	concrete
MoIntegrator	base class for numerical integration	abstract
MoAdamsIntegrator	Adams-Moulton-Bashfort integration	concrete
MoExplicitEulerIntegrator	one-step explicit Euler integration	concrete
MoRungeKuttaIntegrator	4th order Runge-Kutta integration	concrete
MoDriver	base class for objects generating kinematical input	abstract
MoConstantStepDriver	generation of constant step increments for scalar variables	concrete
MoOutputGenerator	printout of scalars, vectors or matrices	concrete

Table 4.1: Selection of kinetostatic transmission elements in M□BILE

4.2.1 Model of a Kinetostatic Transmission Element

At the heart of the object-oriented modeling of M□BILE is the notion of the “kinetostatic transmission element”, an abstraction which describes the *minimal* basic services that any mechanical component must provide.

Consider a mechanical system consisting of several hundred of pieces ranging from bolts over joints to complete subsystems such as gear boxes, transmission elements, suspension mechanisms, etc. In real world, the designer will typically strive to use standardized, simple interfaces that make possible a rapid assembly and disassembly of the pieces. This makes the construction easily maintainable and extendable. In M□BILE, this idea is carried over to the modeling of multibody systems by employing as interfaces for the models not only the data, but also the functions. By this measure, the user does not need to peek into the core of the models to obtain information about internal implementation details, but can construct a global model by making only use of abstract services. Hence, a model can be completed incrementally by adding or removing parts without affecting

the rest of the operational model.

The transmission behaviour, or the “services”, of a kinetostatic transmission element can be best described by considering a simple element mapping a set of n scalar (kinematical) *input* variables $\underline{q} \in \mathbb{R}^n$, to a set of m scalar (kinematical) *output* variables $\underline{q}' \in \mathbb{R}^m$, as displayed in Fig. 4.2.

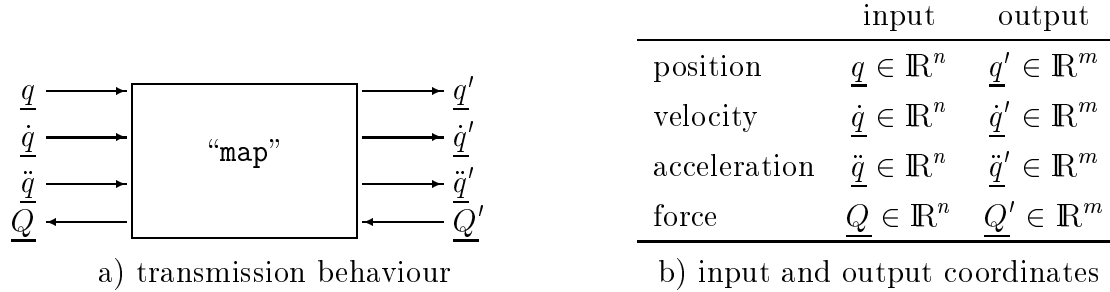


Figure 4.2: Model of a kinetostatic transmission element

The transmission of motion involves the mapping of the input variables \underline{q} to a corresponding set of output variables \underline{q}' . As well, the time derivatives $\underline{\dot{q}}$ and $\underline{\ddot{q}}$ of the input coordinates, i.e. the velocity and acceleration, are mapped to the corresponding time derivatives $\underline{\dot{q}'}$ and $\underline{\ddot{q}'}$ of the output coordinates. Apart from the kinematic transmission functions, the kinetostatic transmission element also induces a transmission of forces. This transmission is directed in opposite direction to the transmission of motion, mapping the forces \underline{Q}' at the (kinematic) output of the kinetostatic transmission element to corresponding forces \underline{Q} at the (kinematic) input. The reasoning behind this kind of modeling is explained in the following theoretical background information.

Theoretical background: Structure of the kinetostatic transmission functions

The transmission of motion is governed by the following formulas:

$$\begin{aligned}
 \text{position:} \quad \underline{q}' &= \underline{\varphi}(\underline{q}) \\
 \text{velocity:} \quad \underline{\dot{q}'} &= \mathbf{J}_\phi \underline{\dot{q}} \\
 \text{acceleration:} \quad \underline{\ddot{q}'} &= \mathbf{J}_\phi \underline{\ddot{q}} + \dot{\mathbf{J}}_\phi \underline{\dot{q}}
 \end{aligned}$$

where \mathbf{J}_ϕ represents the *Jacobian* of the transmission element

$$\mathbf{J}_\phi = \frac{\partial \phi}{\partial \underline{q}} \in \mathbb{R}^{m \times n} \quad . \quad (4.1)$$

For the transmission of forces, one first assumes that the transmission is *ideal*, i.e., that it neither generates nor consumes power. Then, equality of virtual work at the input and output holds, and it follows

$$\delta \underline{q}^T \underline{Q} = \delta \underline{q}'^T \underline{Q}' \quad .$$

After substituting $\delta \underline{q}' = \mathbf{J}_\phi \delta \underline{q}$ and noting that this condition must hold for all $\delta \underline{q} \in \mathbf{R}^n$, the *force transmission* function

$$\underline{Q} = \mathbf{J}_\phi^T \underline{Q}'$$

follows. If the kinetostatic transmission element is not ideal, it contributes to the ideal transmission of forces an additional term $\underline{\hat{Q}}$ which represents the *source* force of the element. Such source forces can result for example from friction effects, but can also model contributions due to applied forces or mass effects. With the source force $\underline{\hat{Q}}$, the transmission of forces takes on the form

$$\text{force:} \quad \underline{Q} = \mathbf{J}_\phi^T \underline{Q}' + \underline{\hat{Q}} . \quad (4.2)$$

Note that, in general, the Jacobian \mathbf{J}_ϕ is not quadratic, so for most transmission elements this relationship cannot be reversed. Thus, the *natural* direction of transmission of forces is in *opposite* direction to the motion mapping, as was already depicted in Fig. 4.2.

4.2.2 Invoking Motion and Force Transmission

The transmission of motion and force is implemented for each type of mechanical component in a particular (and efficient) manner. The user can invoke these functions either by appending the strings `.doMotion()` and `.doForce()` to the name of the kinetostatic transmission element, or by using the pointer dereferencing mechanism `pointer->doMotion()` and `pointer->doForce()`. Moreover, because the transmission operations are coded as virtual functions, the user can invoke these functions also by treating pointers to existing kinetostatic transmission elements as instances of the generic transmission element `MoMap`.

The following code fragment illustrates this concept. Two concrete kinetostatic transmission elements are introduced and subsequently the motion transmission function is invoked once directly for the given objects and once indirectly by treating them as generic transmission elements and then using the pointer dereferencing mechanism.

```
...
    MoAngularVariable beta ;
    MoVector l ;
    MoFrame K1, K2 , K3 ;
    MoElementaryJoint R ( K1 , K2 , beta ) ;
    MoRigidLink      L ( K2 , K3 , l ) ;
...
// direct invocation ...
R.doMotion() ;
L.doMotion() ;

// indirect invocation ...
MoMap *kinetostaticElements[2] ;
kinetostaticElements[0] = &R ;
kinetostaticElements[1] = &L ;
for ( int i = 0 ; i<2 ; i++ )
    kinetostaticElements[i]->doMotion( DO_ALL ) ;
```

Note that direct invocation leads to more compact code, while indirect invocation allows for automatized treatment of larger assemblies through iterative mechanisms. In general, the user is free to choose the most convenient form for the application at hand.

function	direct invocation	indirect invocation
motion	<code>object.doMotion (...)</code>	<code>pointer->doMotion (...)</code>
motion	<code>object.doForce (...)</code>	<code>pointer->doforce (...)</code>

Table 4.2: Types of motion and force invocation

The two mechanisms of transmission function invocation are summarized in Table 4.2. The ellipsis in the function call represents an optional argument by which the user can select only a subset of the operations involved in the computation of an invoked transmission function. These subtasks are described in the following section.

4.2.3 Selection of Motion and Force Transmission Subtasks

When invoking the motion and force transmission functions, one can pass an argument of type “MoTransmissionSubtask”. The effect of the transmission subtask selection parameter is to select particular terms which are to be calculated or added to the result during traversal of the transmission function.

The individual terms and calculation steps arising in the transmission of motion and force are illustrated in Fig. 4.3. The meaning of these terms is displayed in Table 4.3

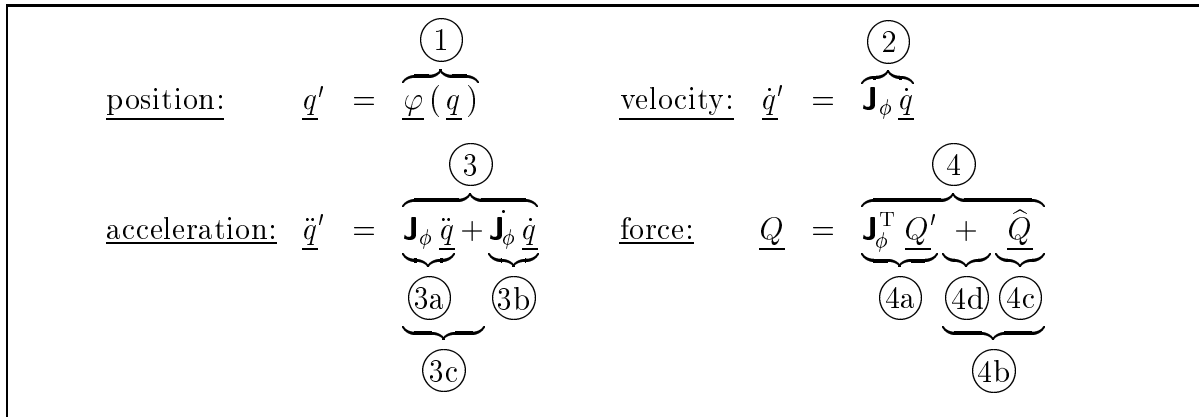


Figure 4.3: Terms and computational steps involved in the transmission of motion and forces

Table 4.4 lists the possible values of the transmission subtask selection parameter for motion transmission and their relationship to the terms identified above.

In the setting of spatial transmission elements, the term $(3a)$ corresponds to the transmission of the reference acceleration together with the relative acceleration due to the second time derivatives of the relative coordinates. This term is denoted the “EULER”

term	functionality
1	transmission of position (rotation/translation)
2	transmission of velocity (no further subtasks)
3	complete transmission of acceleration
3a	transmission of input acceleration term only
3b	computation of quadratic velocity terms only (no transmission)
3c	transmission of quadratic velocity terms only (no computation)
4	complete transmission of forces
4a	transmission of external forces only
4b	computation and transmission of internal forces
4c	computation of internal forces only (no transmission)
4d	transmission of internal forces only (no computation)

Table 4.3: Meaning of the terms in Fig. 4.3

term. The term $\textcircled{3b}$ represents the Coriolis, centripetal and gyroscopic accelerations stemming from quadratic velocity expressions. This expression is denoted the “CORIOLIS” term. When several kinetostatic transmission elements are concatenated, the resulting CORIOLIS terms are added together. In this case, the notation is a little misleading, as the transmission of CORIOLIS terms at the local level involves also the transmission of the reference acceleration stemming from CORIOLIS terms of the predecessor elements. However, from the global point of view the nomenclature is correct, as the calculated terms of the accelerations are only those involving relative velocities, and not relative accelerations.

The subtasks related to the forces are collected in Table 4.5. Note that here two types of source forces are discerned, namely, (i) *applied* source forces and (ii) *inertia* source forces. *Applied* source forces are those generated by springs, frictional effects, etc. Inertia source forces are those generated by mass elements. Setting the subtask selection parameter to “DO_INTERNAL” computes both of these forces. The distinction between the two types of forces stems from the MOBILE algorithm for the generation of dynamical equations, which involves calculating separately the mass matrix and the vector of applied forces. This is discussed in more detail in Chapter 6.

Similarly to the quadratic velocity terms in the acceleration, the internal forces can be computed only (“COMPUTE_INTERNAL”), or transmitted only (“TRANSMIT_INTERNAL”), or both computed and transmitted (“DO_INTERNAL”). This is done to optimize computations when the evaluation of force terms of the motion state is very time consuming.

In most cases, the selection of the appropriate subtasks is performed by the transmission elements automatically, so the user does not need to be concerned about choosing the correct values for the transmission subtask selection parameter. However, direct use of the transmission subtask selection parameter gives to the user better control over the scope of the computations, and by this a means of optimizing performance.

Note that one can combine different subtasks by using the logical ‘OR’ operator ‘|’ of C++.

name of constant	value	meaning	term
DO_NOTHING	0x00000	void action	
DO_POSITION	0x00003	transmit rotational and translational motion	①
DO_TRANSFORMATION	0x00001	transmit rotational motion only	①
DO_TRANSLATION	0x00002	transmit translational motion only	①
DO_VELOCITY	0x00004	transmit velocity	②
DO_ACCELERATION	0x00038	compute <i>and</i> transmit all acceleration terms	③
DO_EULER	0x00008	compute and transmit Euler acceleration term only	③a
COMPUTE_CORIOLIS	0x00010	compute quadratic acceleration term only (no transmission)	③b
USE_CORIOLIS	0x00020	transmit pre-computed quadratic acceleration term (no computation)	③c
DO_CORIOLIS	0x00030	compute and transmit quadratic acceleration terms (both of above)	③b/c
DO_ALL	0xFFFFF	do all of the actions defined above	

Table 4.4: Possible values for the motion subtask selection parameter

name of constant	value	meaning	
DO_NOTHING	0x00000	void action	
DO_EXTERNAL	0x00001	transmit external forces	④a
DO_INTERNAL	0x00006	compute and transmit all source forces	④b
COMPUTE_INTERNAL	0x00002	compute and store source forces (no transmission)	④c
USE_INTERNAL	0x00004	transmit pre-computed source forces (no computation)	④d
DO_INERTIA	0x00008	apply inertia-related source forces (applicable only to mass elements)	
DO_ALL	0xFFFFF	do all of the above	

Table 4.5: Possible values for the force subtask selection parameter

For example, a combination of position and velocity traversal is achieved by

```
object.doMotion ( DO_POSITION | DO_VELOCITY ) ;
```

Theoretical background: Calculation of Jacobians

In MOCBILE 1.3, all calculations are based on a Jacobian-free formulation. This simplifies considerably the implementation of new classes. However, for some applications, it may be required to determine the Jacobian \mathbf{J}_ϕ of a general mapping. In the following, two different methods are proposed for the calculation of Jacobians, which make only use of the motion and force transmission functions.

(A) Velocity-based determination of Jacobians (column-wise evaluation)

Setting at the input of the transmission element all velocity components equal to zero besides the j th one, which is set to $\dot{q}_j = 1$, yields an output velocity vector which is identical to the j th column of the Jacobian:

$$[\mathbf{J}_\phi]_j = \underline{\dot{q}}' \Big|_{\dot{q}_i = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}} \quad , \quad (4.3)$$

where $[\mathbf{J}_\phi]_j$ denotes the j th column of \mathbf{J}_ϕ . An equivalent approach is to use the acceleration transmission with the transmission subtask selection parameter set to DO_EULER.

(B) Force-based determination of Jacobians (row-wise)

A second method of computing the entries of the Jacobians is to set at the output of the transmission element all force components equal to zero besides the j th one, which is set to $Q'_j = 1$. Transmission of forces then yields at the input of the transmission element a force vector which is identical to the j th column of the *transposed* Jacobian, thus of the j th *row*

$$j\text{th row}(\mathbf{J}_\phi) = \underline{Q} \Big|_{Q'_i = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}} \quad . \quad (4.4)$$

Elimination of the influence of applied forces is achieved by setting the force transmission subtask selection parameter to DO_EXTERNAL. The advantage of this method is that it is possible to determine only selected rows of the Jacobian and thus the derivatives of only some specific output variables with respect to all input variables. This situation typically arises when long chains of transmission elements undergo only a few constraint conditions.

Note that the procedures described above are computationally not very efficient, but, due to their simplicity, they are well suited for rapid, draft-style modeling of multibody systems. More efficient procedures can be stated using sparse-matrix techniques.

4.3 Basic Transmission Elements: Links, Joints and Chains

This section describes the basic building blocks for the modeling of open kinematic chains. These elements are:

- rigid links
- elementary joints
- transmission chains
- force and mass elements

Some theoretical remarks are interspersed with the description. These formulae are only informative and need not be regarded by the casual user.

4.3.1 The Object “MoRigidLink”

A rigid link can be thought of as a transmission element that transports a coordinate frame \mathcal{K} via a constant rotation $\Delta\mathbf{R}$ and a constant displacement $\Delta\mathbf{r}$ to another frame \mathcal{K}' (Fig. 4.4). Here, $\Delta\mathbf{R}$ represents the orthogonal matrix transforming coordinates with respect to \mathcal{K}' to coordinates with respect to \mathcal{K} , and $\Delta\mathbf{r}$ is the vector from the origin of \mathcal{K} to the origin of \mathcal{K}' , decomposed in \mathcal{K}' .

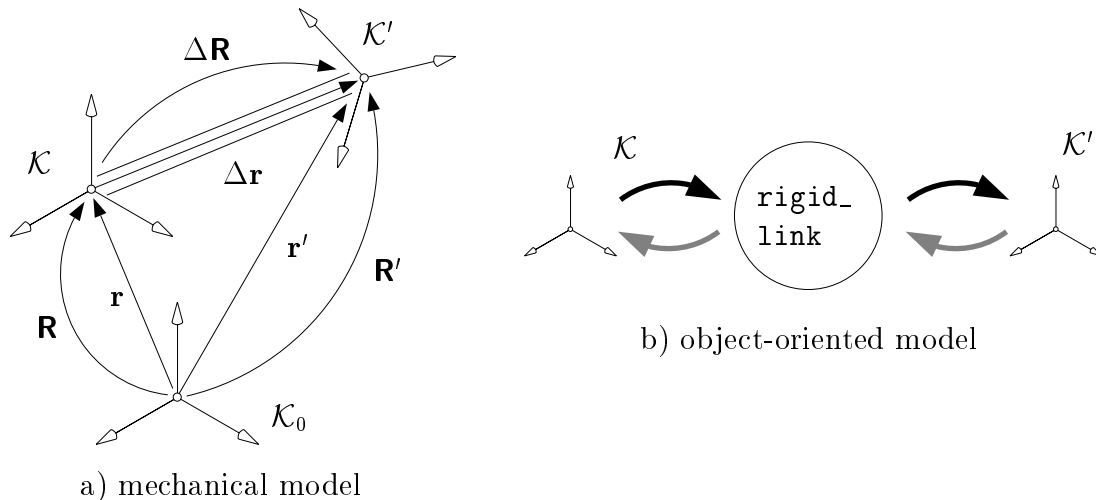


Figure 4.4: Model of a rigid link.

Rigid links are modeled in MOBILE as instances of class `MoRigidLink`. The typical definition of the rigid link consists in stating the input frame, the output frame and the relative displacement and rotation matrices as arguments to the rigid-link object. An example is

```
MoFrame K_Input , K_Output ;
MoVector Delta_r ;
MoRotationMatrix Delta_R ;
MoRigidLink Link ( K_Input , K_Output , Delta_R , Delta_r ) ;
```

Here, “Link” is the name of the newly introduced object representing the rigid link, and “K_Input” and “K_Output” are the input and output frames, respectively. The values of the relative displacement “Delta_r” and relative rotation “Delta_R” need not be specified at this point. The object rigid link “remembers” their location in storage space for later values storage and retrieval. Thus, the user can even change these values during simulation. However, no velocity and acceleration information is regarded for the relative displacement and orientation. Thus this technique works only for quasi-static variation of parameters.

The relative rotation is defined as the transformation matrix from components of the output frame to those of the input frame. Concerning the relative displacement, there is a slight subtlety to be regarded. In the form displayed above, the vector `Delta_r` is

assumed to be decomposed with respect to the *output* frame. However, it is also possible to specify the vector with respect to the input frame via the declaration

```
MoRigidLink Link ( K_Input , K_Output , Delta_r , Delta_R ) ;
```

Note that now the order of appearance of `Delta_R` and `Delta_r` is reversed.

If no relative displacement or no relative rotation occurs within the rigid link, one can omit the corresponding arguments, yielding optimized code for pure fixed-point rotation or pure translation. For example, the following code introduces a pure-rotation and a pure-translation link, respectively:

```
MoFrame K_Input1 , K_Output1 ;
MoRotationMatrix Delta_R ;
MoRigidLink PureRotation ( K_Input1 , K_Output1 , Delta_R ) ;

MoFrame K_Input2 , K_Output2 ;
MoVector Delta_r ;
MoRigidLink PureTranslation ( K_Input2 , K_Output2 , Delta_r ) ;
```

Note that for the pure translation it does not matter with respect to which of the two frames `K_Input` and `K_Output` the relative displacement is decomposed.

The class `MoRigidLink` can also be employed for defining *nary* or *multiple* rigid links, i. e., rigid links comprising n reference frames. These links have one input reference frame, and $n - 1$ output reference frames, which are collected in a reference frame list of type `MoFrameList`. For each output reference frame, a relative transformation matrix as well as a relative displacement vector can be defined, which are collected in corresponding arrays. Again, displacement vectors can be defined with respect to the respective output frame or with respect to the input frame by placing the corresponding array after or before the array of rotation matrices, respectively. In the same way, pure rotation or pure translation can be defined by leaving out the relative translations or relative rotations array, respectively. However, it is only possible to make this selections equally for all output frames. Hence, if for example one output frame is only rotated and another only translated with respect to the input frame, this can be taken advantage of only by introducing two separate rigid link objects. The following example defines a binary link with its output frames located at the tips of a two links swivelled by $\pm 45^\circ$ with respect to the input frame.

```
MoFrame K_Input , K_Output1 , K_Output2 ;
MoRotationMatrix Delta_R[2] ;
MoVector          Delta_r[2] ;
MoFrameList       K_Outs ;
K_Outs << K_Output1 << K_Output2 ;
MoRigidLink binary ( K_Input1 , K_Outs , Delta_r , Delta_R ) ;
Delta_R[0] = MoZRotation ( 45.0 * DEG_TO_RAD ) ;
Delta_R[1] = MoZRotation ( -45.0 * DEG_TO_RAD ) ;
Delta_r[0] = (1/sqrt(2)) * MoVector ( 1 , 1 , 0 ) ;
Delta_r[1] = (1/sqrt(2)) * MoVector ( 1 , -1 , 0 ) ;
```

Theoretical background: Rigid Link

The transmission functions for the rigid link are defined as (quantities of frame \mathcal{K}' are marked with a prime):

Position: (“forwards”)

$$\left. \begin{aligned} \mathbf{R}' &= \mathbf{R} \cdot \Delta\mathbf{R} \\ \mathbf{r}' &= \Delta\mathbf{R}^T \mathbf{r} + \Delta\mathbf{r} \end{aligned} \right\}; \quad (4.5)$$

Velocity: (“forwards”)

$$\begin{bmatrix} \omega' \\ \mathbf{v}' \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{R}^T & 0 \\ -\widetilde{\Delta\mathbf{r}} \Delta\mathbf{R}^T & \Delta\mathbf{R}^T \end{bmatrix} \begin{bmatrix} \omega \\ \mathbf{v} \end{bmatrix}; \quad (4.6)$$

Acceleration: (“forwards”)

$$\begin{aligned} \begin{bmatrix} \dot{\omega}' \\ \mathbf{a}' \end{bmatrix} &= \begin{bmatrix} \Delta\mathbf{R}^T & 0 \\ -\widetilde{\Delta\mathbf{r}} \Delta\mathbf{R}^T & \Delta\mathbf{R}^T \end{bmatrix} \begin{bmatrix} \dot{\omega} \\ \mathbf{a} \end{bmatrix} + \begin{bmatrix} 0 \\ \epsilon_a \end{bmatrix}, \\ \epsilon_a &= \omega' \times (\omega' \times \Delta\mathbf{r}); \end{aligned} \quad (4.7)$$

Force: (“backwards”)

$$\begin{bmatrix} \tau \\ \mathbf{f} \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{R} & \Delta\mathbf{R} \widetilde{\Delta\mathbf{r}} \\ 0 & \Delta\mathbf{R} \end{bmatrix} \begin{bmatrix} \tau' \\ \mathbf{f}' \end{bmatrix}. \quad (4.8)$$

4.3.2 The Object “MoElementaryJoint”

Elementary transformations, i. e., rotations about a coordinate axis or translations along a coordinate axis, play a fundamental role in the modeling of mechanical systems. In MOBILE, such elementary transformations are termed *elementary joints* and are realized by objects of type `MoElementaryJoint`. They form the basis for revolute and prismatic joints. Combinations of both are not regarded as elementary joints. Instead, two additional types of joints are introduced for this purpose. These are the `MoElementaryScrewJoint` for a combination of translation and rotation with a constant pitch (i. e., coupling) between these two, and `MoCylindricalJoint` for a combination of independent translation and rotation with respect to the same axis.

As a transmission element, the elementary joint maps the motion of an input reference frame \mathcal{K} and the value of the joint variable β to the motion of an output frame \mathcal{K}' (Fig. 4.5). Depending on whether the joint variable is an angle $\beta \equiv \Theta$ or a displacement $\beta \equiv s$, the joint becomes revolute or prismatic. For the case of a cylindric or screw joint, both types of variables are active. The joint axis is assumed to correspond to one of the coordinate axes of \mathcal{K} .

For the definition of a joint in MOBILE, one specifies the reference frames at the input and output, the joint variable, and optionally an argument indicating which coordinate axis to use for the transformation. The type of joint, i. e. whether it is prismatic or revolute, is recognized by the type of the variable passed as an argument. For example, the following code fragment initializes a prismatic joint

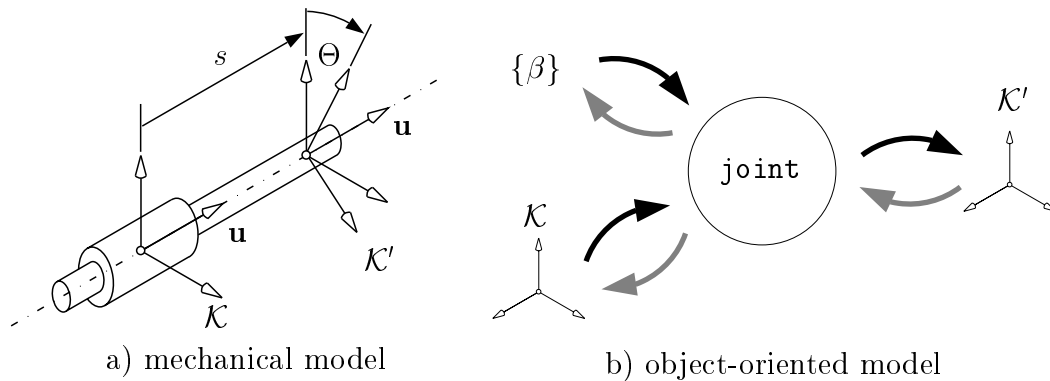


Figure 4.5: Model of an elementary joint.

```

MoFrame K_Input , K_Output ;
MoLinearVariable s ;
MoElementaryJoint Slider ( K_Input , K_Output , s ) ; // prismatic joint

```

while the next one produces a revolute joint

```

MoFrame K_Input , K_Output ;
MoAngularVariable Theta ;
MoElementaryJoint Hinge ( K_Input , K_Output , Theta ) ; // rotational joint

```

Note that the type of joint, i. e., prismatic or revolute, is selected automatically according to the type of state variable passed: in the first case, it was a linear variable, while in the second it was an angular one.

