ORNL/TM-2002/230

JFKengine: A Jacobian and Forward Kinematics Generator

March 2003

Prepared by
Kathleen N. Fischer
David L. Jung
Arel L. Cordero
Warren E. Dixon
François G. Pin

ORNL/TM-2002/230

JFKengine: A JACOBIAN AND FORWARD KINEMATICS GENERATOR

Kathleen N. Fischer
David L. Jung
Arel L. Cordero
Warren E. Dixon
François G. Pin

Date Published: March 2003

Prepared by
OAK RIDGE NATIONAL LABORATORY
P.O. Box 2008
Oak Ridge, Tennessee 37831-6285
Managed by
UT-Battelle, LLC
For the
U.S. DEPARTMENT OF ENERGY
Under contract DE-AC05-00OR22725

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

The authors gratefully acknowledge the U.S. Department of Energy's Environmental Management Science Program (EMSP) for sponsoring the development of JFKengine. They also thank Dr. Lonnie L. Love for his helpful insights and suggestions during the course of this development, and those who helped in the preparation of this document: Karen Harber for her work to format and finalize the text; Rochelle Womble and Linda Terry Malone for their editorial review; and Dr. C. William Nestor for his helpful comments.

ABSTRACT

During robot path planning and control the equations that describe the robot motions are determined and solved. Historically these expressions were derived analytically off-line. For robots that must adapt to their environment or perform a wide range of tasks, a way is needed to rapidly re-derive these expressions to take into account the robot kinematic changes, such as when a tool is added to the end-effector.

The JFKengine software was developed to automatically produce the expressions representing the manipulator arm motion, including the manipulator arm Jacobian and the forward kinematic expressions. Its programming interface can be used in conjunction with robot simulation software or with robot control software. Thus, it helps to automate the process of configuration changes for serial robot manipulators. If the manipulator undergoes a geometric change, such as tool acquisition, then JFKengine can be invoked again from the control or simulation software, passing it parameters for the new arm configuration.

This report describes the automated processes that are implemented by JFKengine to derive the kinematic equations and the programming interface by which it is invoked. Then it discusses the tree data structure that was chosen to store the expressions, followed by several examples of portions of expressions as represented in the tree. The C++ classes and their methods that implement the expression differentiation and evaluation operations are described. The algorithms used to construct the Jacobian and forward kinematic equations using these basic building blocks are then illustrated.

The activity described in this report is part of a larger project entitled “Multi-Optimization Criteria-Based Robot Behavioral Adaptability and Motion Planning” that focuses on the development of a methodology for the generalized resolution of robot motion equations with time-varying configurations, constraints, and task objective criteria. A specific goal of this project is the implementation of this generalized methodology in a single general code that would be applicable to the motion planning of a wide class of systems and would automate many of the processes involved in developing and solving the motion planning and controls equations. This project is funded by the U.S. Department of Energy’s Environmental Management Science Program (DOE-EMSP) as project EMSP no. 82794 and is transitioning to the DOE-Office of Biological and Environmental Research (OBER) as per FY-02.

1. INTRODUCTION

1.1. PROBLEM STATEMENT

In order to perform tasks on different manipulator platforms, a kinematic model has to be developed for each platform. Because the equations can become too cumbersome to deal with manually when the robots have more than just several joints, a way is needed to automate the formation of the kinematic model.

Industrial robots are usually developed for specific pre-determined tasks. The manipulator arm configuration, along with equations needed for the manipulator arm motion, are determined and solved during robot development. This limits the robot to the prescribed tasks and to no others. However, for robots that must adapt to their environment or perform a wide range of tasks, a way is needed for the manipulator arm to adapt to changes. Changes to the equations (Jacobian and forward kinematic expressions), are required when something changes the geometry of the manipulator arm, such as when a tool is added to the end-effector.

In order to accommodate different manipulator platforms and to provide for tool acquisition, a way is needed to automate the formation of the kinematic model that eliminates manual processes.

1.2. BACKGROUND

The Jacobian and forward kinematic expressions for a robot manipulator can be derived analytically from a Denavit-Hartenberg (D-H) table.[1-2] These expressions become complicated when the arm consists of more than just two or three joints. For example, some individual terms of the forward kinematic expressions for a six degree-of-freedom PUMA[3] robot contain more than 30 trigonometric functions each.[4] With the advent of tools such as MATLAB™,[5] it was easier to produce these expressions, but they were still produced for a particular robot through a manual process and then incorporated into robot control or simulation software through another manual process.

The first implementation of JFKengine was developed and programmed by Dr. Warren E. Dixon and Mr. Arel Cordero of the University of Oregon during a summer internship of the latter at Oak Ridge National Laboratory. This first generation was a stand-alone Java application. Its input was a text file containing the D-H table. Its output consisted of computer software: two subroutines that computed the Jacobian and the forward kinematics expressions. This eliminated the problem of producing the cumbersome kinematic expressions manually, but it still required a manual process to compile the resulting subroutines and link them with existing inverse kinematics solvers. The second implementation of the JFKengine was developed to eliminate the need of manual processes altogether. The remainder of this report refers to this second implementation of the JFKengine.

1.3. SOLUTION

The second implementation of the JFKengine software was developed to produce expressions needed for manipulator arm motion for a general n-dimensional system. The JFKengine can formulate the Jacobian and the forward kinematic expressions, as well as evaluating the forward kinematic expression for a given set of joint variables to yield the link locations. Its programming interface can be used in conjunction with robot simulation software or with robot control software. Thus, it helps to automate the process of configuration changes for serial robot manipulators.

The software that is using the JFKengine passes it information about the manipulator that is specified in D-H notation. The JFKengine uses the D-H parameters to produce and pass back the requested expressions. If the manipulator undergoes a geometric change, such as tool acquisition, then the JFKengine can be invoked again from the control or simulation software, passing it the D-H parameters for the new arm configuration.

1.4. APPROACH

The approach is to perform symbolic computations to construct, differentiate, and simplify mathematical expressions used for forward and inverse kinematic computations. The expressions involve variables that can be evaluated numerically for given values of the dependent variables (the joint variables).

A tree data structure on the heap was chosen to store the kinematics and Jacobian expressions. The leaves of the expression tree are the constants or variables, while the nodes higher in the tree contain the operators. The expression tree will be described in greater detail below, along with the C++ classes and their methods that implement the expression differentiation and evaluation operations. The algorithms used to construct the Jacobian and forward kinematic equations using these basic building blocks are then illustrated.

2. EQUATIONS

The equations that describe the manipulator arm motions are discussed in this section. The robotics text *Robot Analysis and Control*,[2] by Asada and Slotine, was followed for this development. Refer to that text for the derivations and details.

2.1. FORWARD KINEMATICS

The manipulator arm can be thought of as a chain of rigid bodies with a coordinate axis or frame attached to each one. The D-H notation defines four parameters that specify the relative location of frame *i* with respect to the previous one (see Fig. 1):

a_i	the length of the common normal, called the link length; that is, the distance from the intersection of the z_{i-1} axis with the x_i axis to the origin of the *i*th frame along the x_i axis
d_i	the joint variable for a prismatic joint; the distance between the origin O_{i-1} and the point H_i, the intersection of the z_{i-1} axis with the x_i axis along the z_{i-1} axis
α_i	the angle between the joint axis *i* and the z_i axis about the x_i axis in the right-hand sense, called the twist of the link
θ_i	the joint variable for a revolute joint; the angle from the x_{i-1} axis to the x_i axis about the z_{i-1} axis, using the right-hand rule

The joint variable of the *i*th joint, q_i is defined as:

$$
\begin{aligned}
q_i &= \theta_i \text{ for a revolute joint} \\
q_i &= d_i \text{ for a prismatic joint}
\end{aligned}
\tag{1}
$$

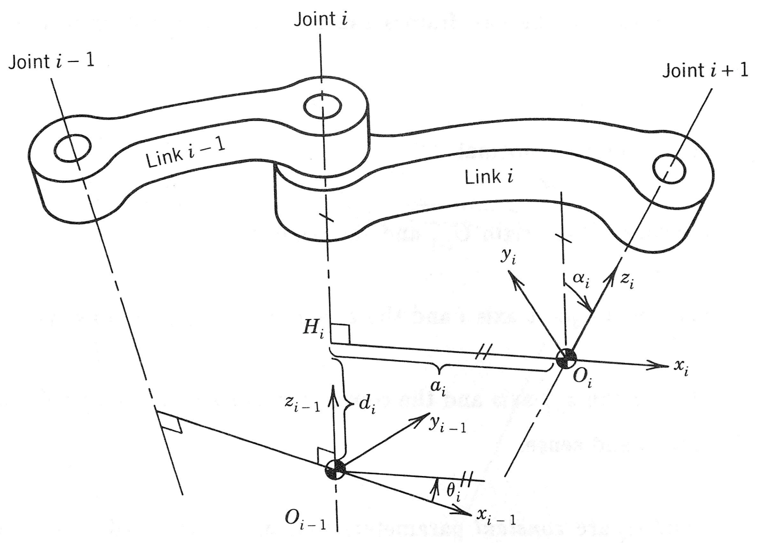


Fig. 1. Denavit-Hartenberg parameters.

The D-H notation provides a way to represent each frame by a 4×4 matrix, $\mathbf{A}_i^{i-1}$, that gives the position and orientation of the *i*th body with respect to the previous one.

$$\mathbf{A}_i^{i-1}(q_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\cos\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

The **A** matrices can be used to develop a matrix of expressions for the forward kinematics equation, **T**, for the manipulator arm. **T** is a 4×4 matrix which gives the position and orientation of the end-effector with respect to the base frame as a function of each of the joint variables q_i.

$$\mathbf{T} = \mathbf{A}_1^0(q_1)\mathbf{A}_2^1(q_2)\ldots\mathbf{A}_n^{n-1}(q_n) \tag{3}$$

2.2. JACOBIAN

The manipulator Jacobian, **J**, is a $6 \times n$ matrix where n is the number of joints in the manipulator arm. The ith column of **J** can be thought of as two 3×1 vectors, $\mathbf{J}_{Li}$ and $\mathbf{J}_{Ai}$, which are associated with the linear and angular velocities, respectively, of the tip of the robot arm due to the ith joint velocity. So we can partition **J** as follows:

$$\mathbf{J}_i = \begin{bmatrix} \mathbf{J}_{Li} \\ \mathbf{J}_{Ai} \end{bmatrix} \tag{4}$$

The Jacobian deals with small motions of the end-effector about its current position and arm configuration, so each of the elements of $\mathbf{J}$ are functions of the joint variables, q_i (see Eq. (1)). The first three rows of $\mathbf{J}$ ($\mathbf{J}_L$) deal with the linear velocity of the end-effector with respect to the base coordinate system. Each column of $\mathbf{J}_L$, vector $\mathbf{J}_{Li}$, is formed by differentiating the expression for the position of the end-effector, which is given as the last column in **T** (Eq. (3)), as follows:

$$\mathbf{J}_{Li} = \begin{bmatrix} \frac{\partial x}{\partial q_i} \\ \frac{\partial y}{\partial q_i} \\ \frac{\partial z}{\partial q_i} \end{bmatrix} \tag{5}$$

The last three rows of $\mathbf{J}$ ($\mathbf{J}_A$) deal with the angular velocity of the end-effector and are due to the angular velocity of the end-effector generated by each joint. There is no contribution to the angular velocity at the end-effector for prismatic joints, so:

$$\mathbf{J}_{Ai}\dot{q}_i = \mathbf{0} \qquad \text{for prismatic joint } i \tag{6}$$