The axis of rotation or translation of elementary joints is implicitly assumed to be the \mathbf{z} -axis. One can overwrite this setting by specifying explicitly the axis through a fourth parameter. Possible values for this parameter are `xAxis`, `yAxis` and `zAxis`. For example, a rotational joint `R` rotating about the y -axis and a prismatic joint `P` translating along the x -axis are specified by

```

MoFrame K1 , K2 , K3 ;
MoAngularVariable Theta ;
MoLinearVariable s ;
MoElementaryJoint R ( K1 , K2 , Theta , yAxis ) ;
MoElementaryJoint P ( K2 , K3 , s , xAxis ) ;

```

Theoretical background: Transmission functions for elementary joints

Below the transmission equations are reproduced for an elementary cylindric joint. The equations can be specialized to the case of a revolute or prismatic joint by setting $s \equiv 0$ or $\Theta \equiv 0$, respectively. The axis is described by a unit vector $\mathbf{u} \in \{\mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_z\}$.

Position: (“forwards”)

$$\left. \begin{aligned} \mathbf{R}' &= \mathbf{R} \cdot \Delta \mathbf{R} ; \quad \Delta \mathbf{R} = \text{Rot}[\mathbf{u}, \Theta] \\ \mathbf{r}' &= \Delta \mathbf{R}^T \mathbf{r} + \mathbf{u} s \end{aligned} \right\} ; \quad (4.9)$$

Velocity: (“forwards”)

$$\begin{bmatrix} \omega' \\ \mathbf{v}' \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{R}^T & 0 & \mathbf{u} & 0 \\ -\widetilde{\Delta\mathbf{r}} \Delta\mathbf{R}^T & \Delta\mathbf{R}^T & 0 & \mathbf{u} \end{bmatrix} \begin{bmatrix} \omega \\ \mathbf{v} \\ \dot{\Theta} \\ \dot{s} \end{bmatrix}; \quad (4.10)$$

Acceleration: (“forwards”)

$$\begin{bmatrix} \dot{\omega}' \\ \mathbf{a}' \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{R}^T & 0 & \mathbf{u} & 0 \\ -\widetilde{\Delta\mathbf{r}} \Delta\mathbf{R}^T & \Delta\mathbf{R}^T & 0 & \mathbf{u} \end{bmatrix} \begin{bmatrix} \dot{\omega} \\ \mathbf{a} \\ \ddot{\Theta} \\ \ddot{s} \end{bmatrix} + \begin{bmatrix} \epsilon_\omega \\ \epsilon_a \end{bmatrix}, \quad (4.11)$$

$$\begin{aligned} \epsilon_\omega &= \omega' \times \mathbf{u} \dot{\Theta}, \\ \epsilon_a &= (\Delta\mathbf{R}^T \omega) \times (s \omega' \times \mathbf{u} + 2 \dot{s} \mathbf{u}); \end{aligned}$$

Force: (“backwards”)

$$\begin{bmatrix} \tau \\ \mathbf{f} \\ Q_\Theta \\ Q_s \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{R} & \Delta\mathbf{R} \widetilde{\Delta\mathbf{r}} \\ 0 & \Delta\mathbf{R} \\ \mathbf{u}^T & 0 \\ 0 & \mathbf{u}^T \end{bmatrix} \begin{bmatrix} \tau' \\ \mathbf{f}' \end{bmatrix}. \quad (4.12)$$

In Eq. (4.9), $\text{Rot}[\mathbf{u}, \Theta]$ designates the orthogonal matrix corresponding to a rotation about the axis with direction \mathbf{u} by an angle Θ ,

$$\text{Rot}[\mathbf{u}, \Theta] \equiv \mathbf{I}_3 + \sin\Theta \widetilde{\mathbf{u}} + (1 - \cos\Theta) \widetilde{\mathbf{u}}^2. \quad (4.13)$$

4.3.3 The Object “MoMapChain”

Mechanical systems usually do not consist of only one part, but result typically from the assembly of several components. In MOBILE, such composite systems are modeled as *chains of kinetostatic transmission elements*, for which a new type “MoMapChain”, is introduced. Because the overall transmission behaviour of a chain of transmission elements just consists of carrying out the transmission operations of the individual transmission elements, chains of kinetostatic transmission elements are simply established by concatenation.

This concatenation is achieved in MOBILE by the user with the shift operator “<<”, which appends the kinetostatic element to the list to the left. For example, the following code fragment defines the object “SimplePendulum” as a concatenation of a revolute joint “R” and a rigid link “L”:

```
MoFrame K0 , K1 , K2 ;
MoAngularVariable beta1 ;
MoVector l1 ;
MoElementaryJoint R1 ( K0 , K1 , beta1 , xAxis ) ;
MoRigidLink L1 ( K1 , K2 , l1 ) ;
MoMapChain SimplePendulum ;
SimplePendulum << R1 << L1 ;
```

Note that `MOBILE` performs no sorting of the kinetostatic elements supplied to the `MoMapChain`. Hence *the order of elements in the concatenation sequence is significant*. Each object must be placed at such a position in the chain that any objects providing values for its inputs are located to the left of this object. As a rule of thumb, objects that correspond to mechanical components arranged along serial branches must be concatenated in exactly the same order as in the real system, starting from the inertial system. For parallel branches one can mix up the objects from the different branches, but it must be ensured that the subset of objects belonging to the same branch appear in correct order relative to one another. For example, linking the objects above as `SimplePendulum<<L1<<R1` would have resulted in erroneous calculations (although no error message would have been issued).

Objects of type “`MoMapChain`” are again instances of kinetostatic transmission elements. This means that chains of transmission elements can be used again as elementary transmission elements in those settings in which this is allowed. For example, if the previous chain is to be expanded by another pair of a revolute and a prismatic joint, one can reuse the previous chain as follows

```
MoFrame K4 , K5 ;
MoAngularVariable beta2 ;
MoVector l2 ;
MoElementaryJoint R2 ( K2 , K3 , beta2 , xAxis ) ;
MoRigidLink      L2 ( K3 , K4 , l2 ) ;
MoMapChain DoublePendulum ;
DoublePendulum << SimplePendulum << R2 << L2 ;
```

4.3.4 A simple example

The foregoing concepts are be illustrated below by the example of two-link manipulator displayed in Fig. 4.6. The manipulator consists of two revolute joints R_1 and R_2 with non-orthogonal, non-intersecting axes. The vertical direction corresponds to the z -axis of the inertial reference frame \mathcal{K}_0 . The first joint rotates about the fixed z -axis, while the second joint is aligned with the x -axis of the moving frame. The links are assumed to point in direction of the fixed z -axis and the moving y -axis, respectively. Both links have a length of 1. The objective is to evaluate the joint torques when a load \mathbf{f}_E is applied to the tip of the manipulator in direction of the z -axis of the frame \mathcal{K}_4 . The corresponding `MOBILE` model is displayed below.

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMapChain.h>

main() {

    // definition of system topology

    MoFrame K0 , K1 , K2 , K3 , K4 ;
```

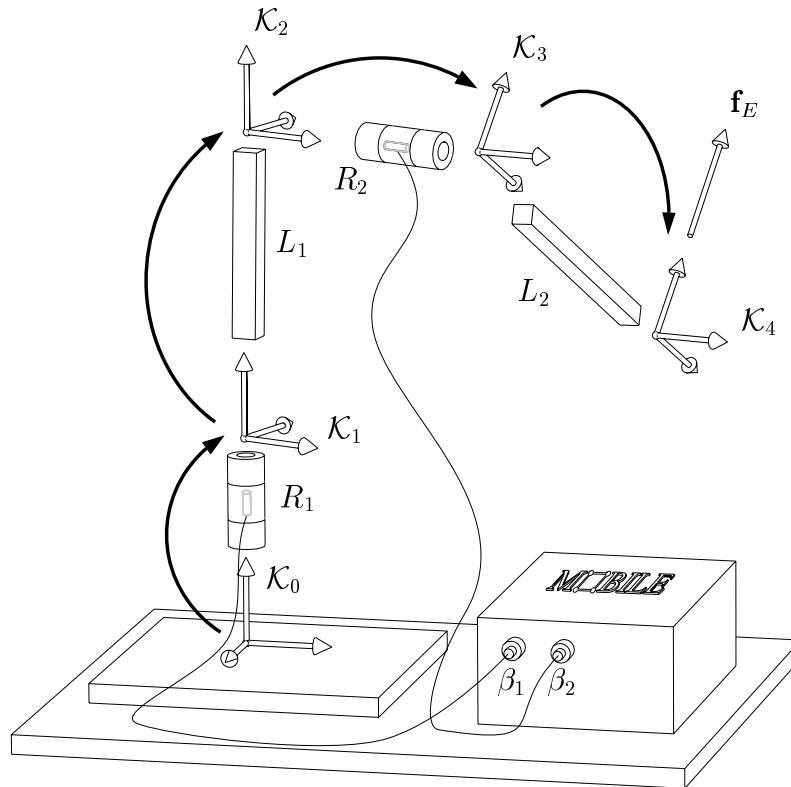


Figure 4.6: A Simple Manipulator

```

MoAngularVariable beta1 , beta2 ;
MoVector l1 , l2 ;
MoElementaryJoint R1 ( K0 , K1 , beta1 , zAxis ) ;
MoElementaryJoint R2 ( K2 , K3 , beta2 , xAxis ) ;
MoRigidLink      L1 ( K1 , K2 , l1 ) ;
MoRigidLink      L2 ( K3 , K4 , l2 ) ;
MoMapChain       SimpleManipulator ;
SimpleManipulator << R1 << L1 << R2 << L2 ;

// initialization of geometrical data

l1 = MoVector ( 0 , 0 , 1 ) ;
l2 = MoVector ( 0 , 1 , 0 ) ;

// simulation

int nsteps = 10 ;
MoReal forceMagnitude = 1.0 ;

beta1.q = 90.0 * DEG_TO_RAD ; // initial values
beta2.q = -45.0 * DEG_TO_RAD ; // ...

for ( int i = 0 ; i < nsteps ; i++ )

```

```

    {
    SimpleManipulator.doMotion ( DO_POSITION ) ;
    SimpleManipulator.cleanUpForces() ;
    K4.f = forceMagnitude * MoVector ( 0 , 0 , 1 ) * K4.R ;
    SimpleManipulator.doForce() ;
    cout << "Torque joint 1 = " << beta1.Q
          << "\nTorque joint 2 = " << beta2.Q
          << "\n" ;
    beta1.q += 90.0 * DEG_TO_RAD / float(nsteps) ;
    beta2.q += 180.0 * DEG_TO_RAD / float(nsteps) ;
    }
}

```

Note that the numerical value for vectors and matrices passed to kinetostatical transmission elements can be redefined anywhere in the program. The kinetostatical transmission elements retrieve this information during each motion or force traversal anew. Note also that the motion traversal is carried out in this example only at the position level. Traversing of the system at velocity or acceleration level is possible by passing as argument to `doMotion` the value “DO_POSITION|DO_VELOCITY|DO_ACCELERATION”.

4.4 Force and Mass Elements

Forces and mass effects are modeled in MOBILE as normal kinetostatic transmission elements. The only difference to the previously introduced components is that force and mass elements have no kinematical output, i. e., that they act as *leaves* attached to the rest of the system. One can imagine the multibody system as consisting on the one side of a kinematic ‘skeleton’, in which the interconnection structure is defined by massless links, joints, etc., and on the other hand of sources of dynamic effects, like forces and inertia, which are added on to the kinetostatic skeleton.

4.4.1 Force Elements

Force elements can produce either scalar or spatial loads. Currently, only two force elements are supported with the MOBILE software. One of these force elements is suitable for applying a spatial load whose components are measured either with respect to the inertial frame or with respect to the actual frame. The other generates scalar forces, either within joints or between frames.

Spatial loads are generated by objects of type “MoConstantWrench”. These objects allow the user to apply a ‘wrench’, i. e. a force and a moment, at an arbitrary reference frame of the system. The moment is always interpreted to be acting at the origin of the frame. However, the components of the force and moment can be interpreted in one of two ways. One can define *global* wrenches, which are defined with respect to the inertial frame, and *local* wrenches, which are decomposed with respect to the actual reference frame. Global wrenches are useful for modeling effects such as gravitational forces, although the objects

for generation of dynamical equations provide an alternative and more efficient way of accomplishing this. Local wrenches are defined with respect to the actual frame. They allow the user to model forces that follow the moving frame.

The following code fragment illustrates the creation of constant spatial loads with respect to the inertial and actual frames, respectively:

```
MoFrame K1 , K2 ;
MoVector f , n ;

MoConstantWrench moving ( K1 , f , n , LOCAL ) ; // moving force & torque
MoConstantWrench fixed ( K2 , f , n , GLOBAL ) ; // fixed force & torque
```

The objects “moving” and “fixed” are instances of an object applying a spatial load. The spatial load consists in both cases of a force “ f ” and a moment “ n ”. In the first case, the components of f and n are interpreted to be given with respect to $K1$. In the second example, the elements of f and n are interpreted to be given with respect to the inertial frame. If no fourth parameter is specified, it is given the default value `LOCAL`.

Scalar loads are generated by instances of class “`MoLinearSpringDamper`”. Objects of this type allow the user to apply a load which is a linear function of the displacement and the velocity of a scalar variable, i. e., which models a classical spring-damper element. The element can be attached either to a scalar chord or to an elementary joint.

In the case that the scalar load is applied to a *scalar chord*, the force is applied to two frames that are not necessarily adjacent. A scalar chord is a scalar measurement between two frames. For example, an object of type “`MoChordPointPointLinear`” measures the distance between the origins of two frames. A scalar load attached to this scalar measurement applies a force proportional to the value and to the first time derivative of this measurement. Fig. 4.7 illustrates the resulting force for this example. Further types of scalar chords will be discussed in the next chapter.

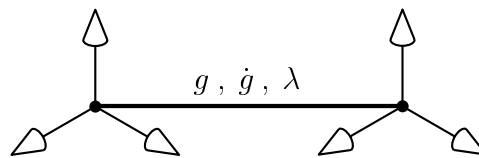


Figure 4.7: Elementary force element attached to a scalar measurement object

In the case that the scalar load is applied to a *joint*, the force is applied to the variable describing the relative motion of the joint as well as to the two frames that are adjacent to the joint. Only objects of type “`MoElementaryJoint`”, i. e., revolute or prismatic joints, are allowed for this kind of scalar loads.

The following code fragment illustrates the use of scalar loads.

```
MoReal k ; // stiffness coefficient
MoReal c ; // damping coefficient
```

```

// chord-based spring-damper element ...
MoFrame KChord1 , Kchord2 ;
MoChordPointPointLinear g ( KChord1 , KChord2 ) ; // scalar measurement
MoLinearSpringDamper spring1 ( g , k , c ) ; // spring-damper

// joint-based spring-damper element ...
MoFrame KJoint1 , KJoint2 ;
MoAngularVariable theta ;
MoElementaryJoint R ( KJoint1 , KJoint2 , theta ) ;
MoLinearSpringDamper spring2 ( R , k , c ) ;

```

4.4.2 Mass Elements

Mass elements model the inertia properties of a rigid body, i. e., its mass m and its inertia tensor Θ_C . In M□BILE, the inertia tensor is assumed to be defined with respect to the center of gravity C of the body. The center of gravity itself can be offset from the origin of a reference frame \mathcal{K} by a vector $\Delta\mathbf{s}$, as depicted in Fig. 4.8. As with the components of `MoFrame`, all tensorial quantities are always assumed to be decomposed with respect to the actual frame \mathcal{K} .

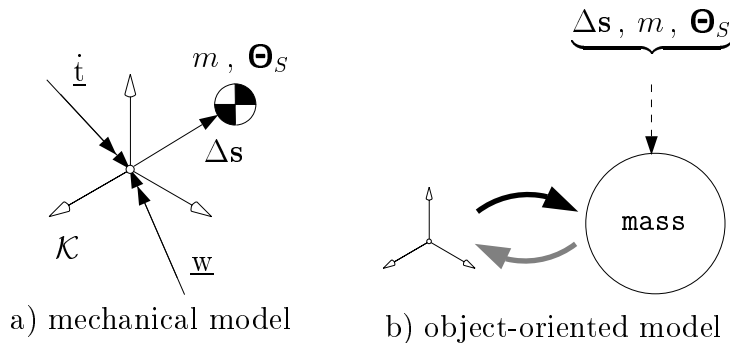


Figure 4.8: Model of a mass element.

Theoretical background: Modeling of inertia forces

In M□BILE 1.3, inertia properties are modeled as d’Alembert’s forces. Under a general motion of the frame \mathcal{K} , the d’Alembert’s forces exerted by the mass upon the origin of the frame \mathcal{K} are

$$\begin{aligned}
 \mathbf{f} &= -m[\mathbf{a} + \dot{\omega} \times \Delta\mathbf{s} + \omega \times (\omega \times \Delta\mathbf{s})] \quad , \\
 \boldsymbol{\tau} &= -[\Theta_S \dot{\omega} + \omega \times \Theta_S \omega] + \Delta\mathbf{s} \times \mathbf{f} \quad .
 \end{aligned}$$

Note that in contrast to the force element, the force of the mass element depends also on the acceleration of the frame to which it is attached. However, this dependency is only linear.

A general mass element takes as arguments

- the frame to which the mass properties are to be attached,

- a scalar parameter describing the mass of the body,
- an inertia tensor describing its moment of inertia, as well as
- a vector describing the offset of the center of mass of the body with respect to the origin to which the mass is attached.

By leaving out the inertia argument, one can model point masses. By leaving out the offset vector, one can model bodies whose center of mass is located at the origin of the frame to which the mass is attached. Leaving out such an argument is more efficient than passing an identity matrix or a zero vector, respectively.

The following code fragment illustrates the use of mass elements.

```
MoFrame K1 , K2 , K3 , K4 ;
MoReal m ;
MoInertiaTensor J ;
MoVector s ;

MoMassElement PointCentered ( K1 , m ) ;          // case 1
MoMassElement PointExcentric ( K2 , m , s ) ;    // case 2
MoMassElement BodyCentered ( K3 , m , J ) ;      // case 3
MoMassElement BodyExcentric ( K4 , m , J , s ) // case 4
```

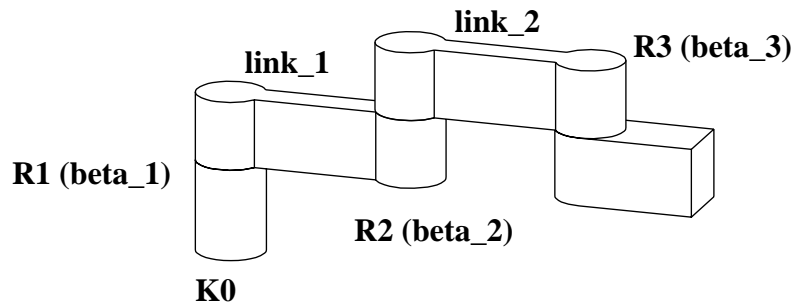
The code generates the following four mass elements

- case 1: a point mass attached to the origin of K1
- case 2: a point mass attached to K2 with offset vector \mathbf{s}
- case 3: a rigid body attached to K3 with center of mass coincident with origin of K3
- case 4: a rigid body attached to K4 with center of mass offset by vector \mathbf{s} from origin of K4

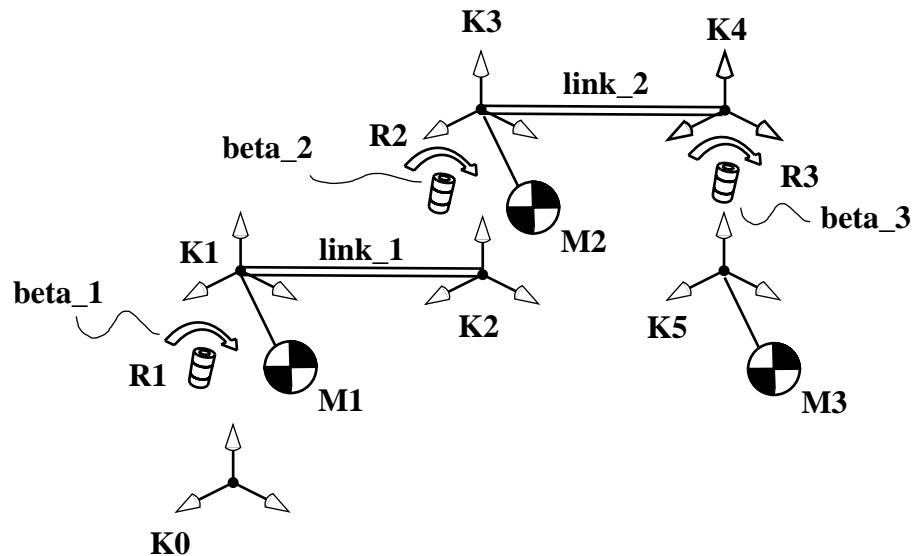
4.5 Example: Modeling of the Inverse Dynamics of a SCARA robot

The following section describes the modeling of the dynamics for a simple robotic system. The goal is to compute the joint torques that are necessary to achieve a prescribed motion of the robot links, i. e., the so-called *inverse dynamics* of the system.

The system is depicted in Fig. 4.9. It consists of three parallel revolute axis and three rigid links. The axes of the revolute joints are directed upwards in positive z direction. The frame K0 represents the inertial reference frame. All links have equal masses and



a) mechanical model



b) iconic model

Figure 4.9: Modeling of the inverse dynamics of a SCARA robot

moments of inertia. The last link is modeled only by its mass properties as its geometric dimensions are immaterial for the present analysis.

Below the corresponding MOBILE code is reproduced. The inertia tensor is defined by its diagonal entries. Moreover, the center of gravity of the links are offset by vectors p_1 , p_2 and p_3 from the origins of the frames of attachment. Note that the inverse dynamics model is just obtained by appending the mass elements to the kinetostatic chain representing the kinematic skeleton of the system.

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoMapChain.h>
```

```
void main ()
```

```
{
  MoFrame K0, K1, K2, K3, K4, K5 ;

  MoAngularVariable beta_1, beta_2, beta_3 ;

  MoElementaryJoint R1 ( K0, K1, beta_1, zAxis ) ;
  MoElementaryJoint R2 ( K2, K3, beta_2, zAxis ) ;
  MoElementaryJoint R3 ( K4, K5, beta_3, zAxis ) ;

  MoVector l1, l2, p1, p2, p3 ;
  l1 = l2 = p1 = p2 = p3 = MoNullState ;

  MoRigidLink link_1 ( K1, K2, l1 ) ;
  MoRigidLink link_2 ( K3, K4, l2 ) ;

  MoReal m = 2.4 ;
  MoInertiaTensor theta = MoInertiaTensor( 0.021, 0.004, 0.020 ) ;

  MoMassElement M1( K1, m, theta, p1 ) ;
  MoMassElement M2( K3, m, theta, p2 ) ;
  MoMassElement M3( K5, m, theta, p3 ) ;

  MoMapChain system ;
  system << R1 << link_1 << R2 << link_2 << R3 << M1 << M2 << M3 ;

  l1.y = l2.y = 1.0 ;
  p1.y = p2.y = p3.y = 0.5 ;

  beta_1.q = 0.00 ; beta_2.q = 0.20 ; beta_3.q = 0.30 ;
  beta_1.qd = 0.01 ; beta_2.qd = 1.00 ; beta_3.qd = 0.11 ;
  beta_1.qdd = 0.20 ; beta_2.qdd = 0.00 ; beta_3.qdd = 0.25 ;

  system.doMotion ( DO_ALL ) ;
  system.doForce ( DO_ALL ) ;

  cout << " beta_1.Q = " << beta_1.Q << endl ;
  cout << " beta_2.Q = " << beta_2.Q << endl ;
  cout << " beta_3.Q = " << beta_3.Q << endl ;
}
```

5 Objects for Closure of Loops

Multibody systems can feature two fundamental types of structure: (i) *tree-type* structure or (ii) *single* or *multiple loop* structure (see Fig. 5.1).

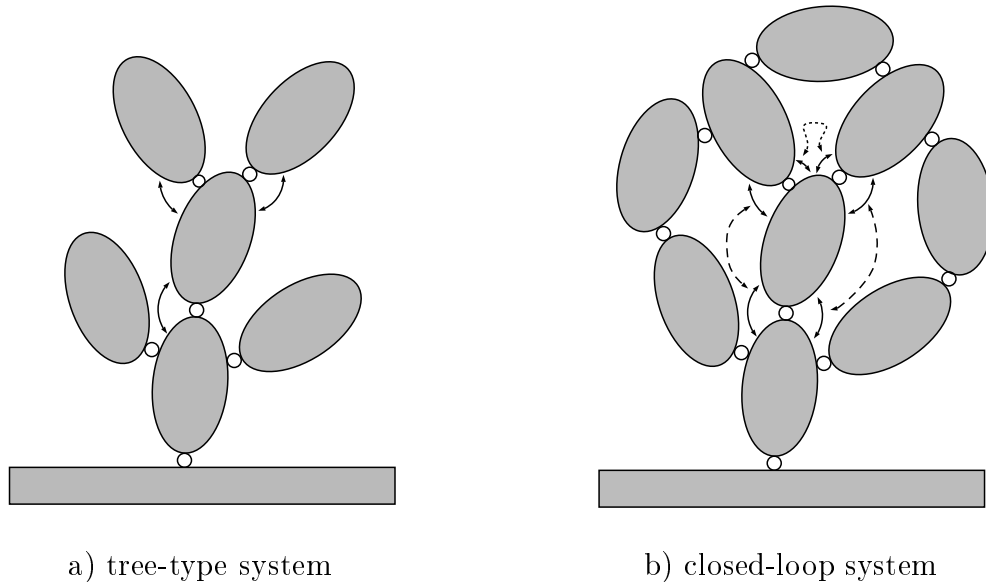


Figure 5.1: Comparison of tree-type and closed-loop systems

In systems featuring tree-type structure, there is one and only one path between any component and the inertial frame. Thus the relative motions between any two pairs of neighboring bodies are independent, and it is possible to process the kinetostatics of the elements on a component by component basis. A user concerned with the modeling of such a system just needs to concatenate its components in an order that is compatible with its topological structure, i. e., starting at the inertial system and ending at the tips of the branches.

When the bodies of the multibody system form closed loops, the relative motions within the loop become dependent; a change of relative motion at one place induces a change of relative motion at another place. Such dependencies make it impossible to proceed joint by joint or body by body as in the tree-type structure case. Instead, one has to formulate and solve so-called *constraint equations* or *closure conditions* that hold the branches of the loop together.