However a revolute joint i rotates the links i to n at the angular velocity ω_i as follows:

$$\mathbf{J}_{Ai}\dot{q}_i = \omega_i = \mathbf{b}_{i\text{-}1}\dot{\theta}_i \tag{7}$$

where $\mathbf{b}_{i-1}$ is the unit vector pointing along the direction of the joint axis *i*. For revolute joint *i*, the rotation is about the z_{i-1} axis, by convention. In terms of coordinate frame *i*, $\mathbf{b}_{i-1}$ is represented as $[0,0,1]^T$. It must be transformed to express it with respect to the base frame. The first 3×1 column vectors of $\mathbf{A}_i^{i-1}$ (Eq. (2)) are the direction cosines of the *i*th coordinate frame. They form a rotation matrix $\mathbf{R}_i^{i-1}(q_i)$:

$$\mathbf{A}_i^{i-1}(q_i) = \left[\begin{array}{ccc|c} & \mathbf{R}_i^{i-1} & & \mathbf{x}_i \\ & & & \\ \hline & 0 & & 1 \end{array} \right] \tag{8}$$

where $\mathbf{x}_i$ is the last 3×1 column vector of **A**. This rotation matrix **R** can transform a vector in the *i*th frame to one in the previous (*i-1*) frame. To determine $\mathbf{b}_{i-1}$ for Eq. (7), we can use successive rotation matrices to express the z_{i-1} axis with respect to the base frame as follows:

$$\mathbf{b}_{i-1} = \mathbf{R}_1^0(q_i) \ldots \mathbf{R}_{i-1}^{i-2}(q_{i-1}) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{9}$$

To summarize, we now have the means to compute $\mathbf{J}_{Ai}$ for revolute and prismatic joints:

$$\begin{aligned} &\mathbf{J}_{Ai} = 0 && \text{for prismatic joint } i \\ &\mathbf{J}_{Ai} = \mathbf{b}_{i-1} = \mathbf{R}_1^0(q_i) \ldots \mathbf{R}_{i-1}^{i-1}(q_{i-1}) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} && \text{for revolute joint } i \end{aligned} \tag{10}$$

3. PROGRAMMING INTERFACE

The JFKengine currently has three public[*] methods that may be called to produce or evaluate the equations described in Sect. 2:

getForwardKinematics	forms forward kinematic expression matrices **A** and **T**, using D-H parameters of the first n joints of the manipulator
getJacobian	forms Jacobian expression **J**, using D-H parameters of the first n joints of the manipulator
getJointLocations	evaluates the forward kinematic expressions to return the link endpoints or joint locations, using the joint variables passed to it

Each of these methods requires that a SerialManipulator object be passed as an argument. The SerialManipulator class is an abstract class that describes the manipulator for the JFKengine. It defines the JointParameters structure that stores the D-H table for a joint. It also defines abstract method getParameters, which returns an array of all the manipulator's joint parameters.

The parameter lists, returned values, and possible exceptions that may be raised are now described for each of the JFKengine public methods. An example using these methods concludes this section.

3.1. getForwardKinematics

ExpressionMatrix JFKengine :: getForwardKinematics (ref < const robot :: SerialManipulator > *manip*,
Int *n* = AllJoints
)

forms forward kinematic expression matrices **A** and **T**, using the D-H parameters of the first *n* joints of the manipulator.

Parameters:

manip	smart pointer (ref< >) to a SerialManipulator for which the forward kinematics expression matrices are to be formed based on the manipulator's D-H parameters
n	use first *n* joints of manipulator for expressions (defaults to all joints)

Returns:

A 4 ∏ 4 ExpressionMatrix containing transformation matrix **T** for the given SerialManipulator; **T** represents the position and orientation of the *n*th link with reference to the base coordinate frame of the manipulator.

Exceptions:

Std::out_of_range	exception, if no joints are specified or *n* is greater than the number of joints in manipulator

References A, AllJoints, d, Debugc, Debugcln, Debugln, DH_AT, Exception, base::ExpressionMatrix, forwardKinematicsCached, base::Int, base::Real, base::array< ExpressionMatrix >::resize, base::array< ExpressionMatrix >::size, and T.

Referenced by getJacobian and getJointLocations.

[*] C++ public methods have unlimited access control; they may be used by any function.

3.2. getJacobian

```
ExpressionMatrix JFKengine::getJacobian(ref < const robot::SerialManipulator >   manip,
                                        Int                                      n = AllJoints,
                                        bool                                     includeOrientation = true
                                        )
```

forms Jacobian expression, using the D-H parameters of the first *n* joints of the manipulator. If forward kinematics have not already been calculated for this manipulator, then getForwardKinematics is called by getJacobian.

Parameters:

manip	smart pointer (ref<>) to a SerialManipulator for which the Jacobian is to be formed, based on its D-H parameters
n	use first *n* joints of manipulator (defaults to all joints)
includeOrientation	if true, **J** will be a $6 □ n$ ExpressionMatrix with orientation components; otherwise it will be a $3 □ n$ ExpressionMatrix including only position components

Returns:

A $[6|3 □ n]$ ExpressionMatrix containting the Jacobian for the given SerialManipulator; *n* is the number of joints from the given SerialManipulator that were used.

Exceptions:

s*td::out_of_range*	exception, if no joints are specified, or *n* is greater than the number of joints in the manipulator

References AllJoints, Debugcln, Debugln, DH_J, Exception, base::ExpressionMatrix, getForwardKinematics, base::Int, J, Jcached, and T.

3.3. getJointLocations

```
array < Vector > JFKengine::getJointLocations(ref < const robot::SerialManipulator >   manip,
                                              const Vector &                           jointValues
                                              )
```

evaluates the forward kinematic expressions to return the joint locations.

If forward kinematics have not already been calculated for this manipulator, then getForwardKinematics is called.

Parameters:

manip	smart pointer (ref< >) to a SerialManipulator for which the forward kinematic expressions are to be used, based on its D-H parameters
jointValues	array of Reals; jointValues[0] is the value of the joint variable for the first joint

Returns:

loc	array<Vector> which contains the $[x, y, z]$ location of the coordinate frame of each link, which corresponds to the end of the link or the joint location

References Debugcln, Debugln, Exception, getForwardKinematics, base::Int, T, and base::Vector.

3.4. SAMPLE CODE

The following portion of C++ code shows how the Jacobian can be formed and then evaluated for given joint variable values.

```
// JFKengine use example

// First we need to get an instance of a SerialManipulator class that
//  describes the manipulator structure for the JFKengine.
//  This is obtained from a Robot class that describes a complete robot
//  (including any manipulators it has)
// Typically, a concrete Robot subclass is defined for any real robot, but
//  for this example we use a test class called TestRobot that can be
//  supplied directly with the D-H parameters for a single manipulator arm.

// first setup some Vectors of D-H parameters for a 3-DOF arm
IVector jt(3); // joint type
Vector alpha(3), a(3), d(3), theta(3); // D-H params

jt[0] = TestRobot::Revolute; a[0] = 1;   d[0] = 0;   alpha = consts::Pi/2; theta[0] =
0;
jt[1] = TestRobot::Revolute; a[1] = 2;   d[1] = 0.1; alpha = consts::Pi;   theta[1] =
0;
jt[2] = TestRobot::Revolute; a[2] = 0.5; d[2] = 0;   alpha = 0;            theta[2] =
0;

ref<TestRobot> robot(NewObj TestRobot(jt, alpha, a, d, theta));

// In general, a robot can have multiple manipulators of various types;
//  however, the TestRobot has only a single manipulator that is serial.
// get list of robot's manipulators
const reflist<Manipulator>& manipulators( robot->getManipulators() );

// take the first Manipulator (from the front of the list) and cast it into a
// SerialManipulator
ref<SerialManipulator> manipulator(
   narrow_ref<robot::SerialManipulator>(manipulators.front()) );

// Now instantiate an instance of the JFKengine
ref<JFKengine> jfk(NewObj JFKengine());

// Ask the JFKengine to give us an ExpressionMatrix that is the symbolic
//  representation of the Jacobian given the D-H parameters (via SerialManipulator)
ExpressionMatrix J( jfk->getJacobian(manipulator) );  // 6x3 matrix of Expressions

// Now we can evaluate the Jacobian for any joint variable vector. For example:
Vector q(3);
q[0] = consts::Pi/4; q[1] = -consts::Pi/3; q[2] = consts::Pi/6;

// evaluate symbolic J at numeric q for numeric Jeval result
Matrix Jeval( evaluate(J, q) );

std::cout << Jeval << std::endl; // print it out

// evaluate for a different q vector
q[1] = consts::Pi/2;
Jeval = evaluate(J,q);

std::cout << Jeval << std::endl; // print it out
```

4. IMPLEMENTATION

4.1. EXPRESSION TREE

4.1.1. Tree Formation

The elements of **T**, the forward kinematics equation of the manipulator arm, and those of the manipulator Jacobian **J**, are functions of the joint variables, making them dependent on the manipulator arm configuration. An implementation was needed that would allow the equations to be evaluated for different joint variables, q_i.

A tree data structure was chosen to implement the equations of JFKengine. The forward kinematics and Jacobian equations consist of sums, differences, and products of terms that involve sines and cosines. The operators, constants, and variables are assembled into expression trees by JFKengine. For this discussion of how the trees are formed, a few definitions are necessary.

A *tree*[6] is a collection of *nodes*. The collection can be empty; otherwise, a tree consists of a distinguished node *r*, called the *root*, and zero or more nonempty *subtrees*, each of whose roots are connected by a directed edge from *r*. The root of each subtree is said to be a *child* of *r*, and *r* is the *parent* of each subtree root. Nodes with no children are known as *leaves*. Nodes with the same parent are *siblings*. A *binary tree* is a tree in which no node can have more than two children.

C++ was chosen as the implementation language with which to develop class JFKengine. A C++ class called Expression was developed to serve as the root of the expression tree, and it contains a smart pointer[7] to an ExpressionNode. Abstract base class ExpressionNode was developed to provide a model for the nodes in the expression tree. The nodes in the tree consist of operators and their operands.

The leaves of the tree are the operands, and they can be either constant values or variables. C++ classes ConstantExpression and VariableExpression were developed to model operands in the expression tree. ConstantExpression stores an actual number, whereas VariableExpression stores the index that will be used to identify the desired variable from a list of parameters.

The operators encountered in the kinematic equations are either binary, requiring two operands, or unary, requiring just one operand. Abstract base class UnaryOpExpression was developed to model unary operators, as Fig. 2 shows. The unary operators needed for the equations **A** and **T** are -, sin( ), and cos( ). Concrete subclasses of UnaryOpExpression are SinExpression, CosExpression and NegateExpression.

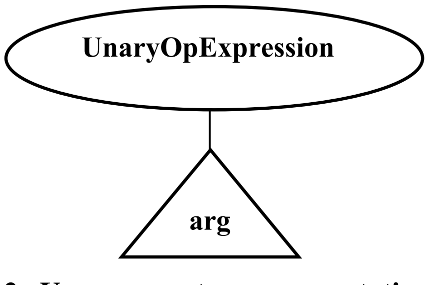


Fig. 2. Unary operators representation.

Example 1: In the case of a revolute joint *i*, the term $\cos\theta_i$ would appear in the kinematic equations. It would be represented in the tree as operator cos( ) with the argument being a variable expression with

$\text{index} = i$, to allow the expression to be evaluated for different values of θ_i. Figure 3 shows what $\cos\theta_i$ would look like in the expression tree.

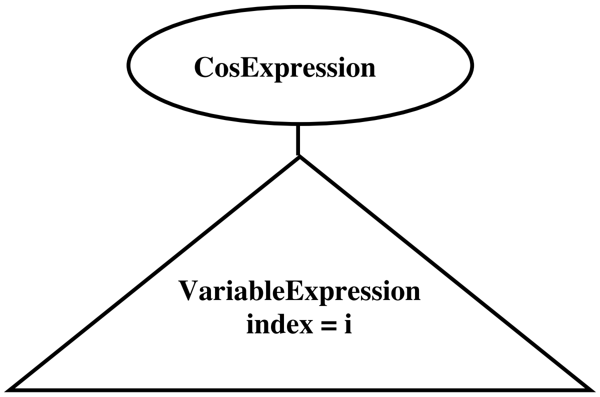


Fig. 3. Expression tree for $\cos\theta_i$.

A binary subtree is used in JFKengine to model binary operators. It is implemented in C++ as abstract base class BinaryOpExpression, with references to the left argument and the right argument of the operator, as Fig. 4 shows. The binary operators needed for the equations **A** and **T** are (+, - , and *). Concrete classes SumExpression, DifferenceExpression and ProductExpression are subclasses of BinaryOpExpression which model the binary operators.

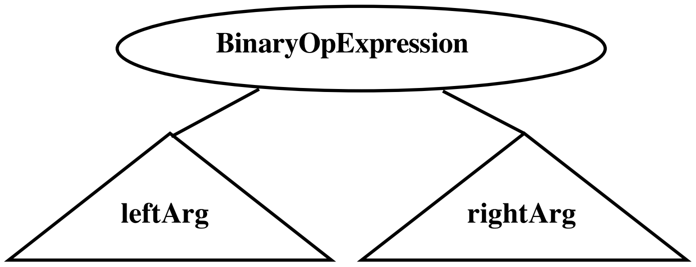


Fig. 4. Binary operators representation.

Example 2: A binary operation that appears frequently in the forward kinematic expressions is a constant, multiplied by $\cos\theta_i$, for example $(0.6 * \cos\theta_4)$. This would be represented in the expression tree as operator (*), with the left argument being a constant. In this example we see that the argument of an operator may be another operator. The right argument of the product is another operator, that is, $\cos(\ )$. This is modeled by CosExpression, which takes a single argument. In this case the argument is a joint variable for the fourth joint. The zero-based index is set to indicate which joint variable it corresponds to; in this case $\text{index} = 3$ for the fourth joint. Figure 5 shows what this would look like in the expression tree.

4.1.2. Tree Evaluation

Class ExpressionNode provides storage for a value and a cachedValue flag to indicate if the value has been cached. It also provides an evaluate method. When an expression node is "evaluated" by invoking its evaluate method, the flag is checked to determine if the value has already been evaluated. If it has, the value has already been stored or "cached" so that it will not need to be evaluated again. If the value has not yet been evaluated, it calls the cacheValue method. The abstract virtual function cacheValue is defined appropriately in the concrete subclasses of ExpressionNode. Figure 6 shows how evaluation starts at the root of the tree from an invocation of the evaluate method of Expression. The details of what occurs when cacheValue is called will be discussed below.

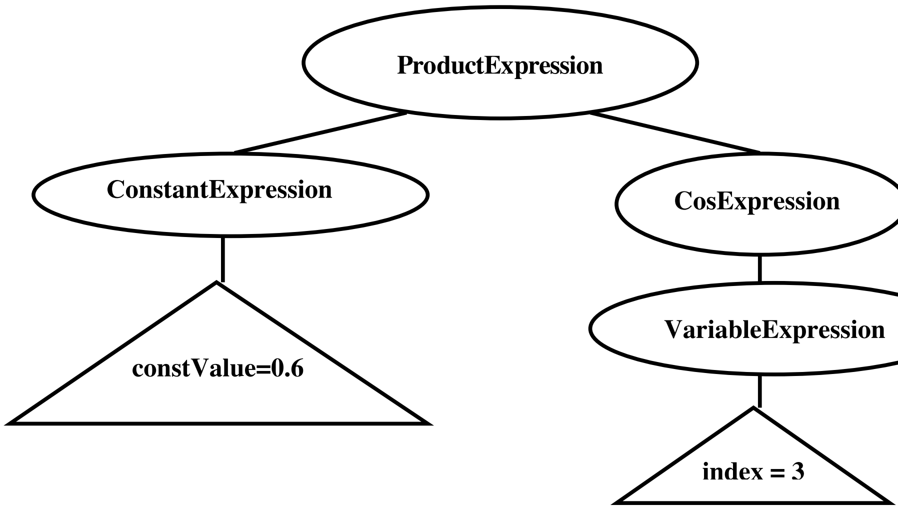


Fig. 5. Expression tree for $(0.6 * \cos \theta_4)$**.**

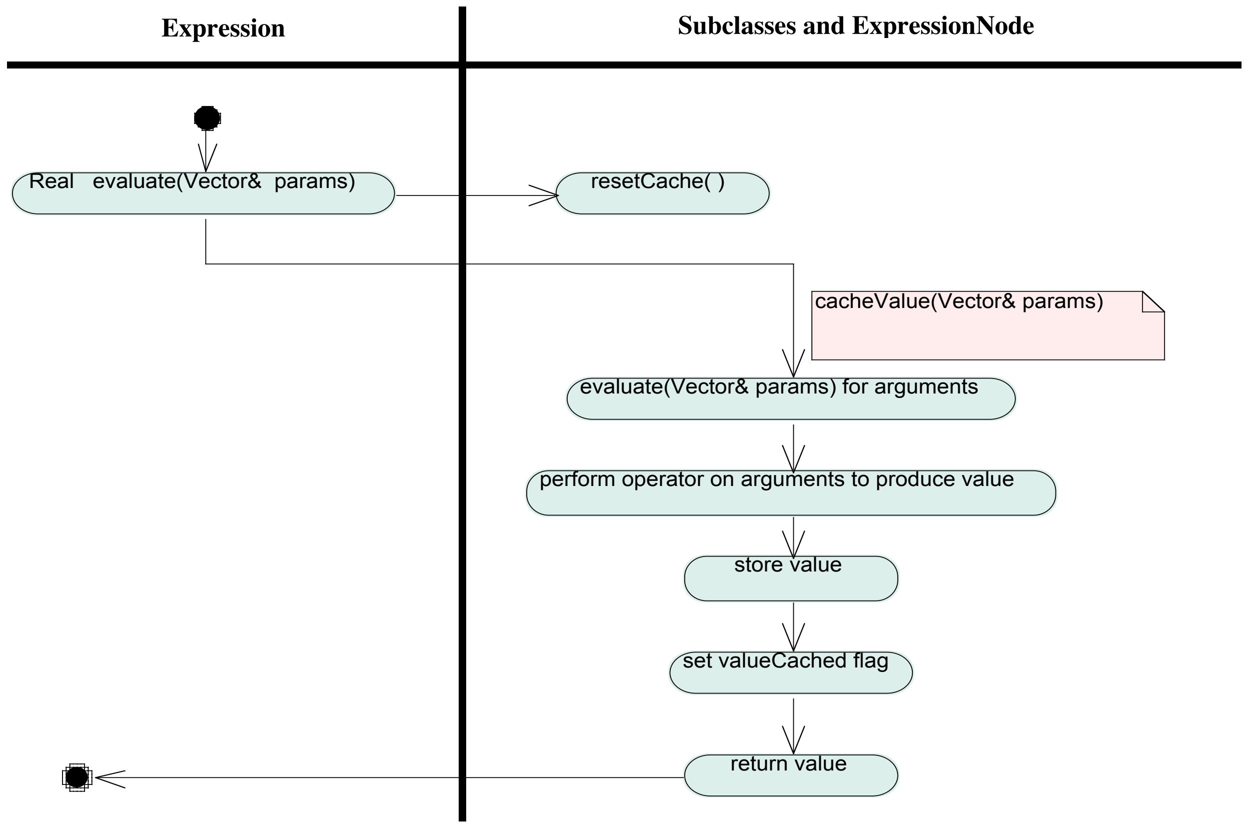

Expression
Subclasses and ExpressionNode
Real evaluate(Vector& params)
resetCache( )
cacheValue(Vector& params)
evaluate(Vector& params) for arguments
perform operator on arguments to produce value
store value
set valueCached flag
return value

Fig. 6. Expression evaluation overview.

To evaluate an expression, the joint variables are passed to the evaluate method of the expression. When an instance of VariableExpression is created, it receives a zero-based index number that it stores. This indicates to which joint variable it applies. For example, to represent θ_4 as a VariableExpression, 3 would be passed to the VariableExpression constructor when it is created. When the VariableExpression is evaluated, it receives a reference to a Vector[†] of parameters, *p*. It would return the value of p[3], the one stored at the fourth position in the passed Vector.

Example 3: Figure 7 is a Unified Modeling Language (UML) sequence diagram. A few notes are included here on its notation.[8-9] The boxes at the top show objects. Their names take the form *objectName* : ClassName. The vertical line, called the object's lifeline, represents the object's life during the interaction of the classes. Arrows between the lifelines of two objects represent a message being passed between them. This is done through calls to the object's methods (functions). In some cases the argument list for the call has been included. There are several *self-calls*, where the object sends itself a message, indicated by an arrow that goes back to the same lifeline.