In M□BILE, the closure of loops is accomplished as a two-stage process:

- In a first stage, a set of “**characteristic measurements**” is defined whose vanishing indicates the closure of the loop. These measurements, also called “*chords*” in M□BILE, are typically generalized distances between geometric elements such as points, planes and lines. M□BILE provides a whole family of classes for making such measurements, which are derived from the (abstract) super-ancestor class “MoChord”. The objects instantiated from these classes are again kinetostatic trans-

mission elements, i. e., they can be used as any other kinetostatic transmission element to propagate motions and forces.

- In a next stage, one or more objects termed “**solvers**” are defined that are set to determine the dependent relative motions within the loop such that the measurements vanish. M□BILE supplies two classes for this purpose, which are both derived from the (abstract) super-class ‘MoSolver’. One solves the constraint equations by iterative, Newton-based procedures. This is the universal, generally applicable method. The other takes a scalar equation and solves it in *closed form* for an unknown joint variable. This method only works for special types of measurements and loop architectures. In both cases, the resulting solver objects behave again like kinetostatic transmission elements, supplying a motion and force transmission function.

The choice of optimal closure conditions and solution strategies for a given multibody system is a non-trivial task that renders no unique solution. Several approaches exist today for this purpose, each having its advantages and disadvantages depending on the objectives of the simulation. For example, users seeking a high degree of efficiency need to access closed-form solutions where possible in order to avoid redundant computations, while users requiring a rapid yet maybe not so efficient modeling are satisfied with iterative solution procedures. At present, M□BILE provides only the basic constituents for implementing the different loop closure and solution strategies. Users must select the ones that most closely fit their needs and concatenate them to appropriate transmission chains. Automatic loop-closure and resolution strategies will be installed in future versions of M□BILE. At present, the user is referred to the dedicated CA (Computer Algebra) program “SYMKIN”, which is a symbolical manipulation program written in *Mathematica* that automatically eliminates redundant computations and produces closed-form solutions where possible.

This chapter describes the basic mechanisms for generating and solving constraint equations with M□BILE. Some sections, particularly those concerning scalar measurements and closed-form solutions, are quite involved and may thus cause difficulties of understanding to the casual reader. However, these sections describe advanced solution techniques that can be skipped when only basic modeling features are required. Casual readers are advised to only browse through these sections, in order to grasp the basic ideas, and to apply the universal, better understandable iterative methods.

5.1 Basic Methods for Formulating Loop Closure Conditions

The basic procedure for tackling multibody loops in M□BILE is to first dissect the originally closed loop into serial chains and then to bring again the loose ends of the serial chains together by requiring the fulfillment of appropriate closure conditions. Hereby, the following three basic methods are possible (see Fig. 5.2):

- C1 Body Assembly Method.** The loop is dissected at a *body*. The loop closure condition corresponds to equality of *pose* for the two reference frames at both sides

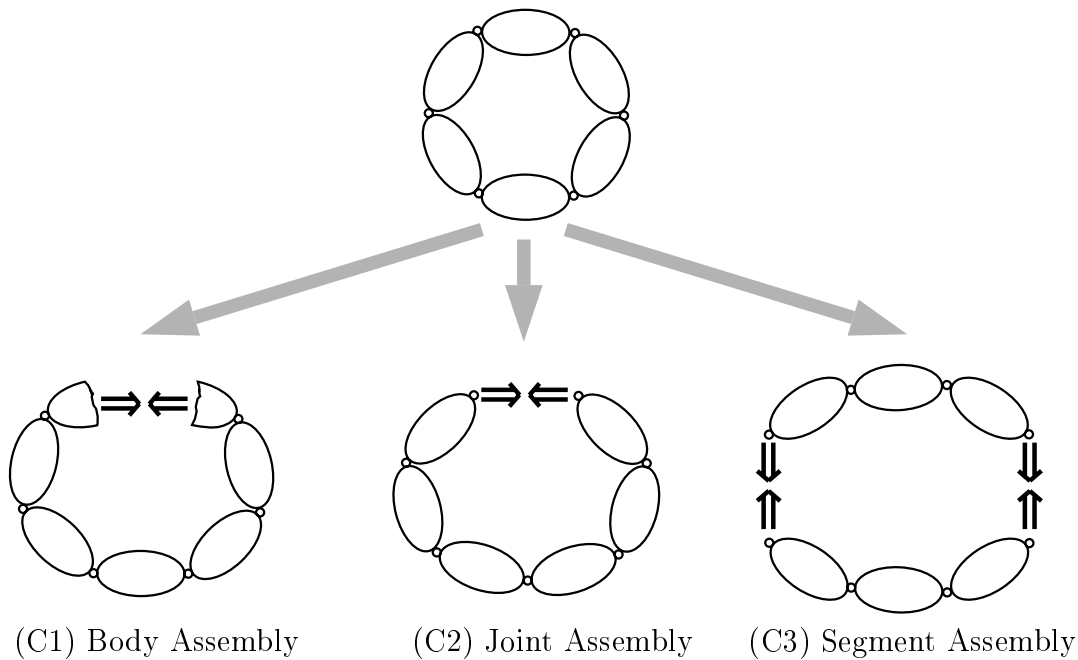


Figure 5.2: Three basic methods for modeling loops in MABLE

of the cut, where the term *pose* stands for displacement and rotation. This kind of assembly is applicable to any type of loop. It may however perform only poorly both in terms of computational efficiency and in terms of numerical stability of the computed solution. In MABLE, pose closure conditions are generated by objects of type “MoChord3DPose”.

- C2 Joint Assembly Method.** The loop is dissected at a *joint*. The loop closure condition consists in the equality of the geometric elements left invariant by the joint for the two reference frames at both sides of the cut. Currently, this type of assembly is only applicable to loops featuring a spherical joint; the cut is then performed at the spherical joint and the closure condition is produced by an object of type “MoChord3DTranslation”.
- C3 Segment Assembly Method.** The loop is dissected at *two joints*. The loop closure condition is formulated by taking characteristic measurements between the end frames of the two resulting segments and setting these measurements equal. The characteristic measurements depend on the type of the two cut joints; the types of measurements currently supplied with the MABLE software are described in Section 5.2.5. This type of assembly is only applicable to systems of constraint equations that are solvable in closed form. It is the most involved of all methods described here; however, its numerical advantages, such as computational efficiency and numerical stability, make it a good choice for advanced modeling. Currently, MABLE is the only multibody package supporting this kind of modeling.

For all three methods, the user has to carry out the following steps

1. decide where to cut the loop apart
2. decide which of the joint variable(s) of the loop are to be treated as dependent variable(s), and put these together in an object of type “MoVariableList” in case there are more than one unknowns; the other variables and motions are regarded as independent variables or *kinematic inputs* of the loop
3. create one or more object(s) modeling the *dependent* chains of the dissected loop; each dependent chain is typically an object of type MoMapChain containing the kinetostatics from the dependent variables to the cut frames
4. create one or more object(s) derived from type “MoChord” that describe the loop closure condition(s)
5. create an object of type “MoSolver”, passing to it the dependent chain(s), the (list of) dependent variable(s), and the object representing the closure condition(s)

After carrying out these steps, the user can employ the resulting object of type MoSolver as a simple kinetostatic transmission element representing the kinetostatics of the closed loop(s). The doMotion function of the solver generates the motion of the dependent chains so that they follow the input motion while keeping the loop closed; the doForce function computes the forces at the cut frames and within the dependent chains so that static equilibrium is achieved. Usage of this object is then fully equivalent to the usage of any other kinetostatic transmission element such as an elementary joint or a rigid link, i. e., solver objects can be used again as constituents of chains of kinetostatic transmission elements or even “super loops” exhibiting in their branches other loops.

Note that the choice of independent (and by this also of dependent) variables for a given loop is in general a non-uniquely solvable and in some cases difficult task. From a topological viewpoint, all joint variables are potential candidates for being selected as input variables. However, the possible occurrence of turning points and limit (e. g., stretched) configurations limits the usefulness of some of these selections. Luckily, in most industrial applications the places to be chosen as inputs are clearly marked by driving units or major motion directions. However, for some (academic) examples, such a choice may be non-evident or even impossible to make for a general simulation. These cases require a careful assessment by the user. This manual provides no instructions on how to choose appropriate input variables for a mechanism. For this purpose, the reader is referred to the supplied examples or, for more difficult cases, to the specialized literature.

5.1.1 Example: Inverse Dynamics of a Spatial Shaker Mechanism

For illustration of the just mentioned concepts, a MOBILE model for the inverse dynamics of a simple multibody loop is regarded. The objective is to compute the motion of the system at a prescribed configuration of the input variable(s) and the input torque(s)

required to drive the mechanism at this position at a given constant speed. The corresponding MOBILE code is reproduced below and shall be commented in the following paragraphs. The model makes use of some classes that are new to the reader at this point. These classes will be explained in more detail in the subsequent sections. At present, it is sufficient to grasp the basic ideas behind the implementation of the five steps described above and the use of objects of type “MoChord” and “MoSolver”.

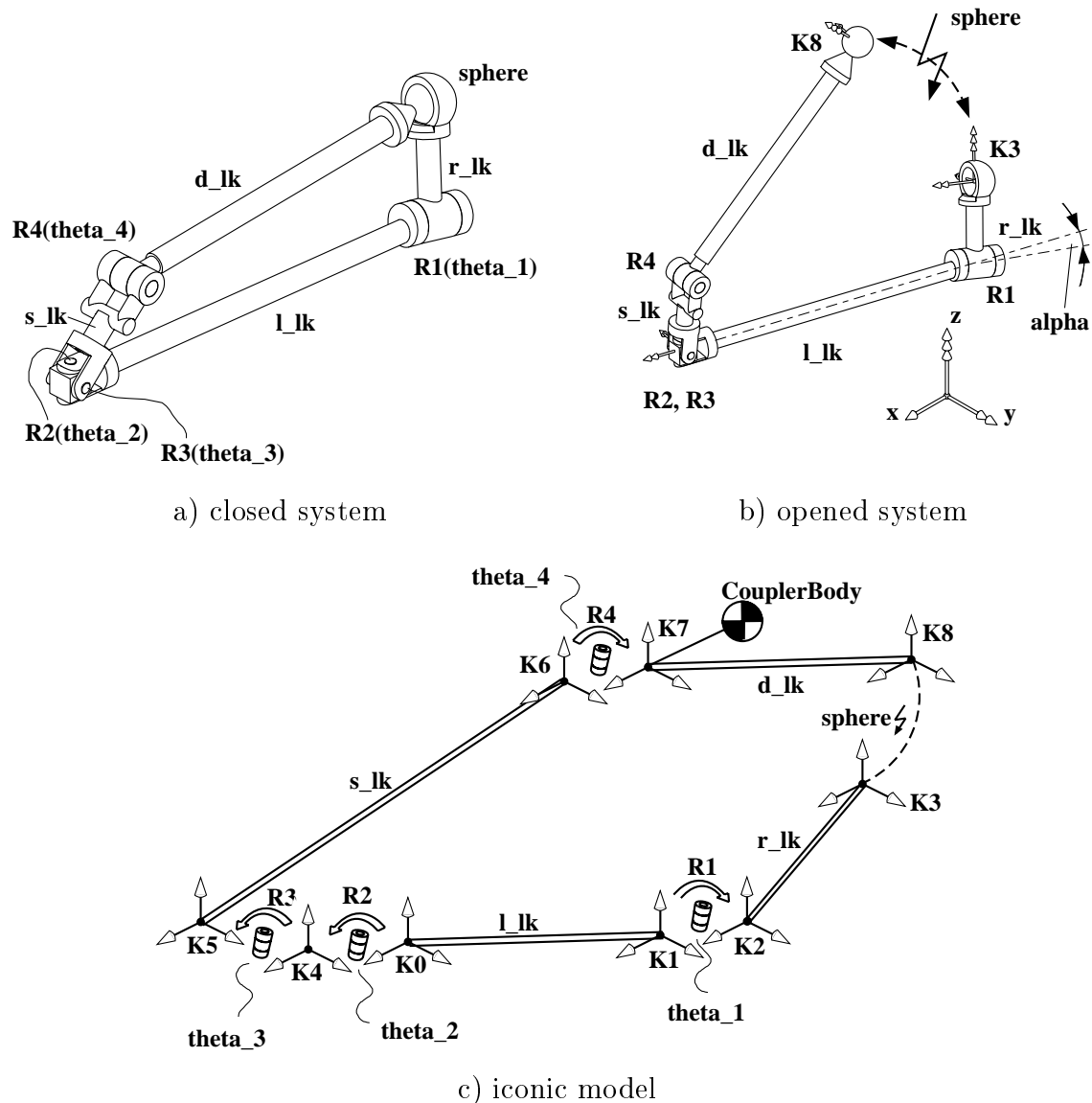


Figure 5.3: Analysis of a shaker mechanism

The regarded system involves seven joint variables, two at the revolute joints R1 and R4, two at the Hooke joint, which is modeled by two revolute joints R2 and R3 with orthogonally intersecting axes, and three at the spherical joint “sphere” (see Fig. 5.3a). According to the six general spatial loop closure conditions, only *one* of these can be chosen as kinematical input, which is here the rotation of joint R1.

For the formulation of the loop kinematics, one of the three methods described above has to be selected. Because of the occurrence of a spherical joint, method C2, i. e., the Joint Assembly Method, is appropriate. This yields the open structure depicted in Fig. 5.3b.

The cut at the spherical joint introduces three constraint equations, corresponding to the concurrence of the origins of the two cut frames K3 and K8. The three joint variables at the spherical joint are hereby eliminated from the analysis. Thus, only the rotations `theta_2`, `theta_3` and `theta_4` at the joints R2, R3 and R4 (see Fig. 5.3c) remain as dependent variables for this analysis. They are collected in the variable list “`dependentVars`”.

Fig. 5.3c illustrates the other components employed in the MOBILE model. The independent motion of the loop is subsumed in the transmission chain “`input`”, which contains the joint R1 and the link `r_1k`. This takes care of the (independent) motion of the right crank. The dependent motion is collected in the transmission chain “`dependentChain`”. The objective of this chain is to produce the motion of the cut frames K3 and K8 as a function of the three dependent variables. In the present case, only K8 depends on the aforementioned variables. Thus only the three revolute joints R2, R3 and R4 as well as the two connecting links `s_1k` and `d_1k` need to be assembled in the dependent chain. The link `l_1k` in the base does not move during motion. It needs to be traversed only *once* during simulation, and is thus not included in any of the two transmission chains described above.

The closure conditions are evaluated by the object “`sphere`” of type `MoChord3DPosition`, which measures the difference vector from the origin `origin` of K3 and to the origin of K8. The solution of the constraint equations is performed by the object “`Solver`”, which is of type `MoImplicitSolver` and thus solves the constraint equations iteratively (based on a Newton-like algorithm). The initialization of `Solver` is accomplished by three arguments: (1) the measurement object “`sphere`”, whose vanishing signals the closure of the loop, (2) the list of variables “`dependentVars`”, in which the three dependent variables are subsumed, and (3) the transmission chain “`dependentChain`”, which implements the motion of the cut frames as a function of the dependent variables. Note that the number of variables in `dependentVars` coincides with the number of components of the closure condition. Note also that chain representing the input motion is not passed to the solver object. This is not necessary because the input motion needs to be evaluated only *once* prior to the loop closure procedure and not multiple times during the solver iterations.

After solving the constraints, the dependent chain is located appropriately by the solver, and one can concatenate more elements to the loop. In the example below this is done by attaching a mass element “`CouplerBody`” to frame “K7” of the coupler link “`d_1k`”.

The complete kinetostatics of the loop can be now subsumed as a transmission chain “`loopKinetostatics`” comprising the following three blocks, which mirror also the recommended loop kinetostatics processing structure to be employed with MOBILE: (1) the transmission chain “`input`”, which implements all motions that can be carried out *before* solving the constraints, (2) the solver element “`Solver`”, which contains also the dependent chain that must be moved *while* solving the constraints, and (3) the *post-solution* element “`CouplerBody`” (in general a transmission chain), which subsumes all tasks that

can be accomplished *after* solving the loop constraints. Note that, as the solver invokes internally the kinematics and statics of the dependent chain, it is not necessary (or even correct) to traverse this chain again after processing the constraint solver.

The ensuing simulation consists of a kinematical and statical traversal of the system. In the kinematic part, first the input crank is moved, then the kinematics of the constraint solver are processed (which implies also moving the dependent chain) and then the mass element is put in place; in the statics part, first the force produced by the mass element is applied, then the statics of the constraint solver are processed (which involves also traversing the dependent chain), and finally the statics of the input crank are computed. The value of the force state subentry “.Q” of the variable “theta_1” then represents the sought generalized force that has to be applied in order to move the loop as required.

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoImplicitConstraintSolver.h>
#include <Mobile/MoMassElement.h>

void main() {
    // reference frames
    MoFrame    K0 , K1 , K2 , K3 , K4 , K5 , K6 , K7 , K8 ;

    // state-variables
    MoAngularVariable theta_1 , theta_2 , theta_3 , theta_4 ;

    // joints
    MoElementaryJoint R1 ( K1 , K2 , theta_1 , yAxis ) ;
    MoElementaryJoint R2 ( K0 , K4 , theta_2 , zAxis ) ;
    MoElementaryJoint R3 ( K4 , K5 , theta_3 , xAxis ) ;
    MoElementaryJoint R4 ( K6 , K7 , theta_4 , xAxis ) ;

    // links connecting joints
    MoVector r , l , s , d , d_s ;
    MoRotationMatrix A ;
    MoRigidLink l_lk ( K0 , K1 , l , A ) ;
    MoRigidLink r_lk ( K2 , K3 , r ) ;
    MoRigidLink s_lk ( K5 , K6 , s ) ;
    MoRigidLink d_lk ( K7 , K8 , d ) ;

    // subsystem containing only input motion
    MoMapChain input ; input << R1 << r_lk ;

    // subsystem to be iterated while solving the constraints
    MoMapChain dependentChain ;
    dependentChain << R2 << R3 << s_lk << R4 << d_lk ;

    // constraint solving objects
    MoChord3DPosition sphere ( K8 , K3 ) ;
    MoVariableList dependentVars ;
    dependentVars << theta_2 << theta_3 << theta_4 ;
    MoImplicitConstraintSolver
        Solver ( sphere , dependentVars , dependentChain ) ;
}
```

```

// mass element
MoReal      m_d ;
MoInertiaTensor  THETA_d ;
MoMassElement  CouplerBody ( K7 , m_d , THETA_d , d_s ) ;

// complete kinetostatics
MoMapChain loopKinetostatics ;
loopKinetostatics << input << Solver << CouplerBody ;

// geometry and mass properties
r = l = s = d = d_s = MoNullState ;
MoReal alpha = DEG_TO_RAD * 10.0;
MoXRotation X = alpha;
A          = X;
l.y        = -0.8 ;           // offset at the base
r.z        = 0.2 ;           // length of r_lk
s.z        = 0.2 ;           // length of s_lk
d.y        = -0.7 ;           // length of coupler
d_s.y      = -0.5*0.7 ;      // position of center of mass
m_d        = 1.0 ;
THETA_d    = MoVector(4.08895833e-2 , 1.125e-4 , 4.08895833e-2);

// initial conditions
theta_1.q  = 1.0 * DEG_TO_RAD;

// compute inverse dynamics
loopKinetostatics.doMotion ( DO_ALL ) ; // propagate motion
loopKinetostatics.doForce ( DO_ALL ) ; // propagate forces

cout << "Computed torque at joint R1 = " << theta_1.Q ;
}

```

5.2 Measurement Objects

Measurement objects are special objects introduced in MOBILE to map the motion of moving frames to scalar or spatial quantities. The measurements are taken either directly between the frames, or between pairs of geometric elements such as points, planes, lines, etc. attached to these frames. In MOBILE, there are several types of measurement objects, also termed “*chords*”, that are a combination of different geometric, topological, and activity types. Below we reproduce the characteristic features these attributes and the basic functioning of the measurement objects. The usage of these objects for constraint equation formulation and solution is explained in Section 5.3.

5.2.1 Basic Properties of Measurements

Measurement objects behave in MOBILE like any other kinetostatic transmission element. The kinematic transmission consists in mapping the motion of the frames to the

measurement quantity. The force transmission involves the mapping of force components associated with the measurement to the corresponding spatial forces at the frames.

For illustration of this concept, imagine a clothes line spanned between two hooks, one at a wall and one at a tree (Fig. 5.4). If a wind gust makes the tree sway, the length of the clothes line as well as its first and second time derivatives will vary. The computation of this length variations corresponds to the motion transmission at position, velocity and acceleration level, respectively. Also, depending on the stiffness of the line, a (scalar) tension will be induced, which produces corresponding spatial forces at the hooks. This represents the force transmission.

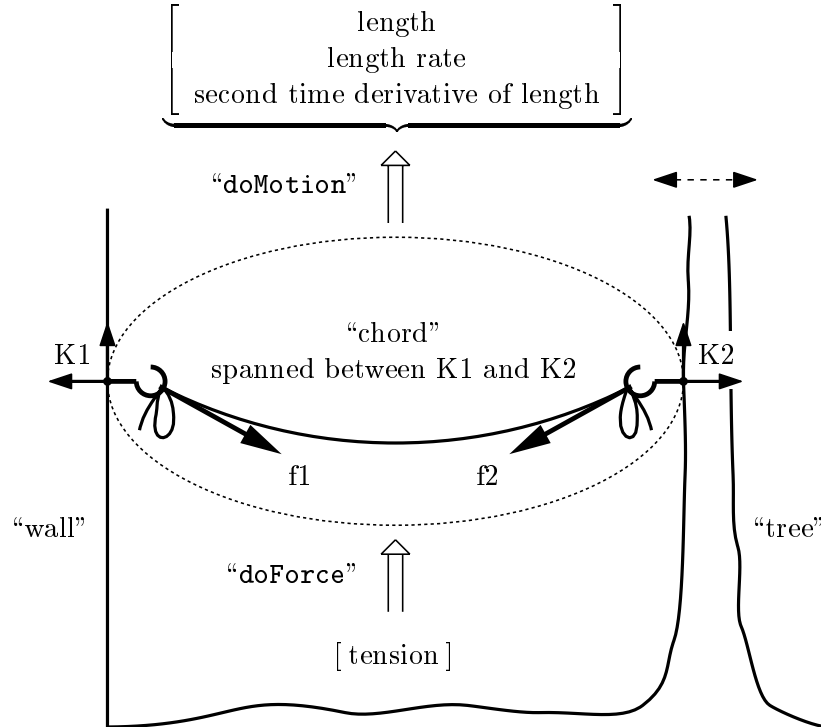


Figure 5.4: Example of a “chord”

The measurement objects of M \square BILE are characterized by three basic attributes:

- (A) the *geometric type*, determined by the type of geometric elements (point, plane, line, or reference system) involved in the measurement; measurements generating tensorial quantities are hereby denoted by *spatial* measurements, while measurements producing scalar outputs are termed *scalar* measurements;
- (B) the *topological type* which is determined by the number of frames involved in the measurement, as well as the type of motion (relative or absolute) regarded in the measurement; and
- (C) the *activity type*, which characterizes the behaviour (static or self-reconfiguring) of the measurement with respect to the motion of the involved frames.

Measurement objects in MOBILE can combine almost any pattern of attributes from the three basic types listed above. This results in a quite large number of potential measurement objects. Table 5.1 lists the geometric types of measurements currently supplied with the MOBILE software. The measurements described there are described based on the topological type involving only two frames. For scalar measurements, other topological types exist which involve up to four frames. This is described in more detail in Section 5.2.5. The two activity types of measurement are explained in Section 5.2.2.

type of measurement	action when applied to two reference frames
MoChord	base class for chord elements
MoChordPlanePlane	cosine of the angle between two coordinate planes
MoChordPlanePoint	shortest distance from a base plane and a target point
MoChordPointPointQuadratic	squared distance between two points
MoChordPointPointLinear	linear distance between two points
MoChordPointPlane	distance from a base point to a target plane
MoChord3DPosition	difference of two radius vectors
MoChord3DOrientation	relative transformation between two frames
MoChord3DPose	difference vector and relative transformation between two frames

Table 5.1: Geometric types of measurement objects

MOBILE supplies for each measurement object a pointer “**state**”. Normally, the user does not need to be concerned with this state, as it is processed automatically when the measurement object is used as constituent for higher-level objects such as solvers, spring-damper elements, etc. However, if required, the user can access the internal state by appending “.**state**” to the name of the object. For scalar measurements, this pointer references a linear variable containing the actual position, velocity, acceleration and load of the measure. For spatial measurements, the pointer addresses an array of linear variables representing the elements of the vectors and/or matrices associated with the measurement. For example, for objects of type **MoChord3DPosition**, the pointer references an array containing the three components of the difference vector between the origins of the two measured frames. This is illustrated by a program example in Section 5.2.2.

5.2.2 Self-Reconfiguring Measurements

Measurements normally just take the actual state of the involved frames and map them to the corresponding scalar or vectorial quantities of the measurement. In order to reduce the modeling effort, MOBILE also allows the user to include a transmission chain in the measurement that is invoked each time the measurement object is traversed. Such measurements are termed *self-reconfiguring measurements*.

For an illustration of this concept, consider the task of measuring the vector from a moving camera to the end-effector of a robot (Fig. 5.5). Let the motion of the robot be carried

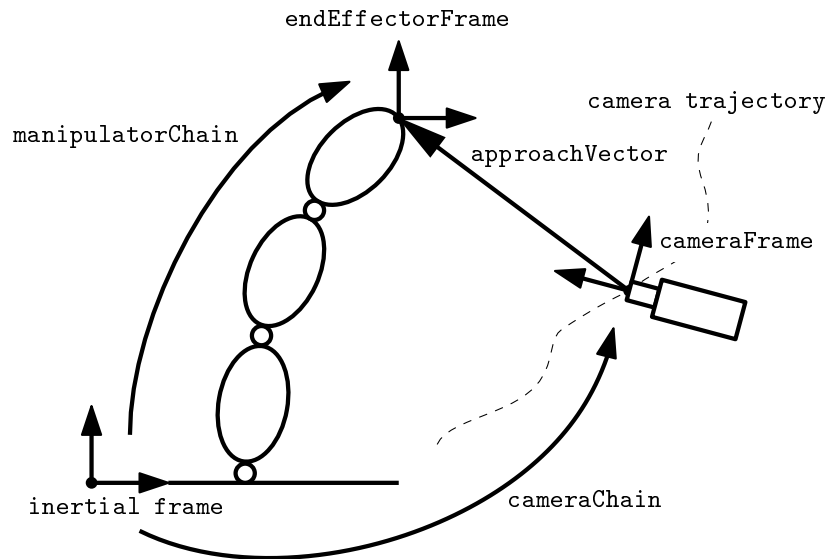


Figure 5.5: Example of a measurement for a moving object

out by a transmission chain “manipulatorChain”, while the motion of the camera is produced by the object “cameraChain”. In order to measure the actual motion, the user first has to invoke the “doMotion()” functions for the manipulator chain and the camera chain, respectively, and then that of the measurement object. A corresponding program fragment might look like this:

```

MoFrame    endEffectorFrame;
MoFrame    cameraFrame;
MoMapChain manipulatorChain; // robot kinematics
MoMapChain cameraChain ;    // camera motion
MoVector   approachVector ; // difference vector

// definition of measurement object

MoChord3DPosition simpleChord ( endEffectorFrame , cameraFrame ) ;

// definition of transmission chains
...
// measurement (simulation)

manipulatorChain.doMotion(); // move the manipulator
cameraChain.doMotion();      // move the camera
simpleChord.doMotion();       // make the measurement

// extraction of coordinates of approach vector

approachVector.x = simpleChord.state[0].q ;
approachVector.y = simpleChord.state[1].q ;
approachVector.z = simpleChord.state[2].q ;

```

By including the motion of the camera in the measurement, a more compact code results:

```

MoFrame EEFrame;           // end effector frame
MoFrame cameraFrame;
MoMapChain manipulatorChain; // robot kinematics
MoMapChain cameraChain ;   // camera motion
MoVector approachVector ;  // difference vector

// definition of measurement object

MoChord3DPosition movingChord ( EEFrame , cameraFrame , cameraChain ) ;
...
// measurement

manipulatorChain.doMotion(); // move manipulator
movingChord.doMotion();      // make measurement moving internally the camera

// extraction of global coordinates of approach vector
...

```

The use of such self-reconfiguring measurement objects can be quite useful when modeling of some types of constraint equations and/or force elements, as discussed further below.

5.2.3 Lists of Measurements

Like state variables or transmission elements, measurement objects can be concatenated into lists. If only the kinetostatic transmission properties of the measurement objects are of interest, the concatenation can take place with objects of type `MoMapChain`. However, if the list of chords are to be employed as a set of closure conditions to be solved by a constraint solver, the list of measurements must be assembled in a special list type termed “`MoChordList`”. An example of the use of chord lists is

```

MoFrame K1 , K2 , K3 , K4 ;
MoChordPointPointLinear chord1 ( K1 , K2 ) ;
MoChordPointPointQuadratic chord2 ( K3 , K4 ) ;
MoChordList chords ;
chords << chord1 << chord2 ;

```

A chord list can be used in any setting allowed for a composite chord (i.e., *not* where only scalar chords are required). The sequence of concatenation is hereby immaterial, as the solution of a system of equations does not depend on the order in which the equations are supplied.

5.2.4 Spatial Measurements

Spatial measurements generate vector and matrix quantities describing the relative pose of two reference frames. Currently, there exist three types of spatial measurement objects in M□BILE:

- objects measuring the **relative displacement** between the origins of two frames (class `MoChord3DPosition`)
- objects measuring the **relative orientation** between two frames (class `MoChord3DOrientation`)
- objects measuring both the **relative displacement and relative orientation** between two frames (class `MoChord3DPose`)

Spatial measurement objects are initialized with two frames: the “*from*” frame, which acts as the base for the measurement, and the “*to*” frame, which is the target. The discerning of these two frames is significant only when the sign of a scalar measurement or the tensorial quantities involved in a spatial measurement are of interest. If the user is only interested in establishing a measurement for later use in a solver, the order is immaterial. An example of an initialization of spatial measurements is

```
MoFrame Kfrom1 , Kto1 ;
MoChord3DPosition distance ( Kto1 , Kfrom1 ) ;
MoFrame Kfrom2 , Kto2 ;
MoChord3DPose pose ( Kto2 , Kfrom2 ) ;
```

Theoretical background: Kinetostatics of Spatial Measurements

Objects for spatial measurements give rise to two geometric entities related to rigid body motion: the vector $\Delta \mathbf{r}$ pointing from the origin of the “from” frame $\mathcal{K}_{\text{from}}$ to the origin of the “to” frame \mathcal{K}_{to} , and the transformation matrix $\Delta \mathbf{R}$ transforming from components with respect to \mathcal{K}_{to} to components with respect to $\mathcal{K}_{\text{from}}$ (Fig. 5.6).

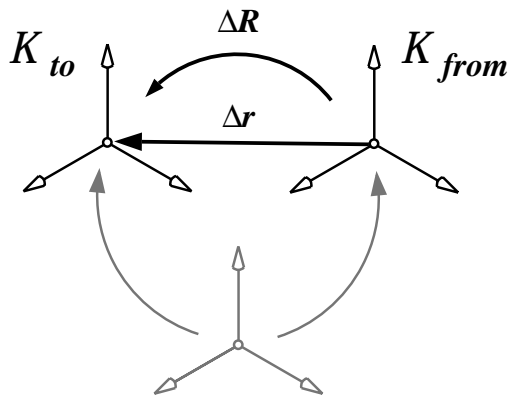


Figure 5.6: A spatial measurement

The formulas involved in the kinematics of the spatial measurements are summarized in Table 5.2.

The operator “vect” in the rotational part extracts three independent quantities from the rotation matrix that are used for formulation of the rotational constraint equations. These three numbers can be regarded as coefficients of a vector representing a ‘small’ rotation increment. This vector behaves like a vector of angular velocity, as explained below.

translational part	rotational part
$\Delta \mathbf{r} = \mathbf{R}_{\text{To}} \mathbf{r}_{\text{To}} - \mathbf{R}_{\text{From}} \mathbf{r}_{\text{From}}$	$\text{vect}(\Delta \mathbf{R}) = \text{vect}(\mathbf{R}_{\text{From}} \mathbf{R}_{\text{To}}^{\text{T}})$
$\Delta \mathbf{v} = \mathbf{R}_{\text{To}} \mathbf{v}_{\text{To}} - \mathbf{R}_{\text{From}} \mathbf{v}_{\text{From}}$	$\Delta \boldsymbol{\omega} = \mathbf{R}_{\text{To}} \boldsymbol{\omega}_{\text{To}} - \mathbf{R}_{\text{From}} \boldsymbol{\omega}_{\text{From}}$
$\Delta \mathbf{a} = \mathbf{R}_{\text{To}} \mathbf{a}_{\text{To}} - \mathbf{R}_{\text{From}} \mathbf{a}_{\text{From}}$	$\Delta \dot{\boldsymbol{\omega}} = \mathbf{R}_{\text{To}} \dot{\boldsymbol{\omega}}_{\text{To}} - \mathbf{R}_{\text{From}} \dot{\boldsymbol{\omega}}_{\text{From}}$

Table 5.2: Basic formulas for spatial measurements (kinematics)

An infinitesimally small rotation $\delta \mathbf{R}$ can be written as

$$\delta \mathbf{R} = \underbrace{\mathbf{I}_3}_{\text{symmetric}} + \underbrace{\begin{pmatrix} 0 & -\delta\varphi_z & \delta\varphi_y \\ \delta\varphi_z & 0 & -\delta\varphi_x \\ -\delta\varphi_y & \delta\varphi_x & 0 \end{pmatrix}}_{\text{skew-symmetric}} = \mathbf{I}_3 + \widetilde{\delta \underline{\varphi}} .$$

Thus, an infinitesimal rotation can be decomposed in the sum of the identity transformation and a perturbing part that is purely skew-symmetric. Hereby, the skew-symmetric part has three independent elements that can be put together in a vector $\delta \underline{\varphi} = [\varphi_x, \varphi_y, \varphi_z]^{\text{T}}$. The original skew-symmetric matrix is then reconstructed by making use of the tilde operator.

Regarding now a general matrix $\Delta \mathbf{R}$, one can decompose

$$\Delta \mathbf{R} = \underbrace{\frac{1}{2}(\Delta \mathbf{R} + \Delta \mathbf{R}^{\text{T}})}_{\text{symmetric}} + \underbrace{\frac{1}{2}(\Delta \mathbf{R} - \Delta \mathbf{R}^{\text{T}})}_{\text{skew-symmetric}} .$$

If $\Delta \mathbf{R}$ is only a small deviation from the identity transformation (as expected for example in a Newton-like procedure), its symmetric part is approximated by the identity matrix and its skew-symmetric part is approximated by a skew-symmetric perturbation term as described above. Hence, the closure condition takes the form

$$g(\Delta \mathbf{R}) = \Delta \mathbf{R} - \mathbf{I}_3 \approx \frac{1}{2}(\Delta \mathbf{R} - \Delta \mathbf{R}^{\text{T}}) \stackrel{!}{=} 0 .$$

The residuum of the rotational closure is thus approximated by a skew-symmetric matrix

$$\widetilde{\delta \underline{\varphi}} = \frac{1}{2}(\Delta \mathbf{R} - \Delta \mathbf{R}^{\text{T}}) ,$$

whose independent elements can be put together in a vector $\delta \underline{\varphi}$. The operation “vect” thus extracts exactly the independent components from the perturbation $\Delta \mathbf{R}$. In MOBILE, this vector is slightly modified in order to cope with rotations near to 180°. The thus ensuing components are

$$\left. \begin{aligned} \hat{\varphi}_x &= \delta\varphi_x + \mu \cdot [3 - \text{trace}(\Delta \mathbf{R})] \\ \hat{\varphi}_y &= \delta\varphi_y + \mu \cdot [3 - \text{trace}(\Delta \mathbf{R})] \\ \hat{\varphi}_z &= \delta\varphi_z + \mu \cdot [3 - \text{trace}(\Delta \mathbf{R})] \end{aligned} \right\} , \text{ where } \mu = \begin{cases} 0 & \text{for } \|\delta \underline{\varphi}\| > \epsilon \\ 1 & \text{for } \|\delta \underline{\varphi}\| < \epsilon \end{cases}$$

and $\epsilon = 10^{-4}$ is a predefined constant that can be changed by the user. This modified residuum vector is what is stored in the array ‘state’ of the rotational measurement.

The equations for the statics of spatial measurements are summarized in Table 5.3. The objects take here a force $\Delta \mathbf{f}$ and/or a torque $\Delta \tau$ related to the measurement and apply them to the frames $\mathcal{K}_{\text{from}}$ and \mathcal{K}_{to} . Note, again, that the order of attachment frames has influence on the sign of the applied forces and/or torques.

translational part	rotational part
$\mathbf{f}_{\text{To}} = \mathbf{R}_{\text{To}}^T \Delta \mathbf{f} ; \mathbf{f}_{\text{From}} = -\mathbf{R}_{\text{From}}^T \Delta \mathbf{f}$	$\tau_{\text{To}} = \mathbf{R}_{\text{To}}^T \Delta \tau ; \tau_{\text{From}} = -\mathbf{R}_{\text{From}}^T \Delta \tau$

Table 5.3: Basic formulas for spatial measurements (statics)

The result of the measurement is stored in an array of linear variables. This array is explained below.

Table 5.4 gives an overview of the state subentries of the array addressed by `state` for the different types of spatial measurements currently supplied with M□BILE.

element	MoChord3D...		
	Position	Orientation	Pose
state[0].q	$\Delta r.x$	$\hat{\varphi}.x$	$\Delta r.x$
state[1].q	$\Delta r.y$	$\hat{\varphi}.y$	$\Delta r.y$
state[2].q	$\Delta r.z$	$\hat{\varphi}.z$	$\Delta r.z$
state[3].q			$\Delta s.x$
state[4].q			$\Delta s.y$
state[5].q			$\Delta s.z$
state[0].qđ	$\Delta v.x$	$\Delta \omega.x$	$\Delta v.x$
state[1].qđ	$\Delta v.y$	$\Delta \omega.y$	$\Delta v.y$
state[2].qđ	$\Delta v.z$	$\Delta \omega.z$	$\Delta v.z$
state[3].qđ			$\Delta \omega.x$
state[4].qđ			$\Delta \omega.y$
state[5].qđ			$\Delta \omega.z$
state[0].qđđ	$\Delta a.x$	$\Delta \dot{\omega}.x$	$\Delta a.x$
state[1].qđđ	$\Delta a.y$	$\Delta \dot{\omega}.y$	$\Delta a.y$
state[2].qđđ	$\Delta a.z$	$\Delta \dot{\omega}.z$	$\Delta a.z$
state[3].qđđ			$\Delta \dot{\omega}.x$
state[4].qđđ			$\Delta \dot{\omega}.y$
state[5].qđđ			$\Delta \dot{\omega}.z$
state[0].Q	$\Delta f.x$	$\Delta \tau.x$	$\Delta f.x$
state[1].Q	$\Delta f.y$	$\Delta \tau.y$	$\Delta f.y$
state[2].Q	$\Delta f.z$	$\Delta \tau.z$	$\Delta f.z$
state[3].Q			$\Delta \tau.x$
state[4].Q			$\Delta \tau.y$
state[5].Q			$\Delta \tau.z$

Table 5.4: Elements of the array ‘state’ for spatial measurements

5.2.5 Scalar Measurements

Scalar measurements generate projections from spatial frames to real numbers. The basic idea of this projection is illustrated in Fig. 5.7 in its most simple form. The measurement

object takes the motion of two frames, termed the *target* frame \mathcal{K}_E and the *base* frame \mathcal{K}_B , and produces a scalar quantity that depends only on the *relative motion* between both frames. An example of such a measurement is the distance between the origins of the frames depicted in Fig. 5.7.

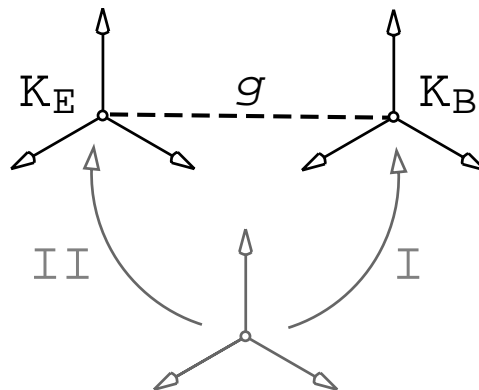


Figure 5.7: Basic form of a scalar measurement

Scalar measurements can exhibit almost any combination of geometrical, topological and activity type currently supplied with the MOBILE software. Only the “segment assembly” topological measurement type always requires that the measurement is self-reconfiguring. MOBILE currently supports five types of scalar geometric measurements, all of which arise from the combinations of the two geometric elements “*point*” and “*plane*”:

1. The **quadratic distance** between the origins of two frames. This class is suited for formulating constraints between two spherical joints, or a Hooke joint and a spherical joint (class name: MoChordPointPointQuadratic).
2. The **linear distance** between the origins of two frames. This class is suited for generating linear springs, but not so well-suited for constraint formulation due to its poor computational performance (class name: MoChordPointPointLinear).
3. the **cosine of the angle** between two coordinate planes. This class can be employed for resolution of variables in a spherical joint (class name: MoChordPlanePlane).
4. The **distance from a point to a plane**, where the point is located at the origin of frame \mathcal{K}_E and the plane is coplanar to a coordinate plane of frame \mathcal{K}_B . This measurement is suited for resolving angles at Hooke joints when the second joint is a spherical joint; one chooses as normal vector the unit vector in direction of the second axis of the Hooke joint and as point the center of the spherical joint, eliminating by this the second joint variable of the Hooke joint and the three joint variables of the spherical joint (class name: MoChordPointPlane).
5. The **shortest distance from a plane to a point**, where the plane is now a coordinate plane of frame \mathcal{K}_E and the point is the origin of frame \mathcal{K}_B . This

measure is suited for resolution of angles at a Hooke joint when the second joint is a planar joint; note that the distribution of the geometric elements “point” and “plane” to the two involved frames is now reversed in comparison to the previous measure (class name: `MoChordPlanePoint`).

Fig. 5.8 illustrates the two geometrical types of measurements most used in multibody analysis.

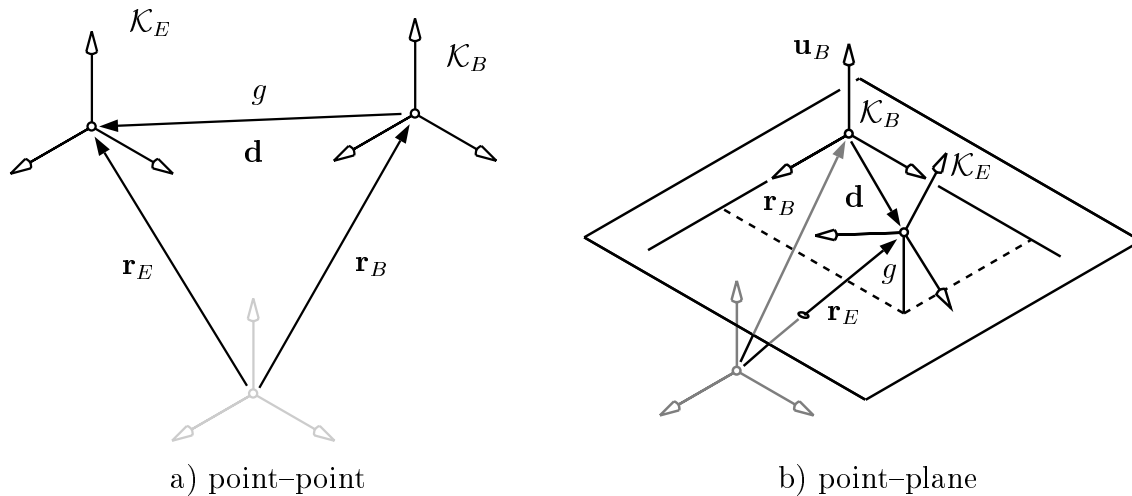


Figure 5.8: Geometric entities involved in the measurements between points and planes

Table 5.5 summarizes the scalar geometric measurements and the underlying measurement expressions at position level. In these expressions, \mathbf{r}_B and \mathbf{r}_E denote the radius vectors to the origins of the reference frames \mathcal{K}_B and \mathcal{K}_E as measured from the inertial system, respectively, and \mathbf{R}_B and \mathbf{R}_E are the corresponding transformation matrices from the reference frames to the inertial system. The vector \mathbf{u}_B is a unit vector normal to the plane involved in the measurement. In `MOBILE`, only coordinate planes are allowed in measurements. Thus, unit vectors can have only one of the three values `xAxis`, `yAxis` and `zAxis`.

MoChord...	geom. entity \mathcal{K}_B	geom. entity \mathcal{K}_E	expression
PointPlane	<i>coordinate plane</i>	<i>origin</i>	$(\mathbf{R}_E \mathbf{r}_E - \mathbf{R}_B \mathbf{r}_B) \mathbf{R}_B \mathbf{u}_B$
PointPointQuadratic	<i>origin</i>	<i>origin</i>	$\ \mathbf{R}_E \mathbf{r}_E - \mathbf{R}_B \mathbf{r}_B\ ^2$
PointPointLinear	<i>origin</i>	<i>origin</i>	$\ \mathbf{R}_E \mathbf{r}_E - \mathbf{R}_B \mathbf{r}_B\ $
PlanePoint	<i>coordinate plane</i>	<i>origin</i>	$(\mathbf{R}_B \mathbf{r}_B - \mathbf{R}_E \mathbf{r}_E) \mathbf{R}_E \mathbf{u}_E$
PlanePlane	<i>origin</i>	<i>origin</i>	$(\mathbf{R}_B \mathbf{u}_B) \cdot (\mathbf{R}_E \mathbf{u}_E)$

Table 5.5: Basic geometric types of scalar measurements

5.2.6 Constructing Measurements with Different Numbers of Frames

The geometrical types of measurements described above can be applied to between one and four reference frames. This yields the *topological type* of the measurement.

The basic entities and notations arising in the definition of the different topological types of measurement shall be first described based on the most general topological type of measurement, the type (IV) measurement depicted in Fig. 5.9. For this measurement, the loop is cut apart into *two* segments. One of the segments is denoted the “*lower*” segment and the other the “*upper*” segment, the upper segment being the one that is *not* connected to the inertial frame. The measurement consists in taking homolog measurements g' and g in the upper and lower segments, respectively, and subtracting these two values. The upper segment is assumed to be modelled by a transmission chain denoted by ϕ' . This chain starts at the upper base frame \mathcal{K}'_B and ends at the target frame \mathcal{K}'_E . The two cut frames of the lower segment are denoted accordingly as \mathcal{K}_B and \mathcal{K}_E , where \mathcal{K}_B matches \mathcal{K}'_B and \mathcal{K}_E matches \mathcal{K}'_E at the respective cuts. The segment cut introduces a total of three branches that are of interest for the subsequent processing of the loop kinematics. These branches have the following meaning:

- **Branch “I”** leads to the base frame \mathcal{K}_B of the lower segment
- **Branch “II”** leads to the end frame \mathcal{K}_E of the lower segment
- **Branch “III”** leads from base frame \mathcal{K}'_B to end frame \mathcal{K}'_E of the upper segment

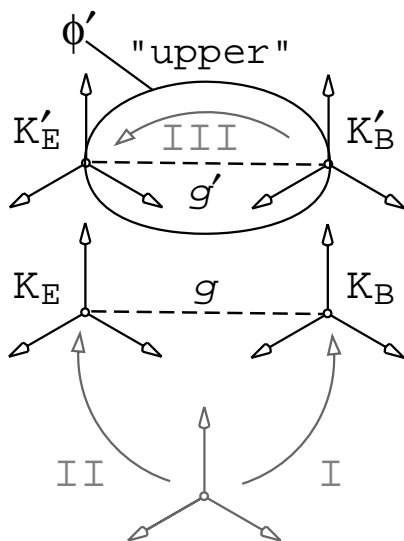


Figure 5.9: Entities of interest for the topological types of measurement

An overview of the currently supported topological types of measurement is given in Table 5.6 and Fig. 5.10. The notation “ $g(\mathcal{K}_X, \mathcal{K}_Y)$ ” in Table 5.6 denotes a scalar measurement between any two frames \mathcal{K}_X and \mathcal{K}_Y , while the frame \mathcal{K}_0 represents the inertial frame. The second column of Fig. 5.10 represents the case in which the kinetostatics of

branches “I” and “II” are passed to the measurement as a transmission chain ϕ . This corresponds to the case of a self-reconfiguring measurement as explained in Section 5.2.2.

type	arguments	measurement	notes
(I)	\mathcal{K}, y	$g(\mathcal{K}_0, \mathcal{K}) - y$	absolute motion of reference frame \mathcal{K} minus scalar y
(II)	$\mathcal{K}_E, \mathcal{K}_B, y$	$g(\mathcal{K}_B, \mathcal{K}_E) - y$	relative motion of \mathcal{K}_E with respect to \mathcal{K}_B minus scalar y
(III)	$\mathcal{K}'_E, \mathcal{K}_E$	$g(\mathcal{K}'_E, \mathcal{K}_0) - g(\mathcal{K}_E, \mathcal{K}_0)$	difference of measurements of absolute motion of two frames \mathcal{K}_E and \mathcal{K}'_E
(IV)	$\mathcal{K}'_E, \phi', \mathcal{K}_E, \mathcal{K}_B$	$g'(\mathcal{K}'_E, \mathcal{K}'_E) - g(\mathcal{K}_B, \mathcal{K}_E)$	difference of two measurements, one between the start and endpoint of a moving chain (branch “III”), and one based on the relative motion between two reference frames of the other segment; the argument “ ϕ' ” represents the transmission function of the moving chain (see below)

Table 5.6: Types of measurements involving different numbers of frames

The topological types of measurements described in Table 5.6 and Fig. 5.10 are applicable for the following tasks:

- **Absolute Motion — Type (I).** These objects project the absolute motion of a moving frame to a real number and subtract from it a constant y . One can use these objects for example for establishing the height of an object over a coordinate plane, or to compute the radius of a particle moving around a pole. By the variable y , which is mandatory, one can establish an offset which is subtracted each time the measurement is carried out. Such an offset can be for example of use for specifying an unloaded spring length, for describing the constraint equation of a particle moving on the surface of a sphere, etc.
- **Relative Motion — Type (II).** This is the “classical” topological measurement type. It projects the relative motion of a reference frame \mathcal{K}_E with respect to another reference frame \mathcal{K}_B to a scalar number. As with type (I), a constant y has to be supplied that is subtracted from that number. Objects of this type are useful for describing constraints arising from massless rods or couplers. Also, these objects serve as a basis for elementary force elements like springs, dampers, etc.
- **Absolute Difference — Type (III).** Objects of this kind compute a scalar number by subtracting the measurement obtained from the absolute motion of one moving frame \mathcal{K}'_E from a corresponding measurement (of the same type) taken for a second frame \mathcal{K}_E . Both measurements are taken with respect to the inertial frame. Objects of this type are useful when one chain is to follow the motion of another. For example, one can let one robot prescribe the desired height of an object and ask another to achieve this height by an appropriate control. The control is then taken over by a constraint solver.

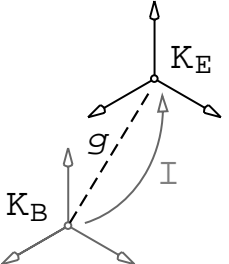
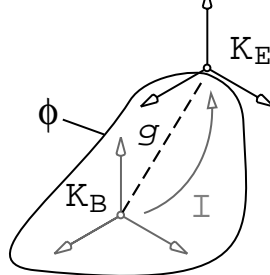
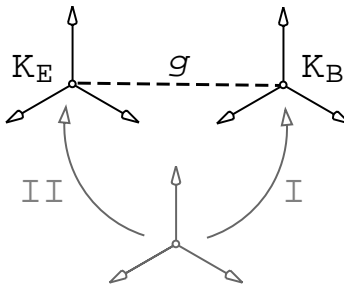
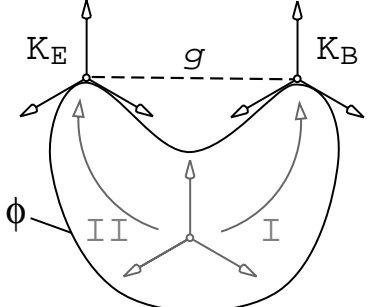
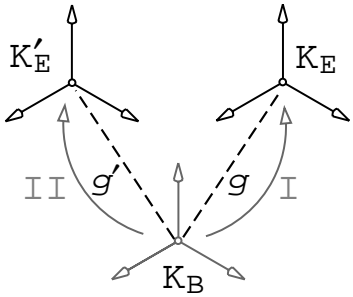
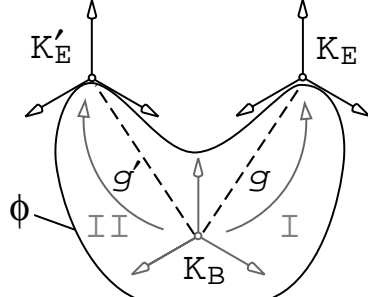
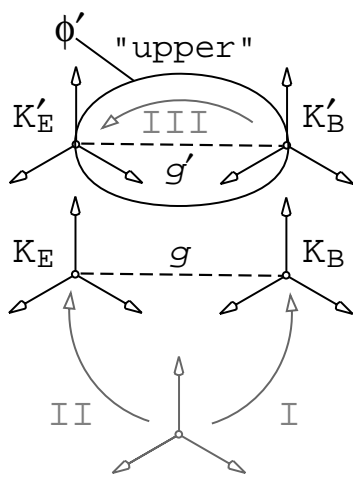
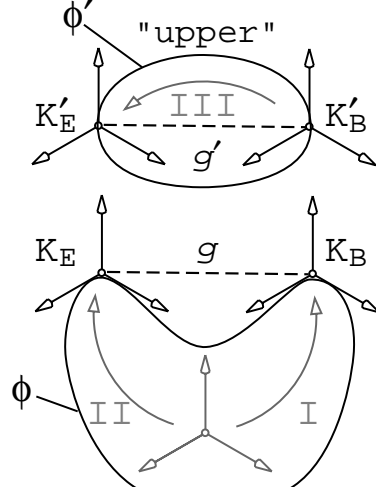
Type	simple measurement	reconfiguring measurement
<p>absolute motion (Type I)</p> $f = g - y$		
<p>relative motion (Type II)</p> $f = g - y$		
<p>absolute difference (Type III)</p> $f = g' - g$		
<p>relative difference (Type IV)</p> $f = g' - g$		

Figure 5.10: Types of measurements based on number of frames

- **Relative Difference — Type (IV)**. This is the most involved topological measurement type. The measurement corresponds to the difference of two relative measurements, one with respect to the “upper” segment, and one with respect to the “lower” segment. As this upper segment is completely isolated from the system, the measurement object needs access to the chain describing its kinetostatics. Thus, the constructor of this type of objects always takes an additional argument of type `MoMap` embodying this isolated chain. Objects of this type are useful for establishing triangular systems of constraint equations which are recursively solvable. Such cases can be found in abundance in technical systems, but the corresponding methodology is very complex and not suited for casual users. Thus, these objects are not intended for users who are seeking rapid-prototyping solutions to their problems. Such users may skip the following sections and go right to Section 5.3, where the solution of constraint equations is described. The use of scalar measurements of type (IV) for efficient solution of constraint equations will be further discussed in Section 5.3.2.

The following code fragment illustrates the initialization of measurement objects for the four topological types described above. As an example, the linear measurement between two points is employed. However, any other scalar measurement could have been employed instead. Note that spatial measurements support only constructors with two reference frames.