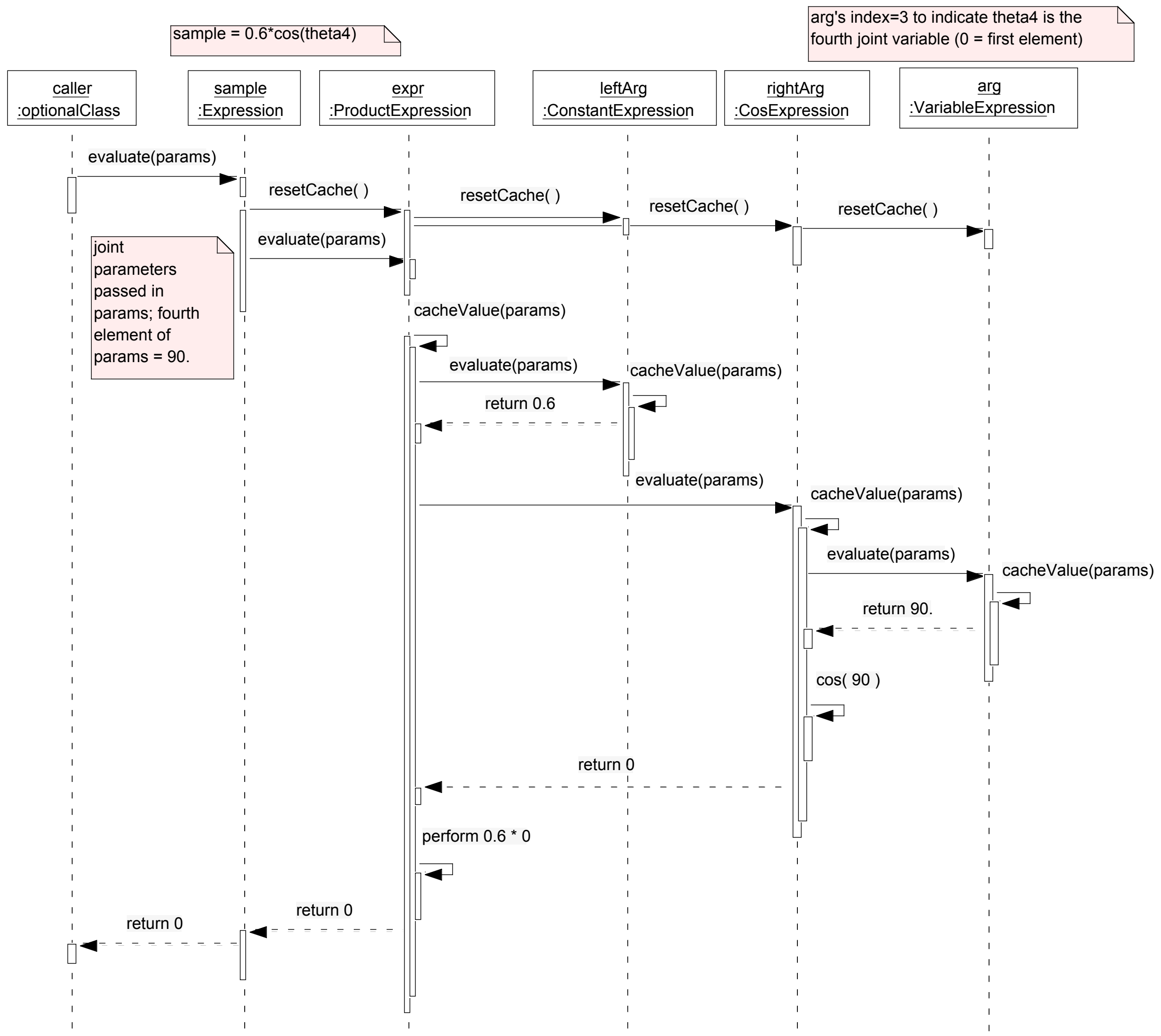


Fig. 7. Evaluate expression $(0.6 * \cos\theta_4)$.

[†] Vector is a class that is part of the basic framework for the OpenSim library. See Sect. 4.3.

Figure 7 shows how calling the evaluate method of an Expression triggers a sequence of calls to the components of the expression tree to produce a numeric result. Note that ConstantExpression, ProductExpression, CosExpression, and VariableExpression are all subclasses of ExpressionNode. (Inheritance is shown on class diagrams in the Appendix). All those classes inherit the same evaluate method from the base class, ExpressionNode. ExpressiooNode's evaluate method calls cacheValue, passing it the joint variables as a Vector reference (Vector&). Although the evaluate method comes from the base class, cacheValue is an abstract virtual function in ExpressionNode. That allows each ExpressionNode specialization to provide its own flavor of cacheValue to supply the different operator and operand implementations.

For this example, the same expression was used as that described in Example 2 in Sect. 4.1.1. Refer to Fig. 5 for the expression tree structure for $(0.6 * \cos\theta_4)$. This example assumes that the value of 90 for θ_4 has been passed to the evaluate method. We see that after the cascade of method calls, the final result of 0 is returned.

4.1.3. Tree Differentiation

The portion of the Jacobian **J** that is associated with the linear velocity is $\mathbf{J}_L$ (see Eq. (5)). $\mathbf{J}_L$ is composed of the partial derivatives with respect to the joint variables of the end-effector position expressions. The partial derivative of an Expression with respect to a joint variable may be taken by calling the differentiate method of the Expression and passing it a VariableExpression with the index set to the desired joint variable. The differentiate method of Expression, in turn, calls the differentiate method on the root of the tree. ExpressionNode, the base class for the nodes in the tree, provides an abstract virtual differentiate method that the derived classes must provide. It receives, as its argument, an integer that specifies with respect to what joint variable to differentiate. That allows each ExpressionNode specialization to provide its own version of differentiate to supply the different operator and operand implementations.

Basic rules of differential calculus were followed in implementing the derivatives. For example,

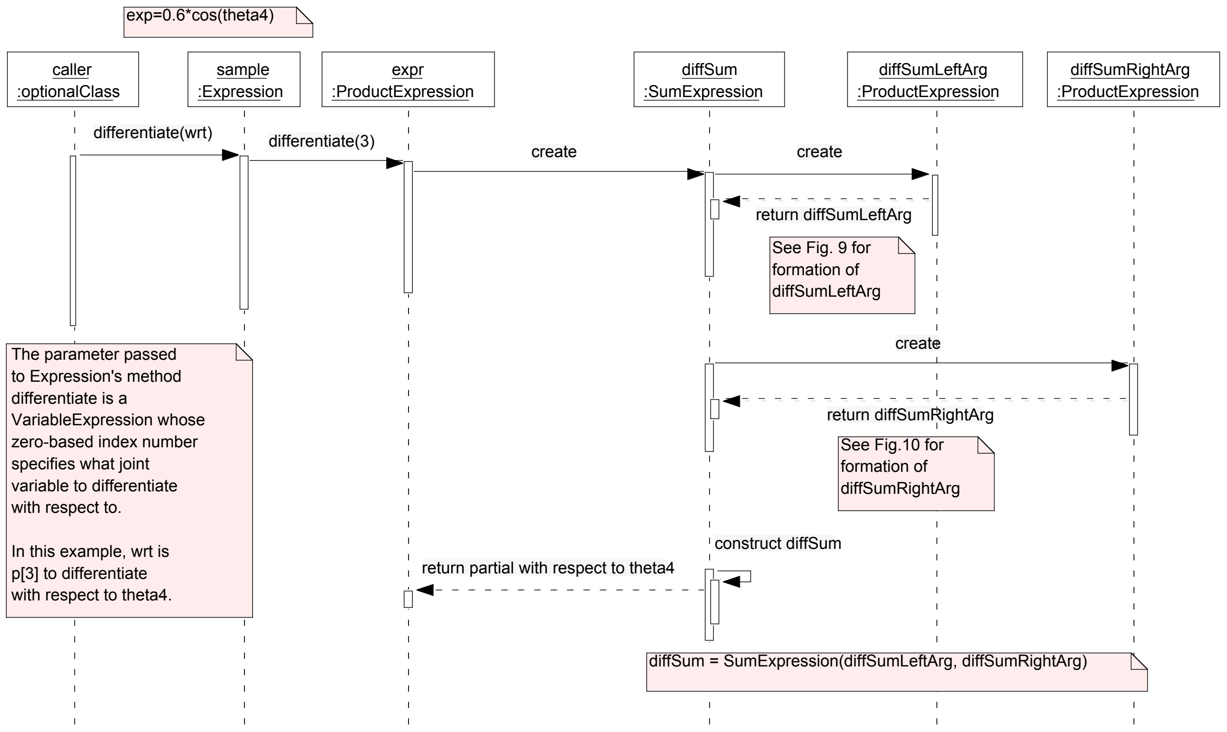
$$\frac{d(\cos q_i)}{dx} = -\sin q_i \bullet \frac{dq_i}{dx}$$

is followed in CosExpression's differentiate method. When the differentiate method of CosExpression is invoked, it returns a newly-created NegateExpression that has as its argument a newly-created ProductExpression. The left argument of the ProductExpression is a SinExpression with its argument being the argument of the CosExpression. The right argument of the ProductExpression is the derivative of the CosExpression argument, q_i.

The derivative of q_i shows another reason why VariableExpression classes need to store an index that indicates which joint variable applies to this VariableExpression. It was noted in Sect. 4.1.2 that the index in a VariableExpression is needed when evaluating the expression tree. It is also needed in taking the derivative of the joint angles with respect to q_i. The derivative with respect to q_i of a VariableExpression representing joint *i* will be 1; it will be 0 with respect to all the other joint variables.

Example 4: Figures 8, 9, and 10 are sequence diagrams that show how calling the differentiate method of an Expression triggers a sequence of calls to the components of the expression tree to produce a new Expression. The new Expression represents the partial derivative of the original Expression with respect to a joint variable. For this example, the same expression was used as that described in Example 2 in Sect. 4.1.1. Refer to Fig. 5 for the expression tree structure for $(0.6 * \cos\theta_4)$. The expression tree of the

Expression to be differentiated is composed of classes ConstantExpression, ProductExpression, CosExpression, and VariableExpression, all subclasses of ExpressionNode.



exp=0.6*cos(theta4)
caller :optionalClass
sample :Expression
expr :ProductExpression
diffSum :SumExpression
diffSumLeftArg :ProductExpression
diffSumRightArg :ProductExpression
differentiate(wrt)
differentiate(3)
create
create
return diffSumLeftArg
See Fig. 9 for formation of diffSumLeftArg
create
return diffSumRightArg
See Fig.10 for formation of diffSumRightArg
The parameter passed to Expression's method differentiate is a VariableExpression whose zero-based index number specifies what joint variable to differentiate with respect to.
In this example, wrt is p[3] to differentiate with respect to theta4.
construct diffSum
return partial with respect to theta4
diffSum = SumExpression(diffSumLeftArg, diffSumRightArg)

Fig. 8. Differentiate expression $(0.6 * \cos\theta_4)$ **with respect to** θ_4**.**

Figure 9 shows the formation of diffSumLeftArg, which forms the remaining portion of the differentiation.

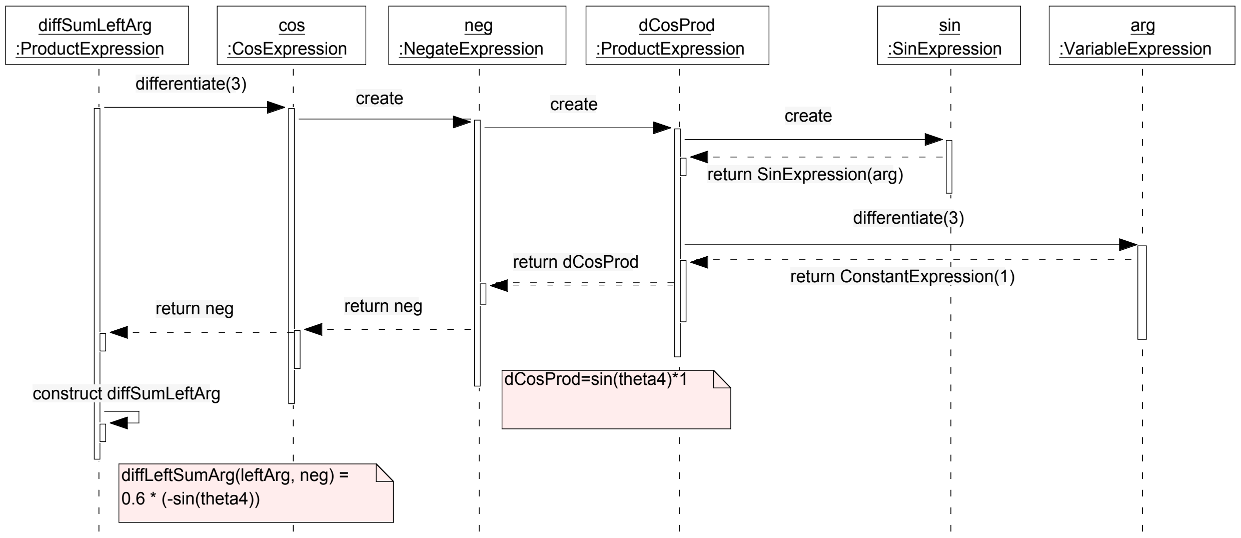
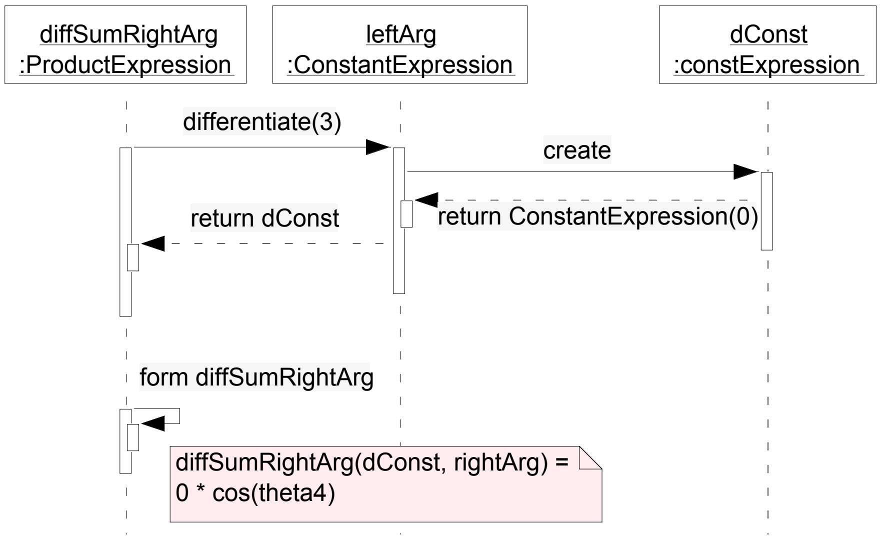


Fig. 9. Formation of $0.6 * \dfrac{d(\cos\theta_4)}{d\theta_4}$**.**

The formation of diffSumRightArg, $\frac{d(0.6)}{d\theta_4}\cos\theta_4$, is shown in Fig. 10. Note that this expression will contain a ConstantExpression containing a value of 0 after the derivative is taken, rather than an integer 0. The simplify method of Expression would need to be called to evaluate the constants and then remove this multiplication by 0.



diffSumRightArg :ProductExpression
leftArg :ConstantExpression
dConst :constExpression
differentiate(3)
create
return ConstantExpression(0)
return dConst
form diffSumRightArg
diffSumRightArg(dConst, rightArg) = 0 * cos(theta4)

Fig. 10. Formation of $\frac{d(0.6)}{d\theta_4}\cos\theta_4$.

A fairly simple form of expression simplification is available through Expression method simplify(). Terms that multiply by 1 or that add 0 are removed, as well as products that contain a multiplication by 0. Future work would include enhancement of the expression simplification.

To summarize this example, a simple expression $(0.6 * \cos\theta_4)$ was differentiated with respect to joint variable θ_4 to produce the expression $(0.6 * (-\sin\theta_4)) + (0 * \cos\theta_4)$. Figure 11 compares the original expression tree to the tree produced from differentiation.

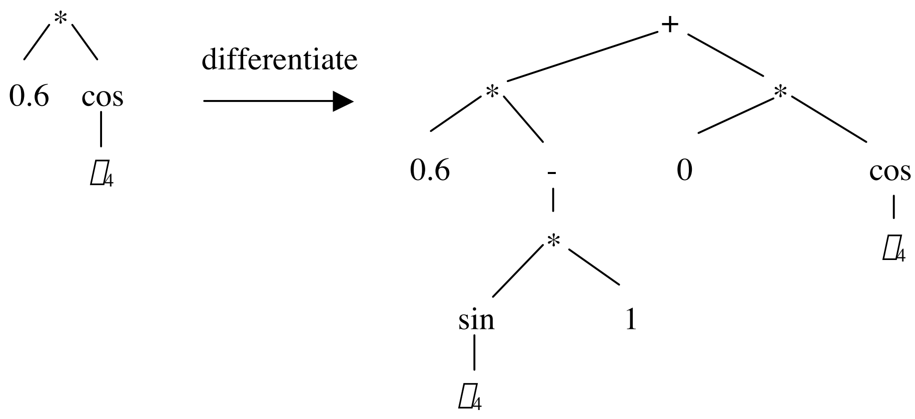


Fig. 11. Expression tree compared to its derivative.

4.1.4. Matrices of Expressions

A C++ type was defined called ExpressionMatrix that allows us to form matrices of expressions. Each element of the matrix is an Expression object, which points to the ExpressionNode at the root of the expression tree for that element. This provides a simple way for referring to elements in equations **A**, **T**, and **J**. The following shows an example of an ExpressionMatrix for **T** for a particular robot after just two joints have been considered:

$$\begin{matrix} (\cos(p[0]) * \cos(p[1])) & (\cos(p[0]) * -\sin(p[1])) & -\sin(p[0]) & ((\cos(p[0]) * (0.4318 * \cos(p[1]))) + (-\sin(p[0]) * 0.14909)) \\ (\sin(p[0] * \cos(p[1])) & (\sin(p[0]) * -\sin(p[1])) & \cos(p[0]) & ((\sin(p[0]) * 0.4318 * \cos(p[1]))) + \cos(p[0]) * 0.14909)) \\ -\sin(p[1]) & -\cos(p[1]) & 0 & -(0.4318 * \sin(p[1])) \\ 0 & 0 & 0 & 1 \end{matrix}$$

The joint variables are written in the expressions above as $p[i]$, where i is the index indicating the joint, and it ranges from 0 to (number of joints – 1).

4.2. ALGORITHMS

This section describes the algorithms employed by the methods of the programming interface: getForwardKinematics, getJacobian, and getJointLocations. These methods make use of member array<□>'s in JFKengine that save the ExpressionMatrices **A**, **T**, and **J**. They also use array<□>'s DH_AT and DH_J, which cache the D-H parameters of the manipulator for which **A**, **T**, or **J** was formed, once they are computed. Flags are set to signal that the forward kinematics or Jacobian has been cached.

4.2.1. getForwardKinematics

A robot::SerialManipulator is passed to this method, as well as the number of joints to use in forming the intermediate **A** expressions (see Eq. (2)) and the expression for **T** (see Eq. (3)). Figure 12 shows the basic steps that are involved in the getForwardKinematics logic. **A** and **T** for each joint are saved in **A**[i] and **T**[i], respectively. The C++ operator overloading features were used to allow Expressions to be added, subtracted, and multiplied, so the code that forms **A**[i] resembles procedural code that performs calculations. It is actually forming a symbolic expression, rather than returning a numeric value. The multiply operator (*) was also overloaded for ExpressionMatrix and ExpressionVector, so that the code simply does T[i] = T[i-1] * A[i], as shown below, to form the manipulator arm equation expressions up to the *i*th joint.

4.2.2. getJacobian

A robot::SerialManipulator is passed to getJacobian, as well as the number of joints to use and a flag indicating whether to include orientation. If the Jacobian has already been formed and cached by a previous call to getJacobian, the cached D-H parameters for each joint are compared with those of the manipulator. If they are all found to be the same, then the cached Jacobian is returned.

If the cached Jacobian cannot be used, then a new one will be formed. Figure 13 shows the basic steps that are involved in the getJacobian logic. The manipulator and the number of joints to use are passed to getForwardKinematics to form the expression for **T** (see Eq. (3)). Then the D-H parameters for this joint are saved in the DH_J array.

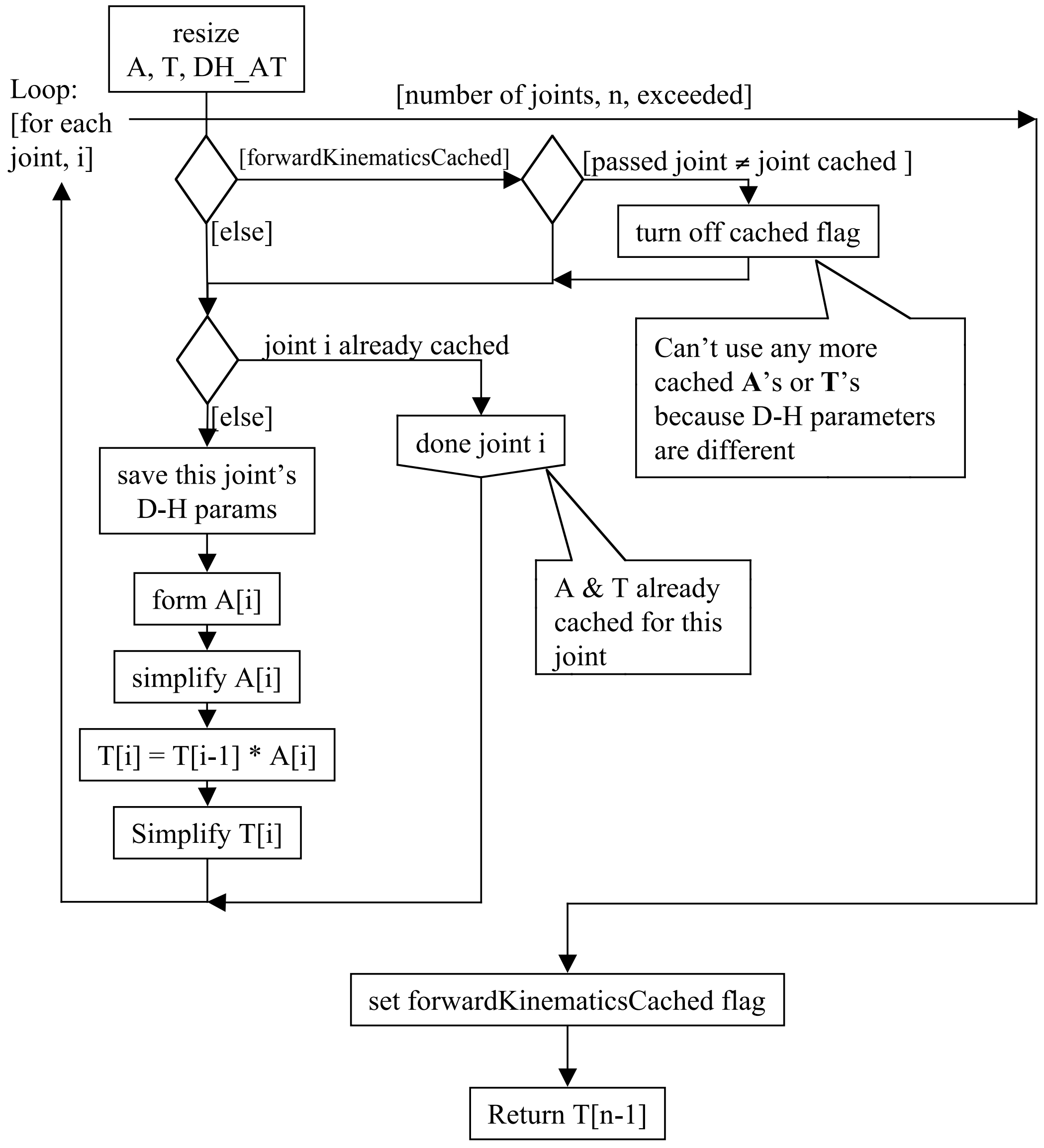

resize
A, T, DH_AT
Loop:
[for each
joint, i]
[number of joints, n, exceeded]
[forwardKinematicsCached]
[passed joint ≠ joint cached]
turn off cached flag
[else]
joint i already cached
Can't use any more cached A's or T's because D-H parameters are different
[else]
done joint i
save this joint's D-H params
form A[i]
A & T already cached for this joint
simplify A[i]
T[i] = T[i-1] * A[i]
Simplify T[i]
set forwardKinematicsCached flag
Return T[n-1]

Fig. 12. Algorithm for getForwardKinematics.