```

MoFrame KE , KB , KEprime ;
MoMapChain phi, phiPrime ;
MoLinearVariable y ;

// Type I
MoChordPointPointLinear chord_Ia ( KE , y ) ;
MoChordPointPointLinear chord_Ib ( KE , y , phi ) ;

// Type II
MoChordPointPointLinear chord_IIa ( KE , KB , y ) ;
MoChordPointPointLinear chord_IIb ( KE , KB , y , phi ) ;

// Type III
MoChordPointPointLinear chord_IIIa ( KEprime , KE ) ;
MoChordPointPointLinear chord_IIIb ( KEprime , KE , phi ) ;

// Type IV
MoChordPointPointLinear chord_IVa ( KEprime , phiPrime , KE , KB ) ;
MoChordPointPointLinear chord_IVb ( KEprime , phiPrime , KE , KB , phi ) ;

```

Note that it is not necessary to pass the base frame of the upper segment, KB' , to the chord. This frame is extracted internally from the transmission element *coupler* during initialization of the object `chord`.

5.2.7 Optimizing Performance by Specification of Active Branches

Measurement objects provide some rudimentary mechanisms for optimizing performance in the calculation of their outputs. The basic idea is to tell the chord at which of the

possibly three existing branches I, II, III it is necessary to apply an action, as e.g. motion transmission or force application. For example, if the base frame \mathcal{K}_B is kept fixed (though not necessary congruent to the inertial frame), the measurement object does not need to upgrade that information and it can reuse the values computed at initialization time. Optimization information does not have any effect on the accuracy of the results, but can lead to up to 20% of improvement in execution time during solution of the constraint equations. Users not concerned with computational performance issues can skip this section.

Optimization information is passed to the measurement object by up three optional arguments appended to the normal list of arguments:

- `MoChord...` (`...` , `whereUnknown`, `whereForce`, `whereInput`);

The functionality of these arguments is described in Table 5.7.

variable name	informs about ...
<code>whereUnknown</code>	which branches contain <i>unknowns</i> ; motion is evaluated and test forces are applied (during computation of the Jacobian) only at the tips of these branches
<code>whereForce</code>	for which branches is <i>force traversal required</i> ; this corresponds to the branches containing unknowns as well as those containing generalized coordinates (for statics and dynamics calculations)
<code>whereInput</code>	which branches contain <i>input motions</i> , but not unknowns; motion is evaluated for these branches at their tips only <i>once</i> at the start of each iteration

Table 5.7: Optional parameters for performance optimization of measurement objects

The possible values for the optimization parameters are

- `DO_BRANCH_I`, do the calculations for branch “I”
- `DO_BRANCH_II`, do the calculations for branch “II”
- `DO_BRANCH_III`, do the calculations for branch “III”
- `DO_ALL_BRANCHES`, do the calculations for all branches (**default**)

If a variable is omitted, it is given the default value `DO_ALL_BRANCHES`. Note that it is not possible to omit an optimization parameter if one further to the right is to be prescribed. Note also that, by prescribing a non-default value, the measurement will do *less* work than if the variable is omitted. However, the user’s prescriptions are not checked for inconsistency. For this reason, it is not recommended to use the optimization parameters

before a reference simulation has been carried out. Omitting all of optimization parameters will simply imply redundant calculations, but no errors will be incurred. Users not familiar with the multibody systems need not to be concerned with these issues.

The following code fragment illustrates a possible optimization setting for a linear point-to-point measurement of type IV:

```
MoFrame KB ; // motion of this frame depends on input variable
MoFrame KE ; // motion of this frame depends on dependent variable
MoFrame KBp , KEp ; // terminal frames of the upper segment
MoVector d ;
MoRigidLink coupler ( KBp , KEp , d ) ; // upper segment
MoChordPointPointLinear chord ( KEp , coupler , KE , KB ,
    DO_BRANCH_II , DO_BRANCH_I | DO_BRANCH_II , DO_BRANCH_I ) ;
```

This constructor tells the measurement object “chord” that the dependent variable is situated somewhere below reference frame KE (because of *whereUnknown*=DO_BRANCH_II), that the forces are requested for frames KB and KE (because of *whereForce*=DO_BRANCH_I | DO_BRANCH_II), and that input motion will only occur within the transmission chain below KB (because of *whereInput*=DO_BRANCH_I).

5.2.8 Interlinking Measurements

In the treatment of closed loops it may happen that the complete system of constraint equations can be decomposed into a cascade of scalar equations, each holding exactly one unknown more than its predecessor, and each being solvable in closed form for that unknown. M□BILE offers a technique for treating such situations explicitly. This technique works only for measurements of type IV above.

The basic idea for solving a cascade of constraint equations is to pass the previous measurement objects to the measurement object of a newly introduced equation of the cascade. The first equation of this cascade is called the “core” equation, and the rest are termed the “complementary” equations. In progressing from the core equation to the first complementary equation, the situation shown in Fig. 5.11 will result. In this figure, it is supposed that a first measurement was performed as the difference of relative motion between the terminal frames \mathcal{K}'_E and \mathcal{K}'_B of the upper segment and the terminal frame \mathcal{K}_E and the previous base frame prev- \mathcal{K}_B of the lower segment. It is now assumed that the next unknown to be determined occurs somewhere between the previous base frame prev- \mathcal{K}_B and the new base frame \mathcal{K}_B of the lower segment.

Let ϕ denote the transmission chain that connects the previous lower base frame prev- \mathcal{K}_B to the new lower base frame \mathcal{K}_B , such that it contains exactly one new unknown, and that there are no additional unknowns between \mathcal{K}_B and \mathcal{K}'_E . Then, one can take a new measurement which has the same cut frames \mathcal{K}'_E , \mathcal{K}'_B and \mathcal{K}_E as the core measurement, but instead of the previous lower base frame prev- \mathcal{K}_B the new lower base frame \mathcal{K}_B . This measurement is then employed to determine the new unknown. M□BILE allows to model this situation by a constructor of type

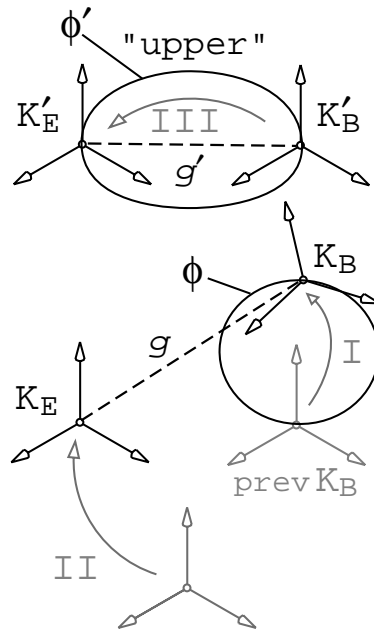


Figure 5.11: Measurement Object for Complementary Variable

- `MoChord... (MoFrame KB , MoChord prev , MoMap phi , ...) ;`

where “prev” is the previous measurement, “KB” is the new lower base frame, and “phi” is the transmission chain connecting the previous lower base frame `oldKB` with the new lower base frame `KB`.

If only one unknown remains in the chain between \mathcal{K}_B and \mathcal{K}'_B , one can repeat two measurements between the four frames depicted in Fig. 5.11, but with different geometric elements than the previous one. `MOBILE` allows the user to model this situation by making use of a constructor of type

- `MoChord... (MoChord prev , ...) ;`

The only case in which such a situation occurs, is when the remaining unknown is an angular rotation, and the measurements to be performed are of the geometric type *point-plane*. Then, the two measurements to be taken are the distances of the origin of frame \mathcal{K}_E or \mathcal{K}'_E to the two planes *parallel* to the rotation axis of the joint containing the new unknown. The user must take care that the order of this measurements is such that the vector product of the normals of these two planes is equal to the unit vector in direction of the axis of action of the angular variable, i.e. if the unknown rotation is about the y -axis, the first measurement should be with respect to the plane normal to the z -axis and the second with respect to the plane normal to the x -axis.

An example of this type of solution displayed in Section 5.4.3.

5.3 Objects for Solving Constraints

Constraint solving objects in M□BILE can process the kinetostatics of one or more closed loops. In order to keep the loops closed, a solver object needs three pieces of information: (i) the measurements whose vanishing will signal the closure of the loops, (ii) the dependent variables whose variation will lead to the closure of the loop and (iii) the dependent chain that will reconfigure the cut frames involved in the measurements after perturbing the dependent variables. Currently, there are two types of solver objects installed:

- (A) **explicit solvers**, which can resolve a scalar constraint equation explicitly in terms of one unknown (class name: `MoExplicitConstraintSolver`, and
- (A) **implicit solvers**, which can resolve any number of constraint equation iteratively for a set of unknowns (class name: `MoImplicitConstraintSolver`).

As was explained in Section 5.1, there exist three basic methods for establishing constraint equations in closed loops:

- (C1) cutting the loop at one body
- (C2) cutting the loop at one joint
- (C3) cutting the loop at two joints

For Methods (C1) and (C2), only implicit solvers are suitable. For Method (C3), there are some cases in which explicit solvers can be applied. These cases are characterized by the fact that by taking the measurement all but one of the unknowns of a loop can be eliminated. If this is not possible, i. e., if more than one dependent variable remains in the measurement, then set of measurements containing the same number of unknowns must be established and an implicit solver has to be used that solves the corresponding equations simultaneously. This happens for example when several loops are coupled yielding a set of scalar measurements that form a system of coupled nonlinear constraint equations. In this case, one collects the corresponding measurement objects in a chord list and solve the complete set of equations by one implicit solver.

Solvers of constraint equations behave like any kinetostatic transmission elements, i. e., they supply a motion and a force transmission function. The motion transmission function consists in establishing (and carrying out) the motion of the dependent chain such that the loop stays closed. The force transmission function involves the computation of the constraint forces within the loops and their propagation within the dependent chain such that static equilibrium is achieved.

Below the basic structure of the constraint solvers and their linkage with the measurement objects are explained. Applications of solver objects to particular loops are displayed in Section 5.4.

5.3.1 Implicit Solvers

The simpler method for resolving constraints is to use implicit or iterative solution schemes. In this case, one just has to gather a set of measurement objects describing the closure conditions of the loop(s), and pass it to the solver together with a list of unknown variables and, optionally, a transmission element.

The possible initializations of implicit solvers of constraint equations are

- `MoImplicitConstraintSolver (MoChord&, MoVariableList&) ;`
This case is suitable when the measurement is of self-reconfiguring type, i. e., when it contains the dependent chain mapping the values of the dependent variables to the cut frames involved in the measurement. Such solvers can be used for closure conditions of type (C1) or (C2) described above.
- `MoImplicitConstraintSolver (MoChord&, MoVariableList&, MoMap&) ;`
This case is equivalent to the first one, only that now the dependent chain is passed explicitly to the solver.
- `MoImplicitConstraintSolver (MoChordList&, MoVariableList&, MoMap&) ;`
This case is applicable to sets of constraint equations gathered at several places in the mechanism, and which are to be solved simultaneously.

Note that the number of variables in the variable list must match exactly the number of scalar measurements contained in the chord or the chord list passed to the solver. Otherwise, the system of equations would be either over- or underdetermined, and no solution could be determined. The number of scalar variables involved in spatial measurement objects can be hereby determined from Table 5.4.

5.3.2 Explicit Solvers

Explicit solvers are applicable only when a single constraint equation contains only one single unknown. The general syntax for the definition of an explicit solver is

- `MoExplicitConstraintSolver (MoChord&, MoLinearVariable&) ;`
- `MoExplicitConstraintSolver (MoChord&, MoAngularVariable&) ;`

In both cases, a scalar measurement object must be passed that depends only on *one* unknown, namely, the dependent variable passed as second parameter. This measurement object must be of the self-reconfiguring type, i. e. it must comprise the dependent chain mapping the dependent variable to the cut frames of the measurement. The type of the second parameter determines which algorithm the constraint solver applies to the resolution of the constraint.

When two measurements are taken for one unknown, it is possible to determine uniquely a solution. The corresponding constructor takes on the form:

```
MoExplicitConstraintSolver ( MoChord&, MoChord&, MoAngularVariable& );
```

This type of solver is only needed for example for determining the second complementary angle at a hook joint.

For a description of the usage of the solver objects, the reader is referred to the examples of the next section and the introductory example of Section 5.1.1.

5.4 Examples

Below, three examples of loop closure formulation and solution are supplied. The examples cover the three basic methods described previously for stating constraints, namely, (C1) body assembly, (C2) joint assembly and (C3) segment assembly. Apart from the pure kinematic modeling, the programs generate also the dynamical equations and integrate them using the built-in numerical integrators. The objects for generation and solution of dynamical equations shall be discussed in the next chapter.

5.4.1 Body Assembly of a Spatial Four-bar Mechanism

```
#include <Mobile/MoSphericalJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoImplicitConstraintSolver.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoAdamsIntegrator.h>

main()
{
// reference frames
// =====

MoFrame K0, K1, K2, K3, K4, K5, K6, K7, K8, K9, K10 ;

// masses
// =====

MoReal      m_d ;
MoInertiaTensor THETA_d ;

// state-variables (angles)
// =====

MoAngularVariable theta_1, theta_2, theta_3, theta_4, theta_5,
                  theta_6, theta_7;

// declare joints and connect reference frames
// =====
```

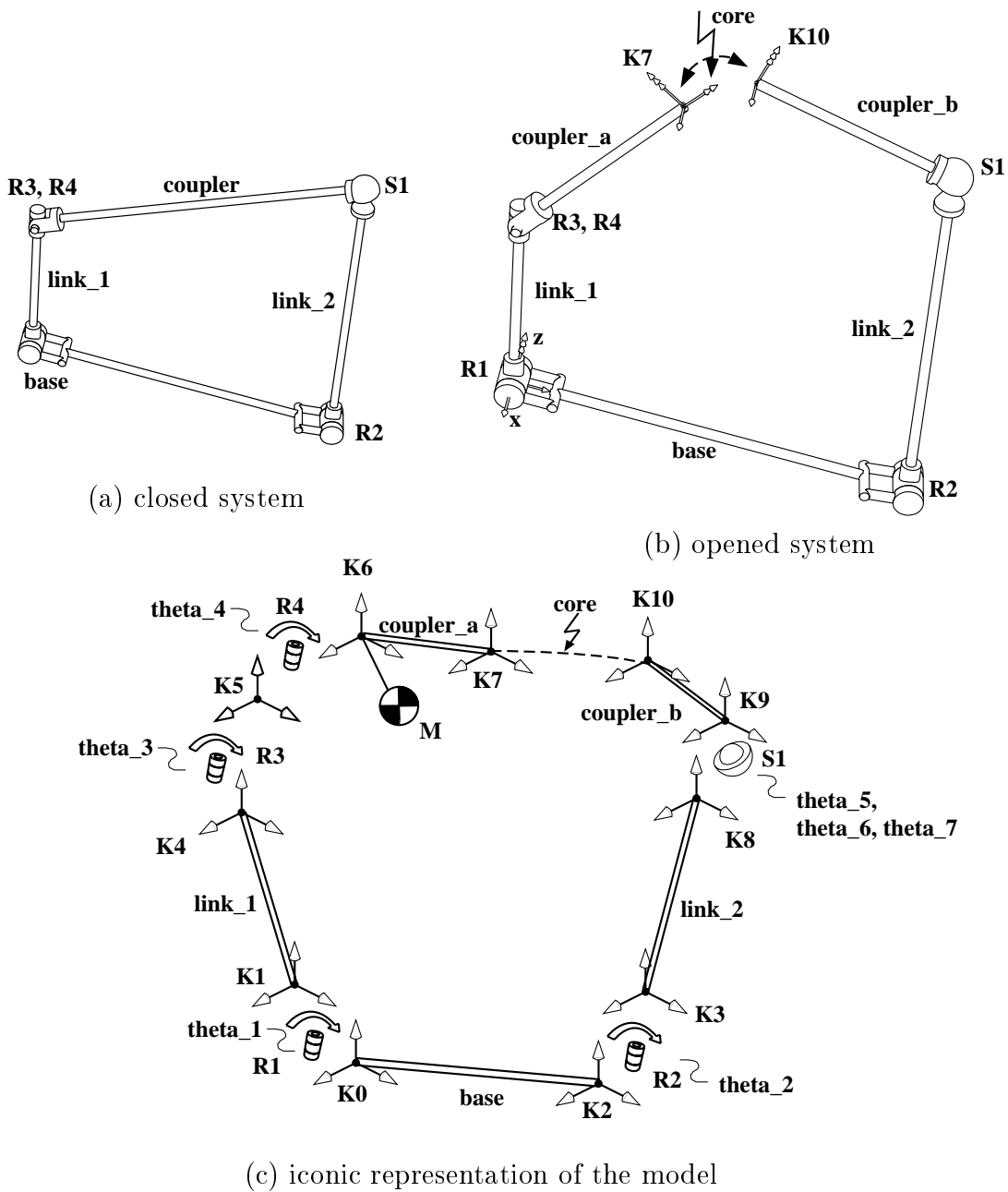


Figure 5.12: Modelling of the spatial Four-bar mechanism

```

MoElementaryJoint R1 ( K0 , K1 , theta_1 , xAxis );
MoElementaryJoint R2 ( K2 , K3 , theta_2 , xAxis );
MoElementaryJoint R3 ( K4 , K5 , theta_3 , zAxis );
MoElementaryJoint R4 ( K5 , K6 , theta_4 , xAxis );

MoVariableList CardanAngles ;
CardanAngles << theta_5 << theta_6 << theta_7 ;

MoSphericalJoint S1 ( K8 , K9 , CardanAngles , BRYANT_ANGLES );
// link-transformations
// =====

```

```

MoVector r, l, s, d_a, d_b;
r = l = s = d_a = d_b = MoNullState;

MoRotationMatrix A;
A = MoNullState;

// declare links for rigid-body transformations
// =====

MoRigidLink base      ( K0 , K2 , l , A );
MoRigidLink link_1    ( K1 , K4 , r );
MoRigidLink link_2    ( K3 , K8 , s );
MoRigidLink coupler_a ( K6 , K7 , d_a );
MoRigidLink coupler_b ( K9 , K10 , d_b );

// subsystems for solution of constraints
// =====

MoMapChain DependentChain, Dp2 , RightBranch;

RightBranch << R1 << link_1 ;
DependentChain << R2 << link_2 << R3 << R4
               << coupler_a << S1 << coupler_b ;

// mass elements
// =====

MoMassElement CouplerBody ( K6 , m_d , THETA_d , d_a ) ;

// declare constraints
// =====

MoChord3DPose core ( K10 ,           // left
                    K7 ,           // right
                    DO_ALL_BRANCHES , // where unknown
                    DO_BRANCH_I ) ; // where force

// declare implicit solver for constraint equations
// =====

MoVariableList dependents ;
dependents << theta_2 << theta_3 << theta_4 << CardanAngles ;

MoImplicitConstraintSolver CoreSolver ( core, dependents, DependentChain );

// declare a subsystem for the full solution of the loop
// =====

MoMapChain SolveAll ;

SolveAll << base           // base body

```

```

        << RightBranch      // move the input crank
        << CoreSolver       // solve the implicit core
        << CouplerBody ;    // compute D'Alembert's forces at coupler

// list of independent coordinates
// =====

MoVariableList  q ;
q << theta_1 ;

// define parameter values
// =====

l.y  = 2.0 ;           // offset at the base
r.z  = 1.0 ;           // length of link 1
s.z  = 2.0 ;           // length of link 2
d_a.z = -0.5*sqrt(5.0) ; // half length of coupler
d_b.z = 0.5*sqrt(5.0) ; // half length of coupler

m_d    = 1.0 ;         // mass of coupler
THETA_d = 1.0 ;       // symmetric inertia tensor for coupler

// generation of equations of motion and integration
// =====

MoMechanicalSystem SystemDynamic ( q , SolveAll , K0 , zAxis ) ;

MoAdamsIntegrator SystemIntegrator( SystemDynamic ) ;
MoReal dt = 0.01 ;
SystemIntegrator.setTimeInterval( dt ) ;

// initial conditions
// =====

theta_1.q  = 1.0 * DEG_TO_RAD;

theta_2.q = DEG_TO_RAD * 0.0 ;
theta_3.q = DEG_TO_RAD * 0.0 ;
theta_4.q = DEG_TO_RAD * 40.0 ;
theta_5.q = DEG_TO_RAD * 20.0 ;
theta_6.q = DEG_TO_RAD * 0.0 ;
theta_7.q = DEG_TO_RAD * 0.0 ;

// run the simulation
// =====

for ( int i = 0 ; i++ < 100 ; )
    SystemIntegrator.doMotion() ;
}

```

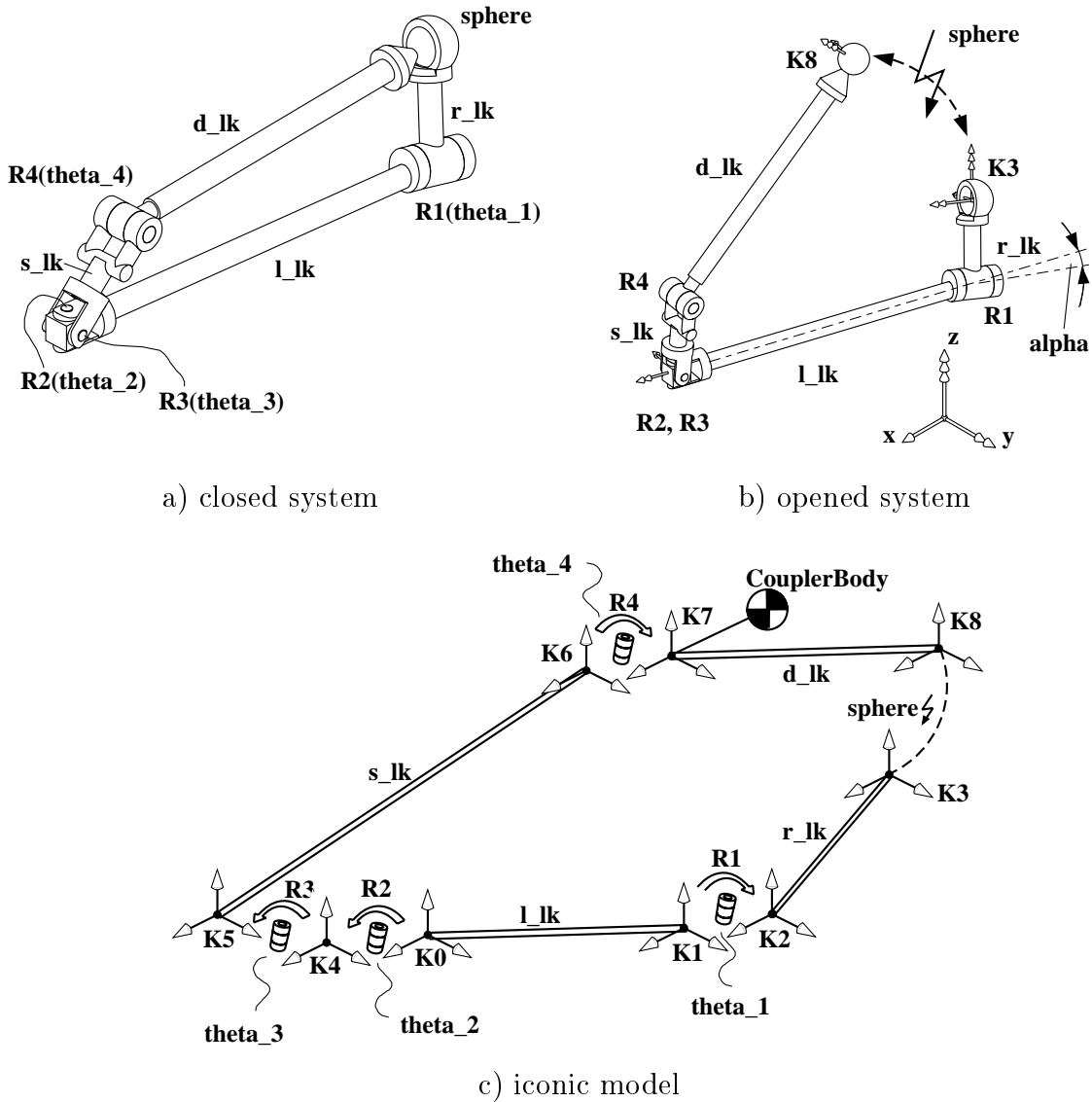



Figure 5.13: Modeling of the Dynamics of a Shaker Mechanism

5.4.2 Joint Assembly of a Shaker Mechanism

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoImplicitConstraintSolver.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoAdamsIntegrator.h>

main()
{
// reference frames
// =====

MoFrame    K0 , K1 , K2 , K3 , K4 , K5 , K6 , K7 , K8 ;
```

```

// state-variables
// =====

MoAngularVariable theta_1 , theta_2 , theta_3 , theta_4 ;

// joints
// =====

MoElementaryJoint R1 ( K1 , K2 , theta_1 , yAxis ) ;
MoElementaryJoint R2 ( K0 , K4 , theta_2 , zAxis ) ;
MoElementaryJoint R3 ( K4 , K5 , theta_3 , xAxis ) ;
MoElementaryJoint R4 ( K6 , K7 , theta_4 , xAxis ) ;

// links connecting joints
// =====

MoVector r , l , s , d , d_s ;
MoRotationMatrix A ;

MoRigidLink l_lk ( K0 , K1 , l , A ) ;
MoRigidLink r_lk ( K2 , K3 , r ) ;
MoRigidLink s_lk ( K5 , K6 , s ) ;
MoRigidLink d_lk ( K7 , K8 , d ) ;

// subsystem to be iterated while solving for constraints
// =====

MoMapChain DependentChain ;
DependentChain << R2 << R3 << s_lk << R4 << d_lk ;

// constraint
// =====

MoChord3DPosition sphere ( K8 , K3 ) ;

// implicit solver for constraint equations
// =====

MoVariableList dependents ;
dependents << theta_2 << theta_3 << theta_4 ;

MoImplicitConstraintSolver Solver( sphere , dependents , DependentChain ) ;

// mass element
// =====

MoReal m_d ;
MoInertiaTensor THETA_d ;

MoMassElement CouplerBody ( K7 , m_d , THETA_d , d_s ) ;