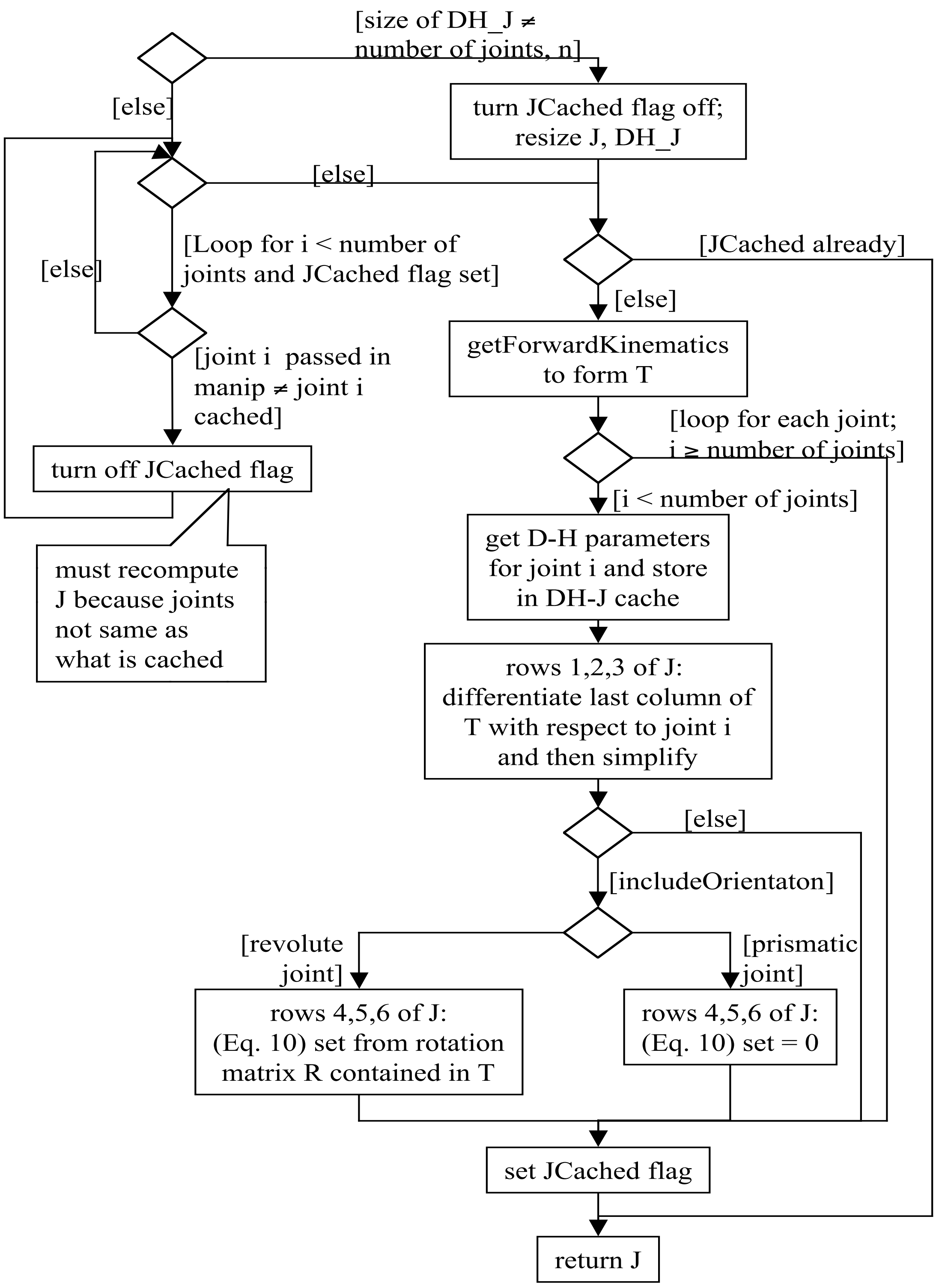

[size of DH_J ≠ number of joints, n]
turn JCached flag off; resize J, DH_J
[else]
[else]
[Loop for i < number of joints and JCached flag set]
[else]
[joint i passed in manip ≠ joint i cached]
turn off JCached flag
must recompute J because joints not same as what is cached
[JCached already]
[else]
getForwardKinematics to form T
[loop for each joint; i ≥ number of joints]
[i < number of joints]
get D-H parameters for joint i and store in DH-J cache
rows 1,2,3 of J: differentiate last column of T with respect to joint i and then simplify
[else]
[includeOrientaton]
[revolute joint]
[prismatic joint]
rows 4,5,6 of J: (Eq. 10) set from rotation matrix R contained in T
rows 4,5,6 of J: (Eq. 10) set = 0
set JCached flag
return J

Fig. 13. Algorithm for getJacobian.

The Jacobian is formed one column at a time. The first column represents the contribution due to joint one, and so forth. $\mathbf{J}_{Li}$, the first three rows of **J**, are associated with the linear velocity. They are produced by taking the partial derivative of the position expressions (found in the last column of **T**) with respect to the joint variable that corresponds to the column of **J** being formed (see Eq. (5)). If orientation is to be included, then the last three rows of the column are formed according to Eq. (10). The manipulator contains the type of joint for each type; revolute and prismatic are the types supported. If the joint type is prismatic, a value of zero is assigned to rows 4, 5, and 6. If the type is revolute, the rotation submatrix of **T** is to be multiplied by a vector representing the *z*-axis, $[0,0,1]^T$. Multiplying the rotation matrix by $[0,0,1]^T$ causes the terms due to columns one and two of **T** to drop out, due to multiplying by 0. Therefore, just the third column of **T** can be used directly. This process is repeated for each joint to produce each column in **J**. After all the columns are formed, the flag is set to signal that **J** is cached, and then **J** is returned.

4.2.3. getJointLocations

A robot::SerialManipulator is passed to getJointLocations, as well as a Vector reference that contains the values of the joint variables that should be used for evaluating the forward kinematics expressions. It returns an array<Vector> which contains the [x,y,z] location of the coordinate frame of each link. This corresponds to the end of the link or the joint location. Figure 14 shows the algorithm used by getJointLocations.

4.3. LIBRARIES

The uBLAS[10] open source software was used for the vector and matrix operations. uBLAS is a C++ template class library that provides Basic Linear Algebra Subprograms (BLAS[11]) levels 1, 2, and 3 functionality for dense, packed and sparse matrices. The design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. uBLAS is now available from Boost.[12] The Boost web site provides free peer-reviewed portable C++ source libraries. The emphasis is on libraries that work well with the C++ Standard Library.

JFKengine is also built upon a framework of classes that provide useful basic facilities, including storage management features such as smart object pointers. The technique of encapsulating design decisions in policy classes that are passed to C++ templates, first proposed by Alexandrescu,[13] is used throughout the basic facilities. A modified version of Alexandrescu's implementation of smart pointers is used for array<☐> and ref<☐> (see the Appendix). These are part of the open source simulation framework OpenSim/IKOR that is being developed. An overall robot model framework is also part of OpenSim/IKOR. In addition to class JFKengine, the expression tree classes, and the basic facilities, JFKengine receives an argument that is an instance of class SerialManipulator from the OpenSim/IKOR framework. This document reflects the state of the framework corresponding to Version 3.0 of OpenSim/IKOR.

4.4. PLATFORMS

JFKengine was developed in a Linux environment (Red Hat 7.3 and Red Hat 8.0).[15] It is written in ISO/ANSI Standard C++ (2002),[16] and the GNU Compiler Collection (gcc),[17] versions 3.1 and 3.2 were used. The Concurrent Versions System (CVS)[18] was used to manage the source code, build files, and documentation generation control files. CVS tag "IKOR3_0" contains the source code and libraries corresponding to this document.

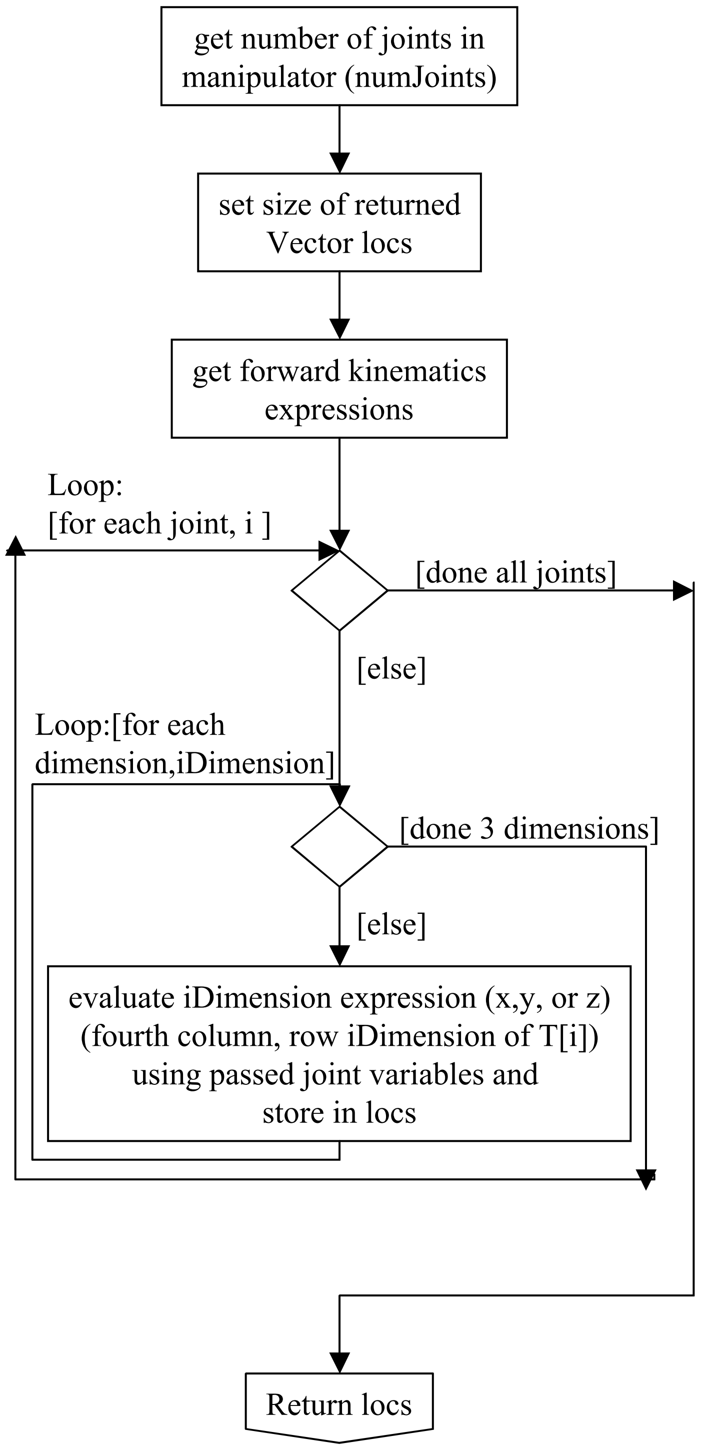

get number of joints in manipulator (numJoints)
set size of returned Vector locs
get forward kinematics expressions
Loop: [for each joint, i]
[done all joints]
[else]
Loop:[for each dimension,iDimension]
[done 3 dimensions]
[else]
evaluate iDimension expression (x,y, or z) (fourth column, row iDimension of T[i]) using passed joint variables and store in locs
Return locs

Fig. 14. Algorithm for getJointLocations.