```

```

// create a subsystem for applying the input motion
// =====

MoMapChain input ;
input << l_lk << R1 << r_lk ;

// create a map chain for the complete kinematics of the loop
// =====

MoMapChain SolveAll ;
SolveAll << input << Solver << CouplerBody ;

// list of independent coordinates
// =====

MoVariableList q ;
q << theta_1 ;

// generation of equations of motion and integration
// =====

MoMechanicalSystem SystemDynamic ( q , SolveAll , KO , zAxis ) ;

MoAdamsIntegrator SystemIntegrator( SystemDynamic ) ;
MoReal dt = 0.1 ;
SystemIntegrator.setTimeInterval( dt ) ;

// geometry
// =====

r = l = s = d = d_s = MoNullState ;
MoReal alpha = DEG_TO_RAD * 10.0;
MoXRotation X = alpha;
A = X;
l.y = -0.8 ; // offset at the base
r.z = 0.2 ; // length of r_lk
s.z = 0.2 ; // length of s_lk
d.y = -0.7 ; // length of coupler
d_s.y = -0.5*0.7 ; // position of center of mass

// masses
// =====

MoVector theta(4.08895833e-2 , 1.125e-4 , 4.08895833e-2);
m_d = 1.0 ;
THETA_d = theta;

// initial conditions
// =====

theta_1.q = 1.0 * DEG_TO_RAD;
// run the simulation
// =====

```

```

for ( int i = 0 ; i++ < 100 ; )
    SystemIntegrator.doMotion() ;
}

```

5.4.3 Segment Assembly of a Shaker Mechanism

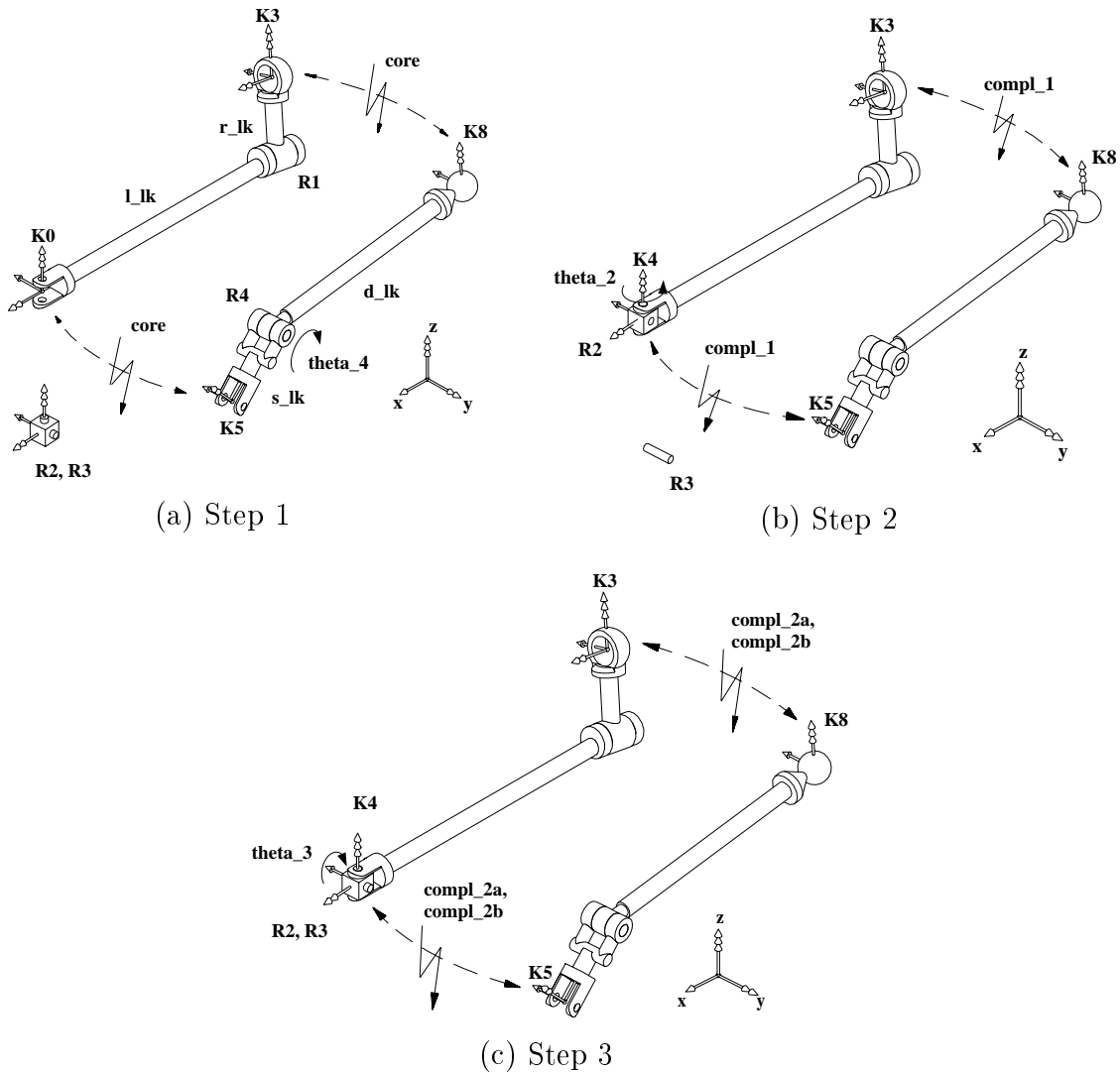


Figure 5.14: Modelling of the Shaker (explicit solution)

```

#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoExplicitConstraintSolver.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoAdamsIntegrator.h>

```

```
main()
```

```

{
// reference frames
// =====

    MoFrame    K0 , K1 , K2 , K3 , K4 , K5 , K6 , K7 , K8 ;

// state-variables
// =====

    MoAngularVariable theta_1 , theta_2 , theta_3 , theta_4 ;

// joints
// =====

    MoElementaryJoint R1 ( K1 , K2 , theta_1 , yAxis ) ;
    MoElementaryJoint R2 ( K0 , K4 , theta_2 , zAxis ) ;
    MoElementaryJoint R3 ( K4 , K5 , theta_3 , xAxis ) ;
    MoElementaryJoint R4 ( K6 , K7 , theta_4 , xAxis ) ;

// links connecting joints
// =====

    MoVector r , l , s , d , d_s ;
    MoRotationMatrix A ;

    MoRigidLink l_lk ( K0 , K1 , l , A ) ;
    MoRigidLink r_lk ( K2 , K3 , r ) ;
    MoRigidLink s_lk ( K5 , K6 , s ) ;
    MoRigidLink d_lk ( K7 , K8 , d ) ;

// subsystem to be iterated while solving for constraints

    MoMapChain coupler ;
    coupler << s_lk << R4 << d_lk ;

// constraints
// =====

    MoChordPointPointQuadratic core ( K8 , // upper left
                                       coupler , // upper chain
                                       K3 , // lower left
                                       K0 , // lower right
                                       DO_BRANCH_III , // where unknown
                                       DO_BRANCH_II , // where force
                                       DO_BRANCH_I |
                                       DO_BRANCH_II ) ; // input

    MoChordPointPlane compl_1 ( K4 , // new lower right
                                core , // same as in core
                                R2 , // lower chain
                                xAxis , // normal of the plane
                                DO_BRANCH_I , // where unknown

```

```

DO_BRANCH_II |
DO_BRANCH_III , // where force
DO_BRANCH_III ) ; // input

MoChordPointPlane compl_2a ( K5 , // new lower right
core , // same as in compl_1
R3 , // lower chain
yAxis , // normal of the plane
DO_BRANCH_I , // where unknown
DO_BRANCH_II |
DO_BRANCH_III , // where force
DO_BRANCH_III ) ; // input

MoChordPointPlane compl_2b ( compl_2a, // all references as in
zAxis ); // "compl_2a"

// explicit solvers for constraint equations
// =====

MoExplicitConstraintSolver CoreSolver ( core , theta_4 ) ;
MoExplicitConstraintSolver FirstHooke ( compl_1, theta_2 ) ;
MoExplicitConstraintSolver SecondHooke ( compl_2a, compl_2b, theta_3 ) ;
FirstHooke.selectBranch();

// mass element
// =====

MoReal m_d ;
MoInertiaTensor THETA_d ;

MoMassElement CouplerBody ( K7 , m_d , THETA_d , d_s ) ;

// create a subsystem for applying the input motion
// =====

MoMapChain input ;

input << l_lk << R1 << r_lk ;

// create a map chain for the complete kinematics of the loop
// =====

MoMapChain SolveAll ;

SolveAll << input // move the input crank
<< CoreSolver // solve the implicit core
<< FirstHooke // solve the first hooke-angle
<< SecondHooke // solve the second hooke-angle
<< CouplerBody ; // compute D'Alembert's forces at coupler

// list of independent coordinates
// =====

```

```

MoVariableList  q ;
q << theta_1 ;

// generation of equations of motion and integration
// =====

MoMechanicalSystem SystemDynamic ( q , SolveAll , KO , zAxis ) ;

MoAdamsIntegrator SystemIntegrator( SystemDynamic ) ;
MoReal dt = 0.1 ;
SystemIntegrator.setTimeInterval( dt ) ;

// geometry
// =====

r = l = s = d = d_s = MoNullState ;
MoReal alpha = DEG_TO_RAD * 10.0;
MoXRotation X = alpha;
A = X;
l.y   = -0.8 ;           // offset at the base
r.z   = 0.2 ;           // lenght of lk 1
s.z   = 0.2 ;           // lenght of lk 2
d.y   = -0.7 ;          // lenght of coupler
d_s.y = -0.5*0.7 ;      // position of center of mass

// masses
// =====

MoVector theta(4.08895833e-2 , 1.125e-4 , 4.08895833e-2);
m_d     = 1.0 ;
THETA_d = theta;

// initial conditions
// =====

theta_1.q = 1.0 * DEG_TO_RAD;

// run the simulation
// =====

for ( int i = 0 ; i++ < 100 ; )
    SystemIntegrator.doMotion() ;
}

```

6 Generating and Solving Dynamic Equations

The generation and solution of dynamical is realized in M□BILE as a three-step process. These steps perform the following operations:

1. **generation of the equations of motion** from a kinetostatic transmission chain (class `MoEqmBuilder`)
2. **transformation of equations of motion into state-space form**, i. e., a system of ordinary first-order differential equations (class `MoDynamicSystem` and classes derived from it, e. g. `MoMechanicalSystem`)
3. **numerical integration of the differential equations** (class `MoIntegrator` and classes derived from it, e. g. `MoExplicitEulerIntegrator`, `MoAdamsIntegrator`, and `MoRungeKuttaIntegrator`)

Below the main properties of these classes are described.

6.1 The class `MoEqmBuilder`

Objects of class for `MoEqmBuilder` are responsible in M□BILE for generating the dynamical equations of mechanical systems. These dynamical equations are determined by computing then inverse dynamics of the system repeatedly with changing input values for velocities and accelerations. A short description of the underlying mathematics is included below for easier reference.

Theoretical background: Generation of the Equations of Motion

The transmission elements described in the previous chapters give a simple means of generating and transmitting motions and loads within a multibody system. Consider a mechanical system having f independent generalized coordinates $\underline{q} = [q_1, \dots, q_f]^T$, and let the transmission of these coordinates to the mass and the force elements be given by a transmission element φ_S denoted ‘global kinematics’. Let also the global kinematics be decomposed in a first part, denominated here the ‘kinematical subsystem’, which encloses the chains of links and joints of the system (including closed loops), and a second part where the set of mass and force elements are subsequentially defined as leaf elements (Fig. 6.1). In this way, the overall system is partitioned in a “skeleton” containing the pure kinematostatic transmission structure of the system and additional elements producing force and mass effects that act as “leaves” attached to the kinetostatic skeleton.

The concatenation of position, velocity, acceleration and force transmission functions of the global kinematics yields a function which maps the generalized coordinates and their time derivatives to a set of *residual* generalized forces $\underline{\bar{Q}}$ at the input of the global kinematics. This function, which represents the inverse dynamics $\varphi_S^{D^{-1}}$ of the system, has the structure

$$\underline{\bar{Q}} = \varphi_S^{D^{-1}}(\underline{q}, \underline{\dot{q}}, \underline{\ddot{q}}; \underline{W}^{(e)}; t) = -\mathbf{M}(\underline{q}; t) \underline{\ddot{q}} - \underline{\hat{Q}}(\underline{q}, \underline{\dot{q}}; \underline{W}^{(e)}; t) , \quad (6.1)$$

where $\underline{W}^{(e)}$ recollects all externally applied forces, and \mathbf{M} and $\underline{\hat{Q}}$ are the generalized mass matrix and the generalized applied forces, respectively. The residual forces can be used to generate \mathbf{M} and $\underline{\hat{Q}}$ by the following simplified procedure:

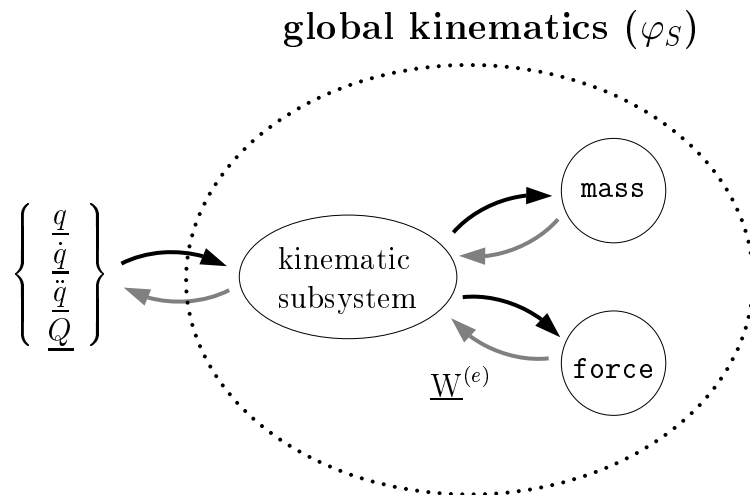


Figure 6.1: Model of the inverse dynamics of a multibody system.

\widehat{Q} : Set at the input of φ_S^{D-1} for the generalized accelerations $\underline{\ddot{q}} = 0$; then in Eq. (6.1) the term $\mathbf{M} \underline{\ddot{q}}$ vanishes and the residual vector obtained at the input is exactly \widehat{Q} .

M: Eliminate in the calculation of φ_S^{D-1} the term \widehat{Q} (this is simply done by ‘switching off’ effects arising from applied and generalized coriolis and centripetal forces), and set a single input acceleration $\ddot{q}_\nu = 1$ while all others vanish; then, the resulting force \overline{Q} is exactly the ν th column of the generalized mass matrix, and, by repeating this procedure for all columns, one obtains the complete mass matrix. Note that, in branched systems, only the subtree starting at the inertial frame and possessing all objects which are successors to q_ν needs to be calculated.

Generation of the equations of motion by this approach requires $f + 1$ traversals of the inverse dynamics for one set of equations, where f is the degree of freedom of the system. The number of evaluations of the inverse dynamics can be reduced when an iterative method is employed for solving the linear system of equations for the accelerations.

In order to accomplish the task of generating the dynamical equations, the object is initialized with the list of independent variables for which to generate the dynamic equations and the kinetostatic transmission chain mapping the motion of these variables to the motion of the reference frames at which mass or force elements are attached, as well as the mass and force elements themselves. Apart from this, the user can provide information about the direction of action of gravity as well as the reference frame acting as the inertial frame. Furthermore, one can give a map list containing for each degree of freedom the subtree which contains all components whose motion depends on this degree of freedom as well as the components places between the degree of freedom and the inertial system. This last list is only required when the user wants to optimize computational performance of the model. Depending on the type and number of arguments passed, the object will perform the activities described below.

```
MoEqmBuilder ( MoVariableList& q , MoMap& phi )
```

The object will generate the equations of motion for the variables contained in q , and using the transmission chain phi for establishing the generalized force vectors and the mass matrix. Gravity must be modeled through force elements.

MoEqmBuilder (MoVariableList& q , MoMapChain& phiM , MoMap& phi)

Same as above, only that now a list *phiM* of transmission elements is supplied in which each transmission element corresponds to the subtree which has to be traversed when establishing the corresponding columns of the mass matrix. Note that this list should always contain exactly the same number of elements as the variable list *q*.

MoEqmBuilder (MoVariableList& q , MoMap& phi , MoFrame& K , MoAxis = zAxis)

The object will generate the equations of motion for the variables contained in *q*, and using the transmission chain *phi* for establishing the generalized force vectors and the mass matrix. Gravity will be modeled by applying an appropriate acceleration to the reference frame *K* in the direction of the coordinate axis supplied as fourth parameter. This models gravity in opposite direction to the coordinate axis supplied as fourth parameter. If the fourth parameter is omitted, gravity will be applied in negative *z*-direction.

MoEqmBuilder

(MoVariableList& q , MoMapChain& phiM , MoMap& phi , MoFrame& K , MoAxis = zAxis)

Same as above, only that now a list *phiM* of transmission elements is supplied in which each transmission element corresponds to the subtree which has to be traversed when establishing the corresponding columns of the mass matrix. Note that this list should always contain exactly the same number of elements as the variable list *q*.

MoEqmBuilder (MoVariableList& q , MoMap& phi , MoFrame& K , MoVector& gravity)

The object will generate the equations of motion for the variables contained in *q*, and using the transmission chain *phi* for establishing the generalized force vectors and the mass matrix. Gravity will be modeled by applying an appropriate acceleration in direction of the vector *gravity* to the reference frame *K*. The magnitude of this vector is ignored.

MoEqmBuilder

(MoVariableList& q , MoMapChain& phiM , MoMap& phi , MoFrame& K , MoVector& gravity)

Same as above, only that now a list *phiM* of transmission elements is supplied in which each transmission element corresponds to the subtree which has to be traversed when establishing the corresponding columns of the mass matrix. Note that this list should always contain exactly the same number of elements as the variable list *q*.

The builders of dynamic equations are not kinetostatic transmission elements in the sense defined above. They thus do not support the transmission functions `doMotion()` and `doForce()`. Instead, one can invoke the functions described below. Normally, the user needs not to be concerned with these functions, because they are called internally by the objects described in the subsequent sections. They are required only the user needs to have direct information about the terms involved in the equations of motion.

buildEquations

This generates the equations of motion, putting the result in internal storage space.

saveEquations

Copies the system matrices generated by **buildEquations** to the user-accessible arrays. These arrays are

MoReal* MassMatrix

Symmetric matrix containing the generalized mass of the system. All coefficients are filled.

MoReal* ForceVector

Vector containing the difference of applied and generalized Coriolis forces.

solveEquations

Determines the system accelerations which are in equilibrium with the applied forces, and stores the result in the corresponding state subentries of the variables in the variable list q . A call to **buildEquations** must be performed prior to this invocation.

printMass

Prints the current values of the generalized mass matrix to standard output. A call to **buildEquations** must be performed prior to this invocation.

printForce

Prints the current values of the force vector defined above to standard output. A call to **buildEquations** must be performed prior to this invocation.

printAcceleration

Prints the current values of the resolved generalized accelerations to standard output. A call to **buildEquations** and **solveEquations**, in this order, must be performed prior to this invocation.

6.2 Generating Ordinary Differential Equations in State-Space Form

For the integration of the equations of motion, it is required that they are transformed to space-state form. The state-space form of a system of ordinary differential equations takes the form

$$\dot{\underline{y}} = \underline{f}(\underline{y}, t)$$

In MOBILE, objects that behave like systems of ordinary differential equations in state-space form are derived from the class **MoDynamicSystem**. The common property of these classes is the support of the following set of functions

```
int getOrder()
```

Return the number of state-space variables included in the dynamic system.

```
void giveYd( MoReal t , MoReal* y , MoReal* yd )
```

Evaluate the function f defined above for time t and state y , returning the result of the function, i.e. the time-derivative \dot{y} , in the array yd .

```
void giveActualConditions( MoReal* y )
```

Returns the actual values of the state variables in the array y .

```
void setActualConditions( MoReal* y )
```

Sets the actual values of the state variables in the array y .

The class `MoDynamicSystem` declares the overall behaviour of the object representing differential equations in state-space form. For the actual generation of these equations, dedicated classes have to be written. In `M□BILE`, there is currently only one class for doing this. This class is termed `MoMechanicalSystem` and transforms a system of differential equations of second order resulting from the equations of motion of a mechanical system to state-space form. Other classes that are conceivable may generate first order equations for other types of systems, such as hydraulic, electric or control systems. If needed, these classes must be currently defined by the user. By deriving them from the base class `MoDynamicSystem`, the user can combine them with other objects representing state-space form representations of dynamic equations and integrate them with the supplied numerical integrators.

6.2.1 The Class `MoMechanicalSystem`

Objects of class `MoMechanicalSystem` transform dynamic equations of mechanical systems into state-space form. Basically, an object of type `MoMechanicalSystem` needs to be passed the name of a builder of equations of motion for generating the corresponding state-space representation. However, `M□BILE` provides also a shortcut that allows the user to avoid the need of constructing an extra object of type `MoEqmBuilder`. This shortcut consists of passing to the object of type `MoMechanicalSystem` the same arguments as to the object of `MoEqmBuilder`. Thus, there exist the following constructors for objects of type `MoMechanicalSystem`.

```
MoMechanicalSystem ( MoEqmBuilder& sys )
```

The object generates the state-space form of the equations of motion established by the builder `sys`.

```
MoMechanicalSystem ( MoVariableList& q , MoMap& phi )
```

```
MoMechanicalSystem ( MoVariableList& q , MoMapChain& phiM , MoMap& phi )
```

```
MoMechanicalSystem ( MoVariableList& q , MoMap& phi , MoFrame& K , MoAxis = zAxis )
```

```
MoMechanicalSystem
```

```
( MoVariableList& q , MoMapChain& phiM , MoMap& phi , MoFrame& K , MoAxis = zAxis )
```

```
MoMechanicalSystem ( MoVariableList& q , MoMap& phi , MoFrame& K , MoVector& gravity )
```

```
MoMechanicalSystem
```

```
( MoVariableList& q , MoMapChain& phiM , MoMap& phi , MoFrame& K , MoVector& gravity )
```

These objects generate the state-space form of the dynamical equations of a mechanical system. The meaning of the arguments is identical to that described in Section 6.1.

6.3 Solving the Differential Equations

Once the equations of motion are available in state-space form, they can be passed to a numerical integrator. The numerical integrator is capable of moving along the solution trajectory of the system in prescribed time steps. In this setting, the integrator plays the role of a generalized joint that supports a “doMotion()” function. For this reason, integrators have been defined in M□BILE as derivations of the class MoMap for general kinetostatic transmission elements.

The base class for objects for integrating dynamical equations is MoIntegrator. Integrator objects are constructed simply by passing a dynamic system whose differential equations are to be integrated. The constructor is simply

```
MoIntegrator ( MoDynamicSystem& sys )
```

The object is then capable of solving the differential equations established by sys.

The currently implemented methods for integrator objects are

```
void doMotion()
```

Progresses along the solution trajectory by the time interval specified by the user via the function setTimeInterval. Note that this time interval is not necessarily the step size of the integrator, but the period after which a result is returned.

```
void doForce()
```

Void function; does nothing.

```
int getNumberOfSteps()
```

Return current number of already performed steps.

```
void setTimeInterval( MoReal& dt)
```

Set the time interval for the method doMotion described above to dt.

```
void setStartTime( MoReal& t)
```

Set the start time of the integration to *t*.

Particular implementations of integrators may make it necessary to extend these methods by further functions. Currently, the following integrator schemes are incorporated into the M□BILE software

MoExplicitEulerIntegrator

Solves the differential equations by the forward Euler method. Apart from the functions described above, the following methods and member data are supplied:

```
void reset()
```

Read the current state of the system into internal storage.

```
MoReal StepSize
```

Step size of the method. Note that this is not the interval of integration set by `setTimeInterval`.

MoAdamsIntegrator

Solves the differential equations by the Adams-Moulton method. Apart from the functions described above, the following methods and member data are supplied:

```
void setRelativeTolerancere( MoReal& reltol )
```

Set the relative tolerance of the integration to *reltol*.

MoRungeKuttaIntegrator

Solves the differential equations by the Runge-Kutta method. Apart from the functions described above, the following methods and member data are supplied:

```
void setRelativeTolerancere( MoReal& reltol )
```

Set the relative tolerance of the integration to *reltol*.

Note: this object works only with the NAG library installed.

6.4 Example: Dynamics of a Triple Pendulum

```
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoAdamsIntegrator.h>

main()
{
    // definition of the system

    MoFrame K0, K1, K2, K3, K4, K5 ;
```

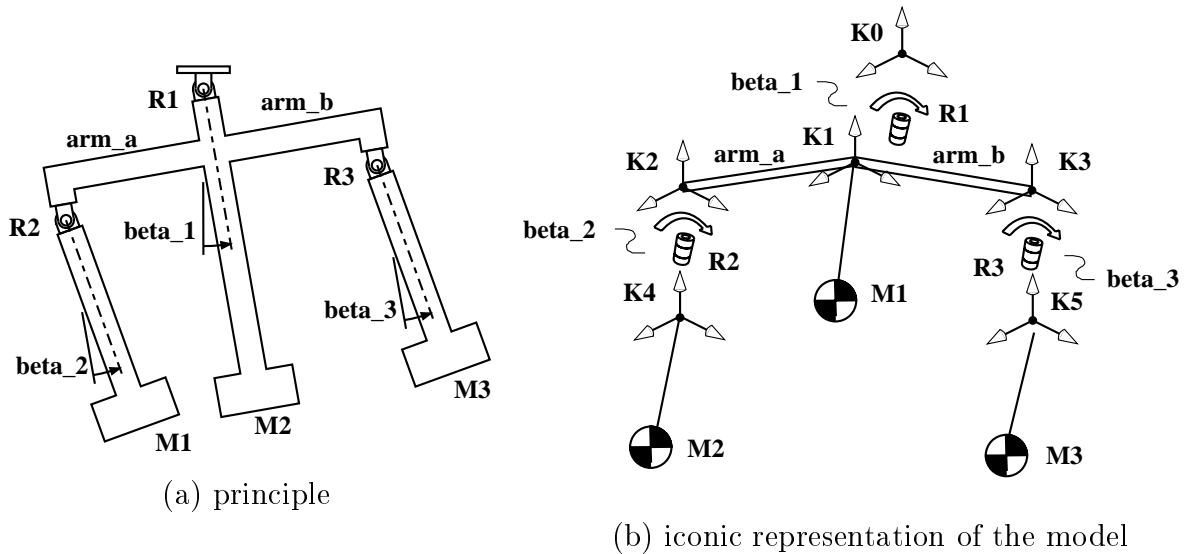


Figure 6.2: Modelling of the TriplePendulum

```

MoAngularVariable beta_1, beta_2, beta_3 ;

MoElementaryJoint R1 ( K0, K1, beta_1, xAxis ) ;
MoElementaryJoint R2 ( K2, K4, beta_2, xAxis ) ;
MoElementaryJoint R3 ( K3, K5, beta_3, xAxis ) ;

MoVector a_1, a_2, a_3, a_4 ;

MoRigidLink arm_a ( K1, K2, a_1 ) ;
MoRigidLink arm_b ( K1, K3, a_2 ) ;

MoReal m1 = 1.0 , m2 = 2.0 ;

MoMassElement M1 ( K1, m1, a_3 ) ;
MoMassElement M2 ( K4, m2, a_4 ) ;
MoMassElement M3 ( K5, m2, a_4 ) ;

MoMapChain pendulum ;
pendulum << R1 << arm_a << arm_b << R2 << R3 << M1 << M2 << M3 ;

// geometry and masses

a_1 = a_2 = a_3 = a_4 = MoNullState ;
a_1.z = -0.5 ;
a_2.z = -0.5 ;
a_1.y = -0.5 ;
a_2.y = 0.5 ;
a_3.z = -1.5 ;
a_4.z = -1.0 ;

m1 = 1.0 ;
m2 = 2.0 ;

```

```
// initial conditions

beta_1.q = DEG_TO_RAD * 30.0 ;
beta_2.q = DEG_TO_RAD * 10.0 ;
beta_3.q = DEG_TO_RAD * 10.0 ;

// dynamics

MoVariableList q ;
q << beta_1 << beta_2 << beta_3 ;

MoMechanicalSystem SystemDynamic ( q , pendulum , K0 , zAxis ) ;

MoAdamsIntegrator SystemIntegrator( SystemDynamic ) ;
MoReal dt = 0.1 ;
SystemIntegrator.setTimeInterval( dt ) ;

for ( int i = 0 ; i++ < 100 ; )
    SystemIntegrator.doMotion() ;
}
```

7 Graphic Rendering and Animation

This chapter describes the graphic interface provided with the M□BILE software. Currently, the graphic interface of M□BILE comprises only animation capabilities. System modeling must be performed in ASCII format using the objects described in the previous chapters. In particular, graphic modeling capabilities are not covered by the present version. These capabilities shall be included in future extensions.

The M□BILE interface for graphic rendering builds substantially on Inventor, the object-oriented graphics library of Silicon Graphics. Similar functionality can be achieved on other machines after installing an Open Inventor-compatible library.

The basic structure of the graphics interface is depicted in Fig. 7.1. There are two basic components involved. One is the model of the mechanical system, which is realized by assembly of the objects described in the previous chapters. Here, motions, forces, etc. are computed according to the prescribed input values and/or user feedback through the interface. The second component concerns the geometric representation of the system's parts on the screen. The definition of geometric information and its rendering are realized by modules of the Inventor package. Here, the man-machine interface is provided through which the user can interact with the system. This involves moving the camera, dragging at joints, changing material properties, starting an animation, etc. The interaction between M□BILE and Inventor is provided by a two-way link. In one direction, M□BILE puts at disposition to the Inventor library the spatial location of frames of interest onto which geometric information is attached. In the opposite direction, Inventor modules set the values of selected input variables either through sliders or manipulators and trigger the motion and force traversal of the M□BILE model.

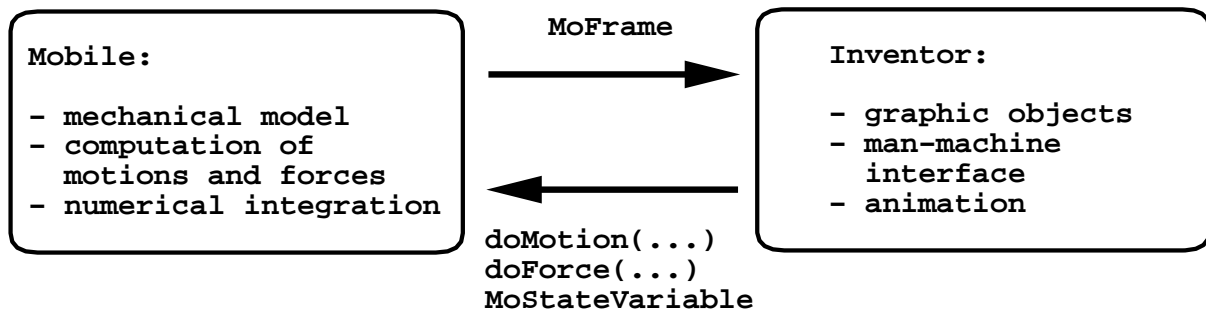


Figure 7.1: Basic structure of the M□BILE-Inventor interface

7.1 Creating a Graphics Interface

In order to produce a graphic interface, the user creates an object of type “MoScene” and passes to it the M□BILE model of the system to be animated. This object is termed the “viewer”. The M□BILE model is responsible for carrying out the computations of motions and forces within the system. These motions are transmitted to the viewer

through specially defined frames, joints and links, to which the user attaches geometric information. The graphic interface then renders this geometric information by superposing the geometric information to the computed motion of the frames, bodies, joints, etc. Moreover, the graphic interface allows the user to move the camera, introduce light effects, manipulate joints, start animations, etc. This produces a virtual prototyping environment in which the user can assess the functioning of the system as if it were operating in real world.

The following code fragment illustrates the use of the graphics interface of MOBILE

```
MoFrame K1 , K2 , K3 ;
MoAngularVariable theta ;
MoVector l ;
MoElementaryJoint joint ( K1 , K2 , theta ) ;
MoRigidLink link ( K2 , K3 , l ) ;
MoMapChain system ; system << joint << link ;
MoScene scene ( system ) ;
scene.makeShape ( joint ) ;
scene.makeShape ( joint , link ) ;
scene.show() ;
scene.mainLoop() ;
```

The object “scene” of type “MoScene” is the actual viewer of the system. This object provides the graphic interface to the user. The mechanical system to be rendered is passed to it as an argument. The viewer takes control over this system, providing data for input motion, and invoking motion and force traversals according to the commands of the user.

At the outset, no graphic information is produced by the viewer. The MOBILE model just represents a skeleton within which motion and force are computed, but no visible geometric properties are considered. In order to make parts visible, the user invokes the function “makeShape()” of the viewer for each part to be rendered. MOBILE provides default geometries for each of the basic modeling objects “MoFrame”, “MoElementaryJoint” and “MoRigidLink”. In order to use default rendering information, the user passes just the names of the corresponding objects to the makeShape() function. The default geometric representation of these objects is reproduced in Table 7.1.

The default geometry information for the basic MOBILE objects is defined in the files

```
MoFrameGeom.so
MoRotationalJoint.so
MoPrismaticJoint.so
MoRigidLink.so
```

These files are shipped in the subdirectory

```
Inventor/examples
```

of the MOBILE home directory. In order to access the default rendering information, the user has to copy these files to the working directory from where the MOBILE program is to be started.

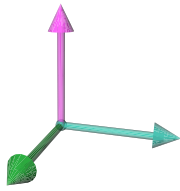


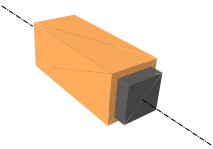
invokation	rendering
<code>makeShape (<frame>)</code>	
<code>makeShape (<joint> , <rigid-link>)</code>	
<code>makeShape (<revolute-joint>)</code>	
<code>makeShape (<prismatic-joint>)</code>	

Table 7.1: Default rendering geometry for the basic M□BILE objects

In the code fragment provided above, default rendering information is used for the objects “joint” and “link”. Note that the “makeShape()” invokation for the rigid link involves also a joint as a first argument. This information is required for aligning the base of the link, which is represented as a fork, with the axis of the joint to which the link is attached.

The actual rendering of the system occurs in the two lines following the “makeShape()” function invokations. The function “show()” creates a static image of the system. The function “mainLoop()” passes control to the graphics engine, allowing the user to interact with the system. These two functions must always be called for the viewer in order for the graphics rendering mechanisms to work properly.

Instead of just creating a shape, the user can also produce so-called manipulators for user interaction. Manipulators provide the capability of user feed-back in addition to pure geometric rendering. For example, with the joint above being defined as a manipulator, the user can accomplish feed-back by picking the joint with a left mouse click and dragging the mouse with the left button pressed down. The joint then moves according to the mouse motion, while the rest of the system follows this motion. The code fragment described below realizes this kind of manipulator action

```
MoFrame K1 , K2 , K3 ;
MoAngularVariable theta ;
MoVector l ;
MoElementaryJoint joint ( K1 , K2 , theta ) ;
MoRigidLink link ( K2 , K3 , l ) ;
```

```

MoMapChain system ; system << joint << link ;
MoScene scene ( system ) ;
scene.makeManipulator ( joint ) ;
scene.makeShape ( joint , link ) ;
scene.show() ;
scene.mainLoop() ;

```

Currently, only manipulators for joints can be created. Invoking the “makeManipulator()” function for links, frames, etc., will result in an error.

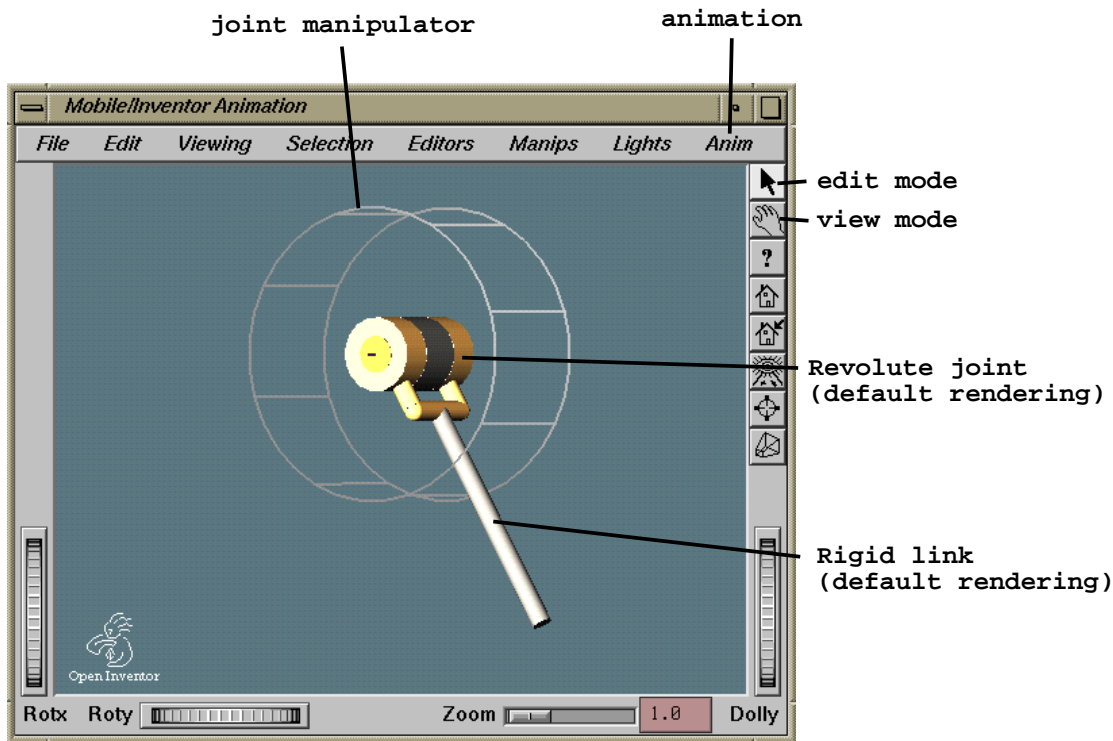


Figure 7.2: Overview of the Inventor interface for MOBILE

Fig. 7.2 shows the viewer resulting for the code described above. One can see the default rendering for the joint and the link. The wireframe “cage” displayed around the joint is the manipulator that pops up when the user hits the joint with a left mouse click while being in “edit” mode. Edit mode is entered by hitting the arrow icon on the right menu bar with a left mouse button click. By hitting the “cage” figure around the joint, and moving the mouse with the left button pressed down, the user can operate the joint. This is called “dragging”. Besides the dragging operations, one can perform viewpoint modification operations. This is accomplished by entering “view” mode. View mode is chosen by clicking on the “hand” symbol on the right menu bar. By moving the mouse with the left button pressed down, the camera is rotated. By moving the mouse with the middle button pressed down, the window is panned. By moving the mouse up and down with the left and middle mouse buttons pressed down, the camera is zoomed out and in, respectively. Other functionalities of the viewer can be inquired by choosing “Help” from

the submenu item “**functions**” in the popup menu obtained by pressing the right mouse button anywhere within the viewer window.

7.2 Importing Inventor Files

Besides using default rendering geometry, M□BILE can be instructed to employ user-defined Inventor files for geometric rendering. Inventor files are imported into the M□BILE model by supplying the file name as an argument to the function invocations “**makeShape()**” or “**makeManipulator()**”. There are three modes for loading Inventor files into the M□BILE model. These three modes are recollected in Table 7.2. In this table, “*file*” represents the name of the file holding the Inventor model for the geometry to be rendered. For information about writing Inventor files, please consult the Inventor Mentor manual provided with the Inventor software.

invokation	action
<code>makeShape(<frame> , file)</code>	attach contents of “ <i>file</i> ” to a frame
<code>makeShape(<revolute-joint> , file)</code>	attach contents of “ <i>file</i> ” to a revolute joint
<code>makeShape(<prismatic-joint> , file)</code>	attach contents of “ <i>file</i> ” to a prismatic joint
<code>makeManipulator(<revolute-joint> , file)</code>	attach contents of “ <i>file</i> ” to a revolute joint
<code>makeManipulator(<prismatic-joint> , file)</code>	attach contents of “ <i>file</i> ” to a prismatic joint

Table 7.2: Importing Inventor files into a M□BILE model

Besides by direct writing, one can also produce Inventor files by translating from other standards to Inventor format. For example, one can translate AutoCAD dxf-files to Inventor format via the function call

```
mobile-home-dir/bin/DxfToIv AutoCad-file.dxf Inventor-file.iv
```

Translating from AutoCad dxf-format to Inventor format allows the user to employ realistic geometric information for the animation.

7.3 Prescribing Motion by Sliders

Apart from manipulators, M□BILE provides also the capability of prescribing user feedback through user-defined sliders. User-defined sliders are grouped in objects of type “**MoWidget**”, termed “slider widgets”. Slider widgets are placed in own windows that can be located anywhere on the screen. A slider widget is initialized with the name of the viewer to which it is to be attached, a kinetostatic transmission element and a character string. The kinetostatic transmission element is traversed in motion mode each time a slider of the widget is actuated. The character string is displayed as the title of the slider widget. One can define several slider widgets for the same viewer.

For each slider widget, one can define several sliders by invoking the functions

```
slider-widget.addSlider ( real , title , min , max ) ;
slider-widget.addSlider ( angle , title , min , max ) ;
```

Here, “*real*” and “*angle*” are names of objects of type “MoReal” and “MoAngle” used as input variables or parameters for previously defined kinetostatic transmission elements. The numbers “*min*” and “*max*” are the minimum and maximum values allowed when operating the sliders. For angles, these values are interpreted in *degrees*. The parameter “*title*” is a character string to be displayed as the title of the slider.

When the user actuates the slider, the value of the scalar quantity attached to the slider is updated and the “doMotion” function of the transmission element attached to the slider widget “*slider-widget*” is invoked. Note that one can use any scalar quantity used within kinetostatic transmission elements as the destination of sliders. Thus, slider widgets make it possible to change parameters online during simulation.

In order for the slider widgets to be rendered on the screen, it is necessary to invoke the “show()” member function explicitly for them. This should be done after the show() function of the viewer object has been invoked.

An example of the use of slider widgets is shown in the program reproduced below. The corresponding result on the screen is displayed in Fig. 7.3.

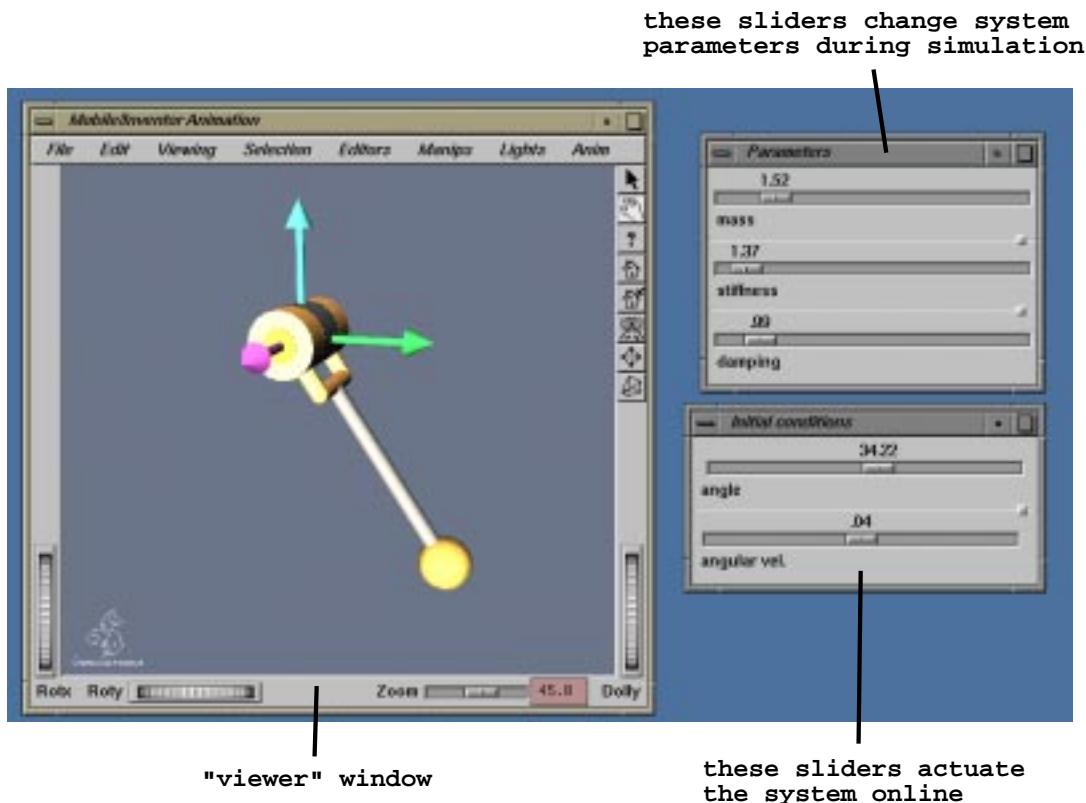


Figure 7.3: An example of the use of slider widgets

```

#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoLinearSpringDamper.h>
#include <Mobile/MoAdamsIntegrator.h>
#include <Mobile/Inventor/MoScene.h>
#include <Mobile/Inventor/MoWidget.h>

void main () {

// definition of mechanical system (see previous section)
MoFrame K0 , K1 , K2 ;
MoAngularVariable phi ;
MoVector l ;
MoElementaryJoint R ( K0, K1, phi ) ;
MoRigidLink rod ( K1, K2, l ) ;
MoReal m ;
MoMassElement Tip ( K2, m ) ;
MoReal k , c ;
MoLinearSpringDamper springDamper ( R , k , c ) ;
MoMapChain Pendulum ;
Pendulum << R << rod << springDamper << Tip ;

// dynamic equation
MoVariableList vars ;
vars << phi ;
MoMechanicalSystem Dynamics ( vars , Pendulum , K0 , yAxis ) ;

l = MoVector ( 0 , -1 , 0 ) ;
m = 1 ;
phi.q = phi.qd = 0 ;

// numerical integrator
MoAdamsIntegrator dynamicMotion ( Dynamics ) ;
MoReal dT = 0.1 ;
MoReal tol = 0.01 ;
dynamicMotion.setTimeInterval(dT) ;
dynamicMotion.setRelativeTolerance(tol) ;

// animation
MoScene Scene ( Pendulum ) ; // interface for 3D-rendering
Scene.makeShape ( K0 ) ; // create shape for inertial frame
Scene.makeManipulator ( R ) ; // create manipulatorfor revolute joint
Scene.makeShape ( R , rod ) ; // create shape for rigid link
Scene.makeShape ( K2 , "MoSphere.so" ) ; // attach ball at end of rod

Scene.addAnimationObject ( dynamicMotion ) ;
Scene.setAnimationIncrement ( 0.1 ) ; // animate in real time if possible

MoWidget parameters ( Scene , Pendulum , "Parameters" ) ;
parameters.addSlider ( m , 0 , 10 , "mass" ) ;
parameters.addSlider ( k , 0 , 30 , "stiffness" ) ;
parameters.addSlider ( c , 0 , 10 , "damping" ) ;

```

```
MoWidget initCond ( Scene , Pendulum , "Initial conditions" ) ;
initCond.addSlider ( phi.q , -360 , 360 , "angle" ) ;
initCond.addSlider ( phi.qd , -10 , 10 , "angular vel." ) ;

Scene.show() ;
parameters.show() ;
initCond.show() ;
MoScene::mainLoop() ; // move the scene
}
```

7.4 Realizing Autonomous Animations

Besides motion generation by user-feed back, M□BILE provides the capability of realizing autonomous motion animation for appropriate objects. Objects for autonomous animation are passed to the viewer through the member function

```
addAnimationObject ( kinetostatic-transmission-element ) ;
```

The object “*kinetostatic-transmission-element*” is an object derived from MoMap and should exhibit a built-in mechanism for progressive motion within the “doMotion()” function. A typical example is an instance of a class derived from “MoIntegrator”.

The user can supply several such objects by repeated invocation of “addAnimationObject()”. These objects are then traversed in the sequence in which they were passed to the viewer during animation. Animation then consists in invoking the “doMotion()” function for the supplied objects (in order of their addition) at constant time intervals.

The (real) time interval between doMotion() invocations can be set via the member function

```
setAnimationIncrement ( time ) ;
```

of the viewer, where “*time*” is a real number representing the time interval. If the computation of the “doMotion()” functions takes less time than the given time interval, the system will wait until the required interval elapses. If the computation of the “doMotion()” functions takes more time than the given time interval, the system will repeat the doMotion() call immediately, progressing at maximum computational speed. The user can force the viewer to work with highest possible speed by specifying a zero value for the animation time interval.

Animation is started from the viewer by selecting “Run Animation” from the drop-down menu under the entry “Anim” at the right of the main menu bar. Animation is stopped by selecting “Run Animation” from the drop-down menu under the entry “Anim” at the right of the main menu bar. Animation can be used in conjunction with manipulators and/or

sliders. Note that attempting to start an animation without supplying appropriate objects through the “`addAnimationObject()`” member function will result in unpredictable errors.

When animating a numerical integration process, one can stop the animation and reconfigure the mechanism parts by operating the manipulators and sliders. Selecting “**Run Animation**” again will then restart the animation from the position attained after reconfiguration. Manipulators store also the speed with which the reconfiguring motion was performed. This speed is then used by the integrator as an initial condition.

7.5 Further Animation Capabilities

Further capabilities of the graphics interface are documented in the M□BILE reference sheets. Furthermore, several examples have been shipped with the M□BILE software in the directory

mobile-home-dir/Inventor/examples

and its subdirectories.

8 M□BILE for PC

The M□BILE for PC version enables the user to write and to display mobile models under Windows (98 and NT) in the same way as on UNIX environments. Moreover, in order to make M□BILE for PC independent of Open Inventor (which requires on PC systems licence fees), the graphic interface was implemented additionally using only OpenGL, which is a royalty free part of Microsoft Windows. Finally, M□BILE for PC integrates a graphic user interface for building models interactively.

8.1 Installation

Installing M□BILE on a PC is as easy as counting one through three:

- (1) Make sure the following products are installed on your PC

Microsoft Visual C++ 6.0

Microsoft FORTRAN Power Station 4.0 or Visual FORTRAN 5.0 (or newer version)

and, if the Open Inventor graphic interface is desired, additionally

TGS Open Inventor 2.5.0

- (2) Copy the M□BILE home directory to your hard-disk. This directory is assumed to be in the following `c:\mobile-home` but can be changed by the user to any other value.

If using M□BILE with Open Inventor, two environment variables have to be set to specific directories. To accomplish this, click the "Start" icon and then traverse the following pop-up menus:

Start → Settings → Control Panel → System → Environment

Then, perform the following two steps (see Fig. 8.1)

- (a) type "MOBILE_INPUTS" in the field "Variable" and "`c:\mobile-home\geom`" in the field "Value"; then click on "Set"
- (b) click on "path" in the list "User Variables for ..." and type "`c:\mobile-home\lib\libInventor`" in the field "Value"; then click on "Set"

[Remark: The "path" variable specifies the location of the file `mobileInvDll.dll`, which is part of the interface between M□BILE and Open Inventor, while the environment variable "MOBILE_INPUTS" specifies the location of standard graphic files of elementary objects like links or joints.]

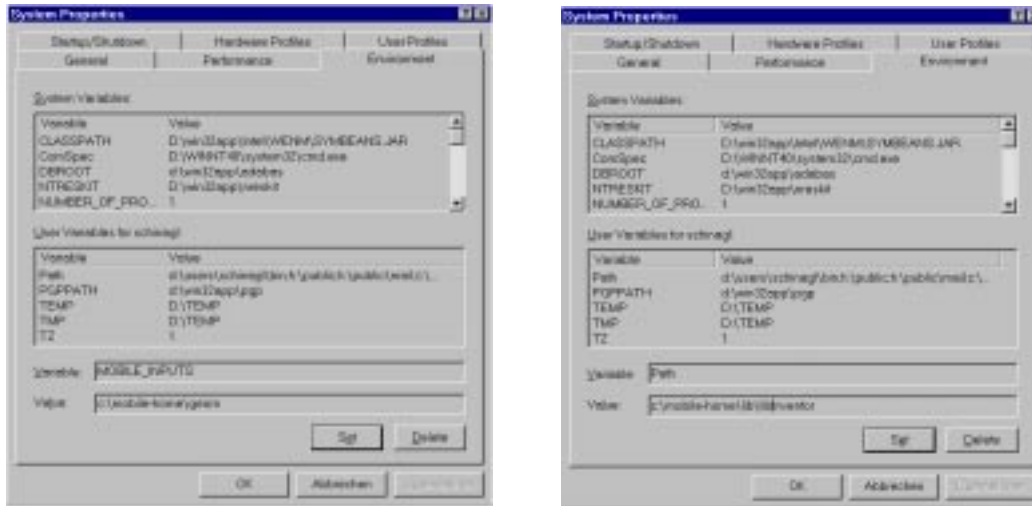


Figure 8.1: Setting a environment variable and a path

(3) Copy the files

`MobileInventor.awx`, `MobileOpenGL.awx`

from the directory

`/mobile-home/pdf/`

to the directory

`/Programme/Microsoft Visual Studio/Common/MSDev98/Template/`

This allows an easy generation of new MOBILE programs with the Application Wizard of Visual C++.