4.5. DOCUMENTATION

The open source documenting tool Doxygen[19] was used to generate online source code documentation. Doxygen is a documentation system for C and C++ that generates an online class browser in HTML and/or an off-line reference manual in LaTeX from source files. Section 3, Programming Interface, of this report, was cut and pasted from the documentation produced by Doxygen. The UML diagrams of Sects. 4.1 and 4.2 were produced by MetaMill.[20] Metamill is a UML software tool for designing systems using UML as a modeling language. Its reverse engineering features were used to produce the class hierarchy diagrams in the Appendix, as well.

4.6. BUILDING

Boost Jam,[21-22] which is an open source replacement for the Unix make facility, was used to build debug or release versions of the code. Jam uses a Jamrules file located in the root directory for the project to specify building configuration defaults. It also uses a Jamfile located in each source directory that specifies what source code is to be compiled and linked.

4.7. FUTURE

This report refers to the first C++ implementation of JFKengine. It forms the equations **A**, **T**, and **J** using expression trees that may be evaluated or differentiated, given the D-H parameters of the manipulator arm. Having done this initial implementation in the last quarter of Fiscal Year (FY) 2002, changes and areas to investigate have come to mind for future work for the next FY. These are documented in the source code using Doxygen codes (/// \todo). The composite list is found off the Related Links menu item in the online documentation.

REFERENCES

1. Denavit, J. and R. S. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *ASME Journal of Applied Mechanics*, Vol. 22, No. 2, June 1955, pp. 215-221.

2. Slotine, J. -J. E. and H. Asada, *Robot Analysis and Control,* John Wiley & Sons, 1986.

3. http://www.ar2.com/puma500.html

4. Fu, K. S., Rafael C. Gonzalez, and C. S. G. Lee, *Robotics : Control, Sensing, Vision, and Intelligence,* New York : McGraw-Hill, 1987.

5. MATLAB^{TM} is a registered trademark of The MathWorks, Inc. See http://www.mathworks.com/products/prodoverview.shtml

6. Weiss, Mark Allen, *Data Structures & Algorithm Analysis in C++*, Addison Wesley Longman, 1999.

7. Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1997.

8. Fowler, Martin and Kendall Scott, *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2000.

9. UML Resource web site is located at http://www.omg.org/uml/

10. http://www.genesys-e.org/ublas/

11. http://www.netlib.org/blas/

12. http://www.boost.org/

13. Alexandrescu, Andrei, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley, 2001.

14. http://www.linux.org/

15. http://www.redhat.com/

16. http://www.comeaucomputing.com/iso/

17. http://gcc.gnu.org/

18. http://www.cvshome.org/

19. http://www.stack.nl/~dimitri/doxygen/

20. http://www.metamill.com

21. http://www.boost.org/tools/build/build_system.htm#1

22. http://freetype.sourceforge.net/jam/index.html

APPENDIX

Diagrams showing the class inheritance hierarchy, member attributes (data) and methods (functions) for the software associated with JFKengine are included on the indicated pages:

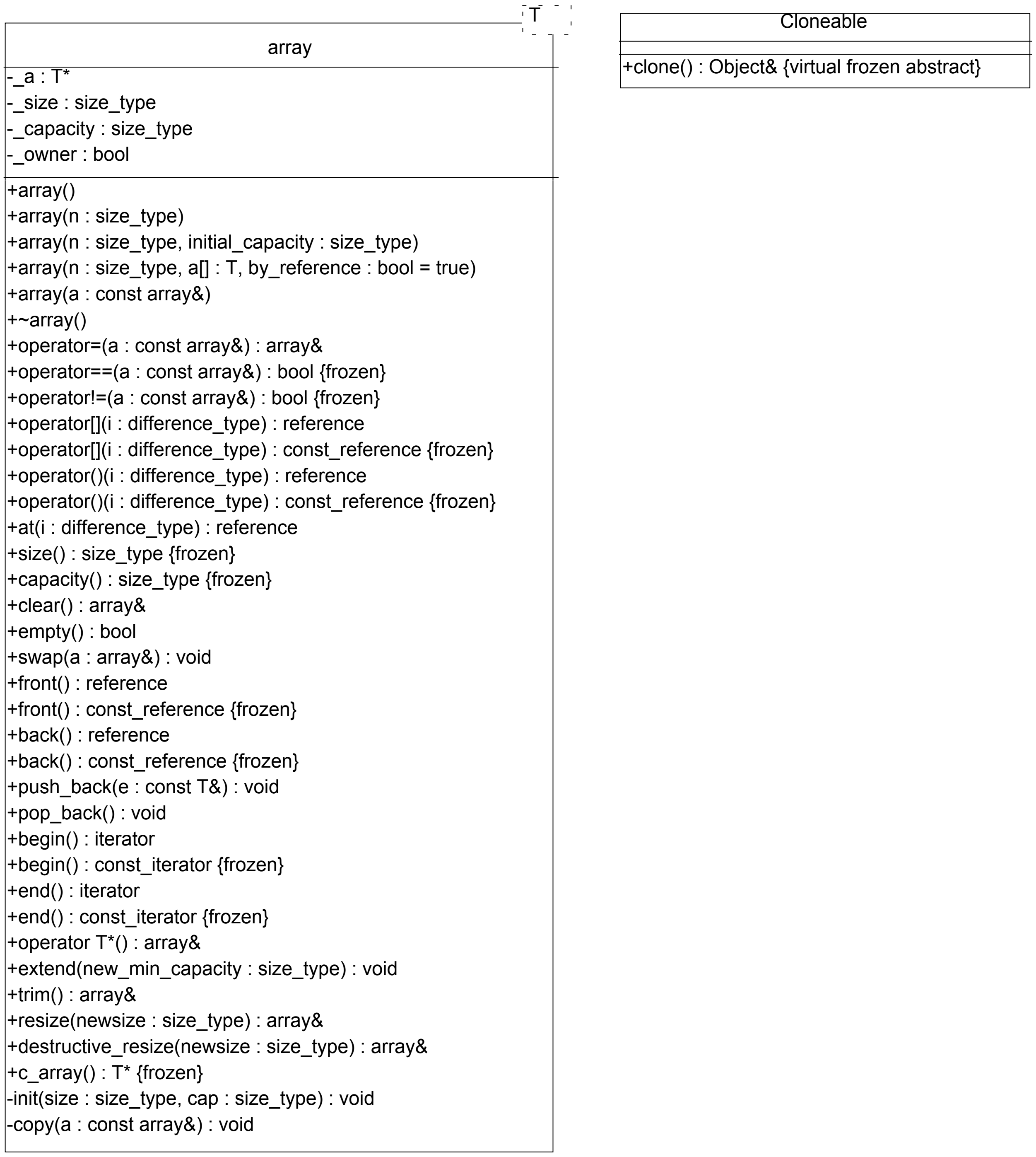
A few notes are included here on the Unified Modeling Language (UML) class diagram notation. The top compartment gives the class name. The middle compartment lists the class's data members. The bottom compartment lists the class's methods. The arrows indicate inheritance. The arrow is drawn from the subclass to the superclass, where the open triangle is placed at the superclass. The variables and methods are listed by:

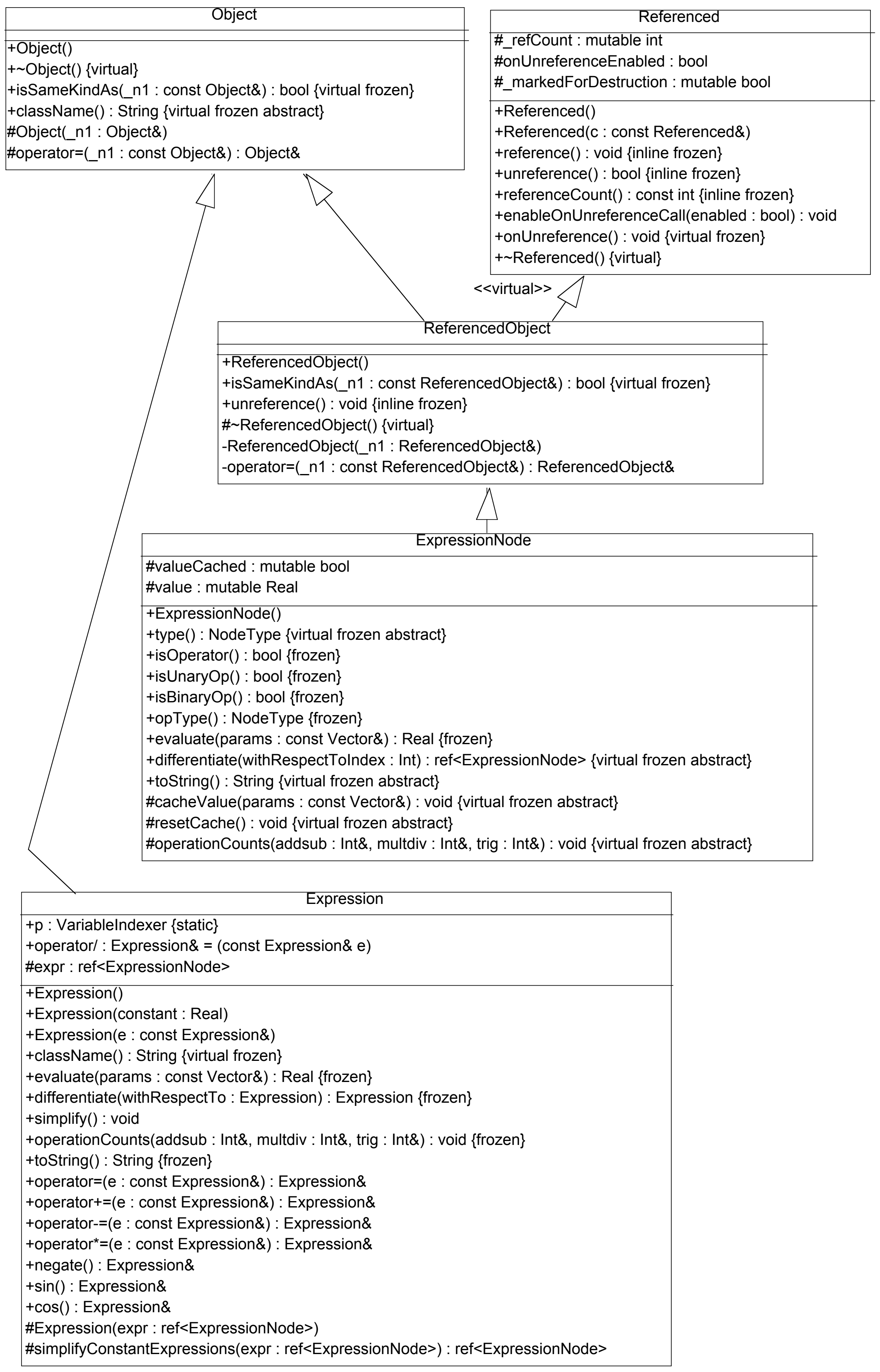
accessControlIndicator itemName : *itemType*