8.2 MOBILE for PC with Open Inventor Graphic Interface

For programming and executing a MOBILE model, the following steps have to be performed:

- (1) Start Microsoft Visual C++
- (2) Select the menu

File → New

In this window, choose **Mobile Inventor AppWizard** and enter a location directory and a project name for the model. For example, in Fig. 8.2 the location typed in is `c:\mobile-home\examples\Inventor` and the project name is `Pendulum`.

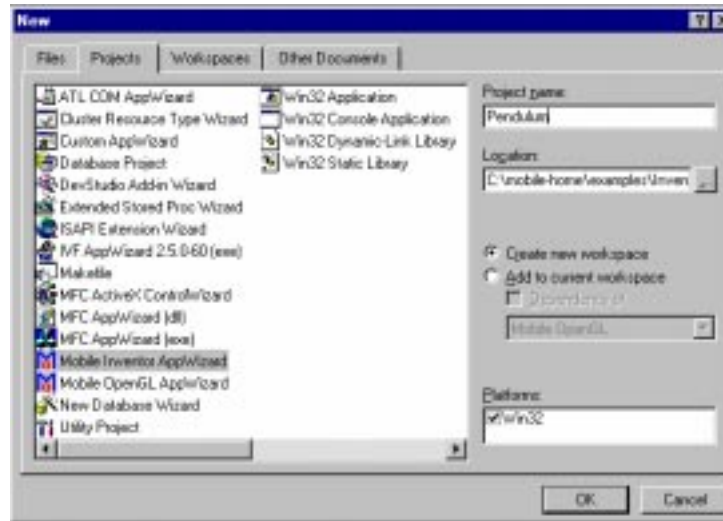


Figure 8.2: Generate a new Project

Also, make sure “**Mobile Inventor AppWizard**” is selected in the Projects window. After closing this and the following window with OK, the **Mobile Inventor AppWizard** generates a new project, including the three new files (in our example MoCmpnt.cpp, MoCmpnt.def, Pendulum.cpp) in the **Source Files** folder of the **Workspace** window. The file Pendulum.cpp, where the actual mobile model source code is typed in by the user, is always generated automatically by the system as the project name with the extension .cpp appended. From the mobile model source file, the executable MOBILE model is lateron automatically compiled and linked as a DLL (Dynamic Link Library). The files MoCmpnt.cpp and MoCmpnt.def are required for the interface between the main program MobileInventor.exe and the executable MOBILE model and should not be changed.

- (3) Open and edit the MOBILE model source file. When opening the MOBILE model source file, you will encounter the following code:

```
#include "Mobile/Inventor/MoMFC.h"
#include "Mobile/Inventor/MoScene.h"
#include "Mobile/Inventor/MoWidget.h"

_void main()
{
    //insert here your Mobile-code

    return MoScene::mainLoop();
}
```

Do not change any of these lines, as otherwise the model will not compile and function correctly!

You may type in a MOBILE model in the place indicated in the template much the same way as in the UNIX system. The only difference is that **all variables must**

be declared as **static**! Not declaring all variables as static will result in loss of information and unpredictable errors!

An example of a program for PC is reproduced below.

```

#include <Mobile/Inventor/MoMFC.h>
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMassElement.h>
#include <Mobile/MoLinearSpringDamper.h>
#include <Mobile/MoAdamsIntegrator.h>
#include <Mobile/Inventor/MoScene.h>
#include <Mobile/Inventor/MoWidget.h>

_void main ()
{
// insert here your Mobile-code
// definition of mechanical system (see previous section)
static MoFrame K0 , K1 , K2 ;
static MoAngularVariable phi ;
static MoVector l ;
static MoElementaryJoint R ( K0, K1, phi ) ;
static MoRigidLink rod ( K1, K2, l ) ;
static MoReal m ;
static MoMassElement Tip ( K2, m ) ;
static MoReal k , c ;
static MoLinearSpringDamper springDamper ( R , k , c ) ;
static MoMapChain Pendulum ;
Pendulum << R << rod << springDamper << Tip ;

// dynamic equation
static MoVariableList vars ;
vars << phi ;
static MoMechanicalSystem Dynamics ( vars , Pendulum , K0 , yAxis ) ;

l = MoVector ( 0 , -1 , 0 ) ;
m = 1 ;
phi.q = phi.qd = 0 ;

Pendulum.doMotion();

// numerical integrator
static MoAdamsIntegrator dynamicMotion ( Dynamics ) ;
static MoReal dT = 0.1 ;
static MoReal tol = 0.01 ;
dynamicMotion.setTimeInterval(dT) ;
dynamicMotion.setRelativeTolerance(tol) ;

// animation

```

```

static MoScene Scene ( Pendulum ) ; // interface for 3D-rendering
Scene.makeShape ( K0 ) ; // create shape for inertial frame
Scene.makeManipulator ( R ) ; // create manipulatorfor revolute joint
Scene.makeShape ( R , rod ) ; // create shape for rigid link
Scene.makeShape ( K2 , "MoSphere.so" ) ; // attach ball at end of rod

Scene.addAnimationObject ( dynamicMotion ) ;
Scene.setAnimationIncrement ( 0.1 ) ;

static MoWidget parameters ( Scene , Pendulum , "Parameters" ) ;
parameters.addSlider ( m , 0 , 10 , "mass" ) ;
parameters.addSlider ( k , 0 , 30 , "stiffness" ) ;
parameters.addSlider ( c , 0 , 10 , "damping" ) ;

static MoWidget initCond ( Scene , Pendulum , "Initial conditions" ) ;
initCond.addSlider ( phi.q , -360 , 360 , "angle" ) ;
initCond.addSlider ( phi.qd , -10 , 10 , "angular vel." ) ;

Scene.show() ;
parameters.show() ;
initCond.show() ;

return MoScene::mainLoop() ;
}

```

- (4) Set the paths to the directories of the include (header) and lib files as follows (see Fig. 8.3):
 - (a) Select the menu Tools → Options... → Directories
 - (b) Select “Include files” in the window “Show directories for”
 - (c) select your M□BILE home directory (in our example “c:\mobile-home”
 - (d) Select “Libraries files” in the window “Show directories for”
 - (e) select your M□BILE Inventor lib directory (in our example c:\mobile-home\lib\libInventor)
- (5) Compile the program by selecting

Build → Build Pendulum.dll (or F7)

The compiler generates the file Pendulum.dll and writes it into the directory /Pendulum/Debug or /Pendulum/Release

- (6) Start the program. To do this, open the directory /mobile-home/mobile-EXE/ and double-click MobileInventor.exe. In the then appearing window (see Fig. 8.4) select the directory and M□BILE model name with the .dll extension (in our example Pendulum.dll). Operate the mechanical model in the same manner as on UNIX environments.

8.3 MOBILE for PC with OpenGL Graphic Interface

The OpenGL Graphic Interface frees the user from the burden of purchasing an Open Inventor license. However, please note that in the OpenGL version not all sophisticated features of Open Inventor will be available. For example, it is only possible to render MOBILE default representations of objects and not CAD files.

New projects are generated with the OpenGL graphics environment in the almost in same way as with the Open Inventor version. The only difference is that in the Projects window **Mobile OpenGL AppWizard** has to be selected instead of **Mobile Inventor AppWizard**, as explained in Page 114, Fig. 8.2.

Please read the instructions in Section 8.2 before continuing, as this section specifies only the differences to the Open Inventor procedure.

The generated file Pendulum.cpp for the OpenGL version is:

```
#include "Mobile/MobileGL/MoMFC.h"
#include "Mobile/MobileGL/MoScene.h"
#include "Mobile/MobileGL/MoWidget.h"

void main()
{
    //insert here your Mobile-code

    return MoScene::mainLoop();
}
```

Again, **do not change any of these lines, as otherwise the model will not compile and function correctly!**

You may type in again a MOBILE model in the place indicated in the template. Again, **all variables must be declared as static!** Not declaring all variables as static will result in loss of information and unpredictable errors!

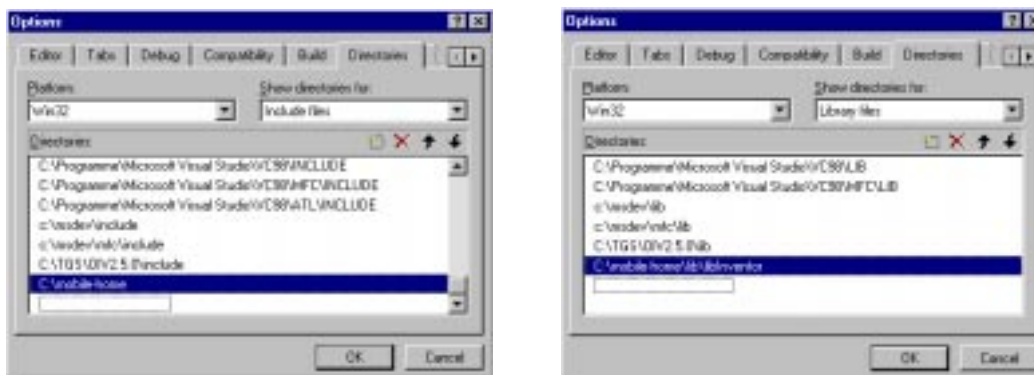


Figure 8.3: Setting the path to header- and library files

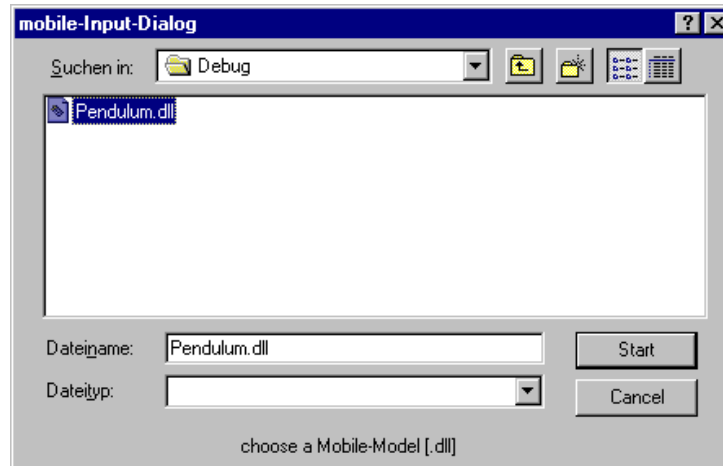


Figure 8.4: Start of a model with Open Inventor

The same program as in section 8.2 yields, then, here

```
#include <Mobile/Inventor/MoMFC.h>
#include <Mobile/MoElementaryJoint.h>
#include <Mobile/MoRigidLink.h>
#include <Mobile/MoMasseElement.h>
#include <Mobile/MoLinearSpringDamper.h>
#include <Mobile/MoAdamsIntegrator.h>
#include <Mobile/Inventor/MoScene.h>
#include <Mobile/Inventor/MoWidget.h>

_void main ()
{
// insert here your Mobile-code
// definition of mechanical system (see previous section)
static MoFrame K0 , K1 , K2 ;
static MoAngularVariable phi ;
static MoVector l ;
static MoElementaryJoint R ( K0, K1, phi ) ;
static MoRigidLink rod ( K1, K2, l ) ;
static MoReal m ;
static MoMassElement Tip ( K2, m ) ;
static MoReal k , c ;
static MoLinearSpringDamper springDamper ( R , k , c ) ;
static MoMapChain Pendulum ;
Pendulum << R << rod << springDamper << Tip ;

// dynamic equation
static MoVariableList vars ;
vars << phi ;
static MoMechanicalSystem Dynamics ( vars , Pendulum , K0 , yAxis ) ;
```



```

l      = MoVector ( 0 , -1 , 0 ) ;
m      = 1 ;
phi.q = phi.qd = 0 ;

Pendulum.doMotion();

// numerical integrator
static MoAdamsIntegrator dynamicMotion ( Dynamics ) ;
static MoReal dT = 0.1 ;
static MoReal tol = 0.01 ;
dynamicMotion.setTimeInterval(dT) ;
dynamicMotion.setRelativeTolerance(tol) ;

// animation
static MoScene Scene ( Pendulum ) ; // interface for 3D-rendering
Scene.makeShape ( K0 ) ; // create shape for inertial frame
Scene.makeShape ( R , rod ) ; // create shape for rigid link
Scene.makeShape ( K2 , SPHERE , 0.1 ) ; // attach ball at end of rod

Scene.addAnimationObject ( dynamicMotion ) ;
Scene.setAnimationIncrement ( 0.1 ) ;

static MoWidget parameters ( Scene , Pendulum , "Parameters" ) ;
parameters.addSlider ( m , 0 , 10 , "mass" ) ;
parameters.addSlider ( k , 0 , 30 , "stiffness" ) ;
parameters.addSlider ( c , 0 , 10 , "damping" ) ;

static MoWidget initCond ( Scene , Pendulum , "Initial conditions" ) ;
initCond.addSlider ( phi.q , -360 , 360 , "angle" ) ;
initCond.addSlider ( phi.qd , -10 , 10 , "angular vel." ) ;

Scene.show() ;
parameters.show() ;
initCond.show() ;

return MoScene::mainLoop() ;
}

```

The program code has only two differences in comparison to the code in Page 116. The first one is that the class MoScene has no member function makeManipulator(joint). The second one is that the function call

```
Scene.makeShape ( K2 , "MoSphere.so" ) ;
```

has been replaced by

```
Scene.makeShape ( K2 , SPHERE , 0.1 ) ;
```

where the value 0.1 indicates the radius of the sphere.

For compilation, select now `/mobile-home/lib/libOpenGL` in Step (4e) of the general procedure for generation and execution of the model displayed in Page 116 (see Fig. 8.3) Compilation of the program can then be performed in the same manner as in Section 8.2.

For starting the MOBILE program, execute the file `MobileGL.exe` in the directory `/mobile-home/mobile-EXE/`. Again, a window appears where the MOBILE model can be selected (see Fig. 8.5).

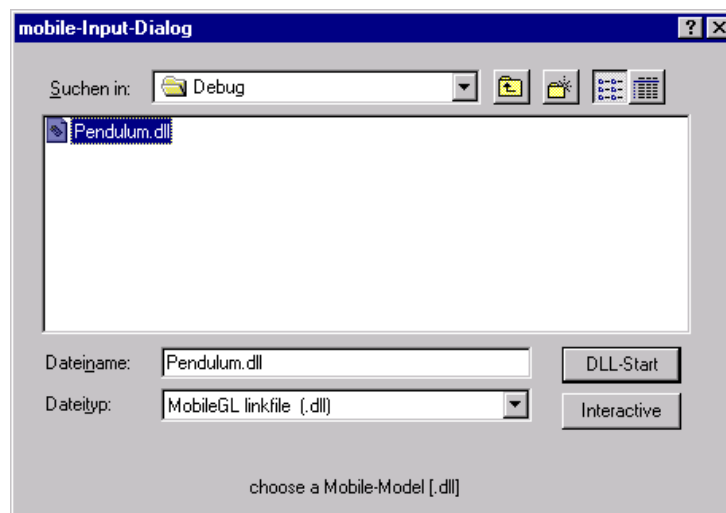


Figure 8.5: Start of a model with OpenGL

8.4 MOBILE for PC with Graphic User Interface

When starting `MobileGL.exe` from the directory

```
/mobile-home/mobile-EXE/
```

(like in section 8.3), the option `Interactive` (see Fig. 8.5) brings up a window with a tool bar on the right side and the initial coordinate frame in the center (see Fig. 8.6). Here, mechanical systems can be modelled, including the closure of loops. Finally, the dynamic equations can be solved, allowing the animation of the system.

Furthermore, the C++ code of the model can be exported in the menu

```
File → Export Source Code
```

Further details for usage of this module can be recognized from the icons displayed in the decoration of the window.

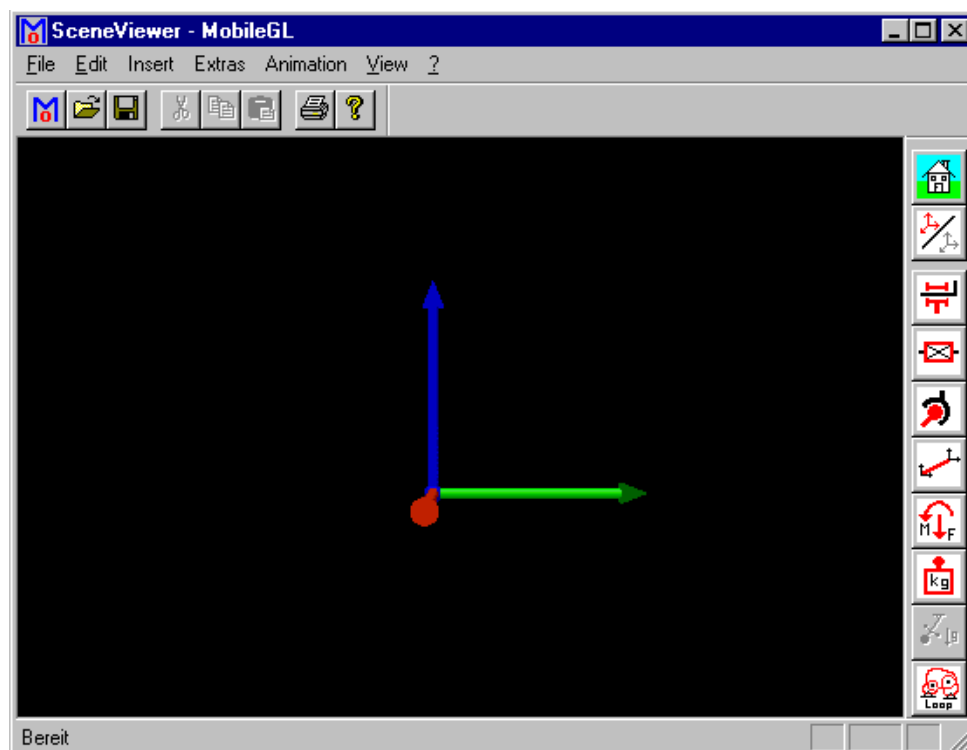


Figure 8.6: Interactive models

Index

- .C, *see* implementation file
- .h, *see* header files
- <<
 - in kinetostatic transmission chains, 48
 - in variable lists, 31

- abstract class, 35
- activity types of measurement, 65
- angle, 16
- angular variables, 29
- animation, 110

- base frame, 72
- basic mathematical objects, 19
- basic mathematical objects, 10

- categories of objects, 10
- chord, 64
- chords, 57
- client-server paradigm, 11
- closed-loop systems, 57
- closure conditions, 57
- compiler invocation, 4
- complementary equations, 79
- COMPUTE_CORIOLIS, 42
- COMPUTE_INTERNAL, 42
- concrete class, 35
- connector paradigm, 29
- constraint equations, 57
- core equation, 79
- Coriolis acceleration term, *see* quadratic acceleration term
- cut, 59
- cyclic coordinates, 22

- decomposition of vectors, 33
- dependent chains, 60
- direction of transmission, 39
- directory structure, *see* MIBILE, directory structure
- DO_ACCELERATION, 42
- DO_ALL, 42
- DO_EULER, 42
- DO_EXTERNAL, 42
- DO_INERTIA, 42
- DO_INTERNAL, 42
- DO_NOTHING, 42
- DO_POSITION, 42
- DO_TRANSFORMATION, 42
- DO_TRANSLATION, 42
- DO_VELOCITY, 42

- examples
 - accessing scalar variables, 30
 - arrays or lists of frames, 34
 - SCARA robot, 54–56
 - shaker mechanism (joint assembly), 60
 - simple manipulator, 49–51
 - simple pendulum (animation), 16
 - simple pendulum (dynamics), 12*ff*
 - simple pendulum (kinematics)*, 5
 - use of MoAngle*, 22
 - use of MoNullState*, 21
 - use of MoVector*, 24

- force*, 51
 - kinetostatic element*, 51
 - definition for a moving frame*, 32
 - generalized applied*, 14
 - generalized Coriolis and centrifugal*, 14
 - source*, 39

- frame*
 - actual*, 32
 - fixed*, 32

- frames*
 - list of*, 34

- geometric types of measurement*, 65

- hardware platforms supported*
 - Silicon Graphics*, 17
- hardware platforms supported*
 - Hewlett Packard*, 17

- header files*, 3
 - container files for*, 9
 - including*, 3
 - overview of existing*, 8–9

- ideal transmission element*, 38
- implementation file*, 3
- implicit constraint solver*, 82

- indexing*
 - C-style*, 32
 - FORTTRAN-style*, 32

- inertia*, 51
- inertia tensor*

- components*, 53
- Jacobians**
 - evaluation in M_QBILE 1.3*, 43
 - force-based determination*, 43
 - velocity-based determination*, 43
- joint**
 - cylindric*, 46
 - elementary*, 46–47
 - prismatic*, 46
 - revolute*, 46
 - screw*, 46
- joints**
 - elementary*, see *elementary joints*
- kinematic inputs*, 60
- kinematic skeleton*, 51
- kinetostatic state subentries*, 28
- kinetostatic state objects*, 10, 27
 - connector paradigm*, 28
 - scalar*, 28
 - types of*, 10
- kinetostatic transmission chain*, 48
- kinetostatic transmission element*
 - chains of*, 48
 - composite*, 48
 - concatenation of*, 48
- kinetostatic transmission elements*, 10
 - basic class hierarchy*, 36
 - chain of*, 48
 - generic model*, 37
 - generic properties*, 36
 - overview of classes*, 37
- linear coordinates*, 22
- linear variables*, 29
- link**
 - binary*, see *binary link*
 - multiple*, see *multiple link*
- lins**
 - rigid*, see *rigid links*
- list of frames*, 34
- lists**
 - of chords*, see *chord lists*
- loop constraint processing*, 58
- lower segment*, 74
- Makefile**, 5
- manipulator*, 105
- mass*
 - kinetostatic element*, 51
- mass (generalized)*, 14
- mass elements*, 53
- matrices**, 24
 - basic operations*, 26
 - basic properties*, 25
 - class hierarchy*, 24
 - columns of*, 26
 - data structure*, 25
 - index expressions*, 26
- measurements**
 - absolute difference type*, 75
 - absolute motion type*, 75
 - characteristic*, 57
 - relative difference type*, 77
 - relative motion type*, 75
 - scalar*, see *scalar measurements*
 - spatial*, see *spatial measurements*
 - topological types of*, 74
- mechanical components**
 - basic operations*, 10
 - responsibilities*, 11
- MoAngle**, 22
 - operations*, 23
- MoAngularVariable**, 29
- M_QBILE**
 - description*, 1
 - directory structure*, 4
 - features of*, 1
 - scope of M_QBILE 1.3*, 2
 - scope of M_QBILE 2.x*, 2
 - scope of M_QBILE 3.x*, 2
- \$MOBILE_HOME_DIR**, 4
- MoChord**, 57
- MoChordList**, 68
- MoCylindricalJoint**, 46
- MoElementaryJoint**, 46
- MoElementaryScrewJoint**, 46
- MoExplicitEulerIntegrator**, 18
- MoFrame**, 32
 - state subentries*, 33
- MoFrameList**, 34
- MoInertiaTensor**
 - operations*, 26
- MoRungeKuttaIntegrator**, 18
- MoLinearVariable**, 29
- MoMap**

- virtual functions*, 35
- MoMapChain, 48
 - sequence of elements in*, 14
- MoMatrix, 23
 - accessing elements*, 26
- MoNullState, 21
- MoReal, 22
- MoRigidLink, 44
- MoRotationMatrix
 - operations*, 27
- MoScene, 17
- MoSolver, 58
- MoStateVariable, 29
- MoAngularList, 31
- MoVector, 23
 - accessing elements*, 23
 - basic operations*, 23
- MoXYZRotationMatrix
 - operations*, 27
- neutral element*, 21
- numerical integrators*
 - Adams-Bashfort-Moulton*, 17
- objects*
 - scalar types*, 22
 - basic mathematical*, 19
 - iconic representation of*, 13
 - not initialized by constructor*, 21
- operators*
 - precedence of*, 20, 26
- parameter passing*
 - pass by reference*, 14
 - pass by value*, 14
- planes of projection*, 73
- pose*, 58
- precedence of operators*, 19, 20
- prerequisites*
 - C++*, 1
 - kinematics and dynamics*, 1
- reference frames*, 10
- rigid link*, 44
 - multiple*, 45
- scalar kinetostatic state objects*, 11, *see state kinetostatic objects, scalar*
- scalar load*, 52
 - applied to a chord*, 52
 - applied to a joint*, 52
- scalar measurements*, 65, 71, 72
 - interlinking of*, 79
 - optimizing performance*, 77
 - Type (I)*, *see scalar measurements, absolute motion type*
 - Type (II)*, *see scalar measurements, relative motion type*
 - Type (III)*, *see scalar measurements, difference of absolute motion type*
 - Type (IV)*, *see scalar measurements, difference of relative motion type*
- scalar types*, 19
- scalar variables*
 - accessing both types of*, 30–31
 - getting the type*, 30
- self-reconfiguring measurements*, 66
- slider widget*, 107
- solver objects*
 - transmission functions*, 60
- solvers*, *see constraint solvers*
- source force*
 - applied and inertia*, 41
- spatial measurements*
 - closure condition*, 70
 - rotational part*, 69
 - statics*, 71
- spatial kinetostatic state objects*, 10, 32
- spatial load*
 - global and local*, 51
- spatial measurements*, 65
 - basic kinematic formulas*, 70
 - basic static formulas*, 71
 - components of state*, 71
 - kinetostatics*, 69
 - types of*, 68
- state objects*, 10, 27
- state** (of measurement), 66
- super loops*, 60
- SYMKIN**, 58
- target frame*, 72
- topological types of measurement*, 65
- transmission*
 - of forces*, 38
 - of motion*, 38
- transmission subtasks*, 40

mathematical terms, 40
tree-type systems, 57

upper segment, 74
USE_CORIOLIS, 42
USE_INTERNAL, 42

variable list, 31
 indexing, 32

vector
 unit, 73
vector components, 23, 33
vector products
 inner, dyadic and vector, 24

viewer, 103, 104
 edit mode, 106
 view mode, 106

whereForce, 78
whereInput, 78
whereUnknwon, 78