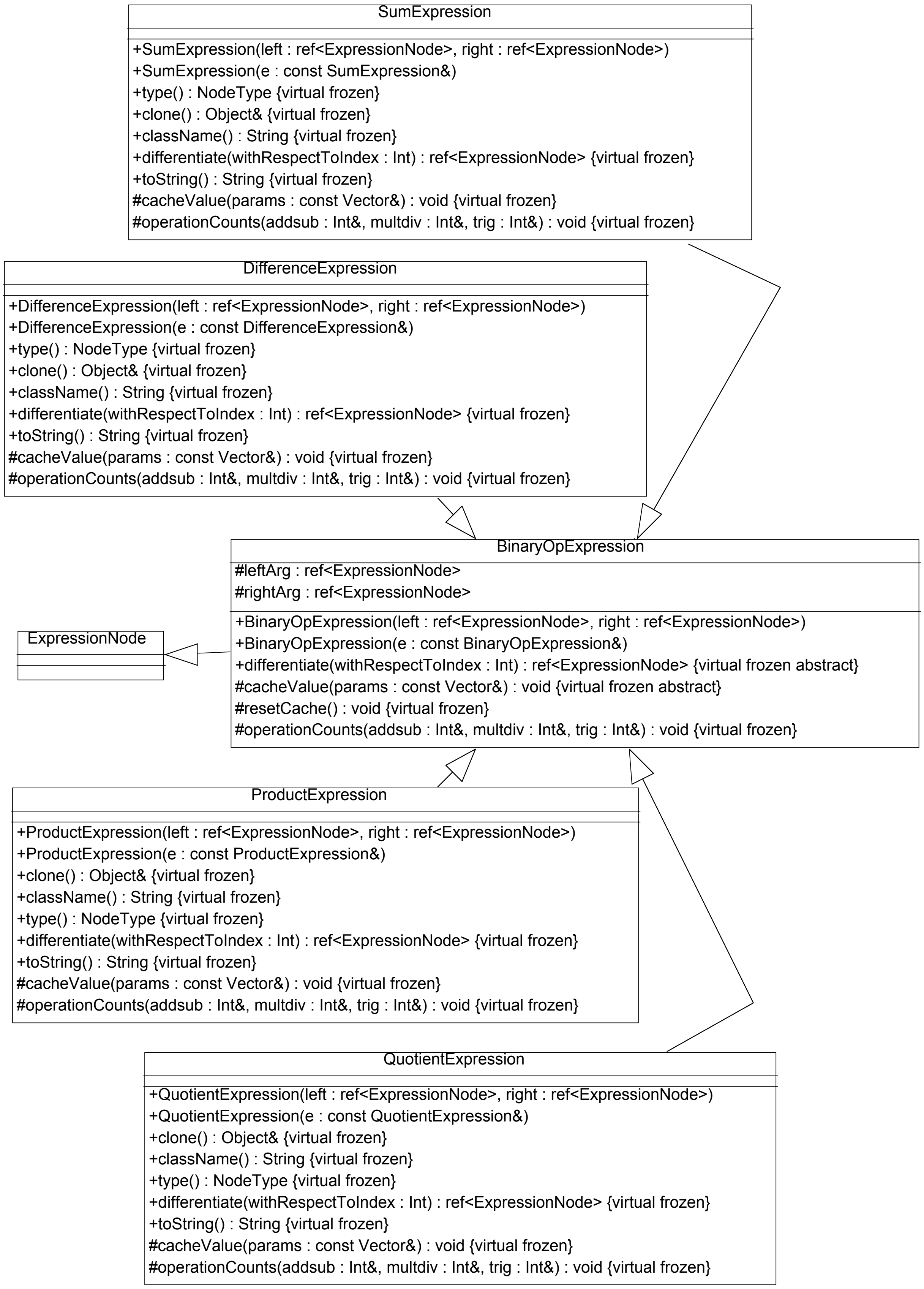
where *accessControlIndicator* shows what kind of access is allowed for the item, as follows:

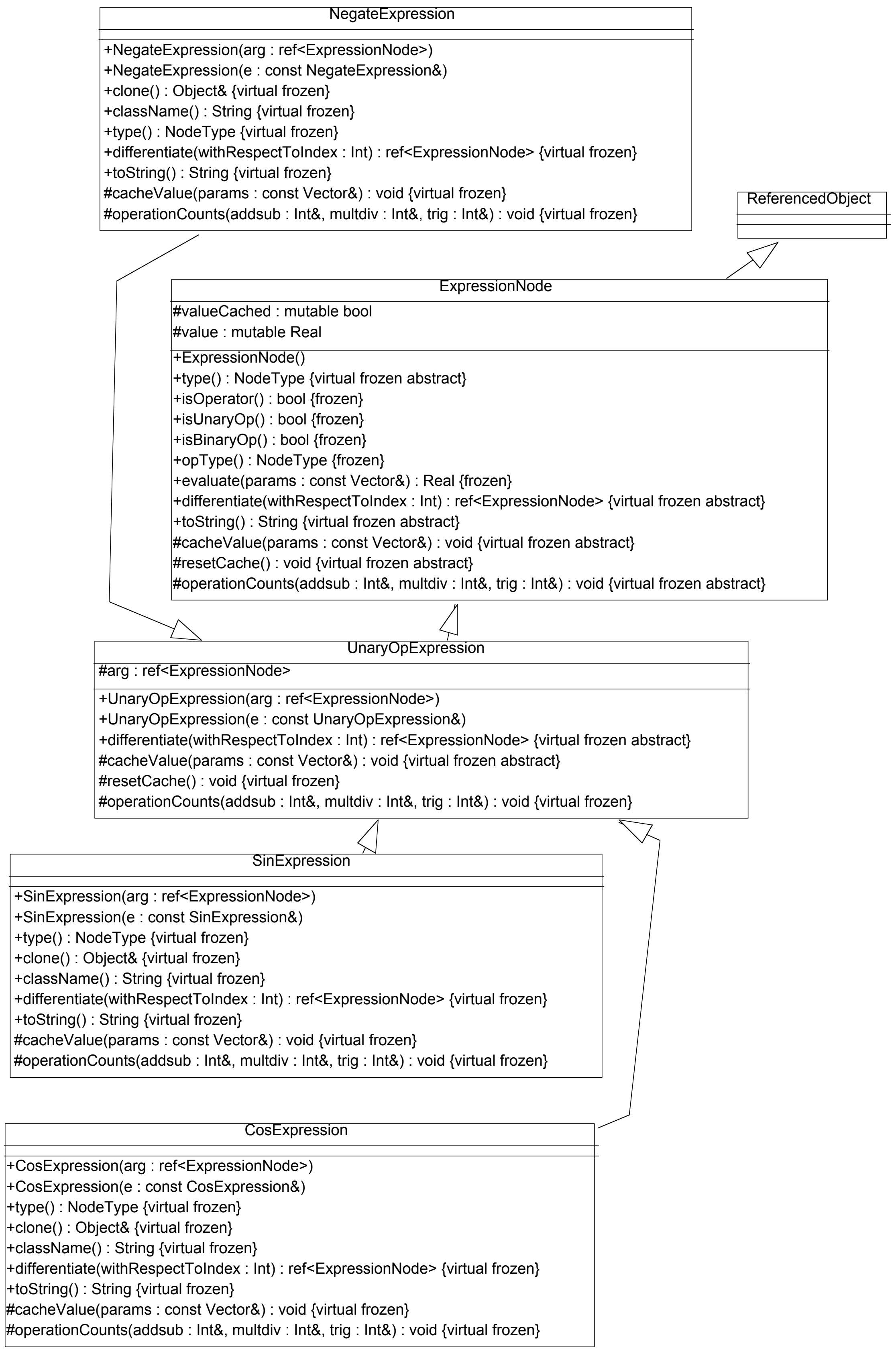
Symbol	**Access**	**Use Permitted**
+	public	can be used by any function
#	protected	can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class
-	private	can be used only by member functions and friends of the class in which it is declared

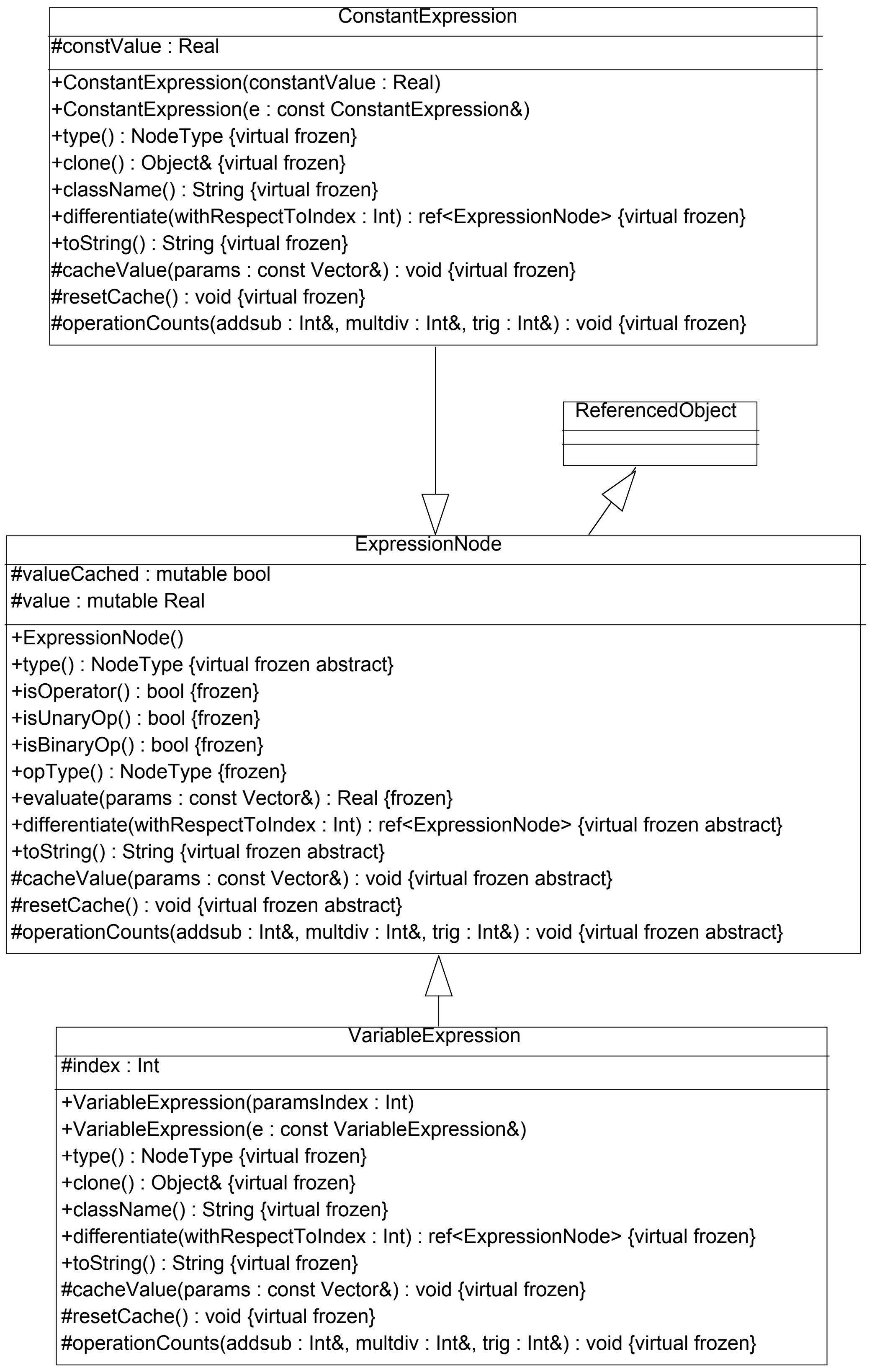
and *itemName* is the name of the variable or method. The type of the variable is given by *itemType*. For methods, *itemType* shows the type of the returned value. Methods also include, within parentheses, the name and type of the formal parameters, separated by commas.

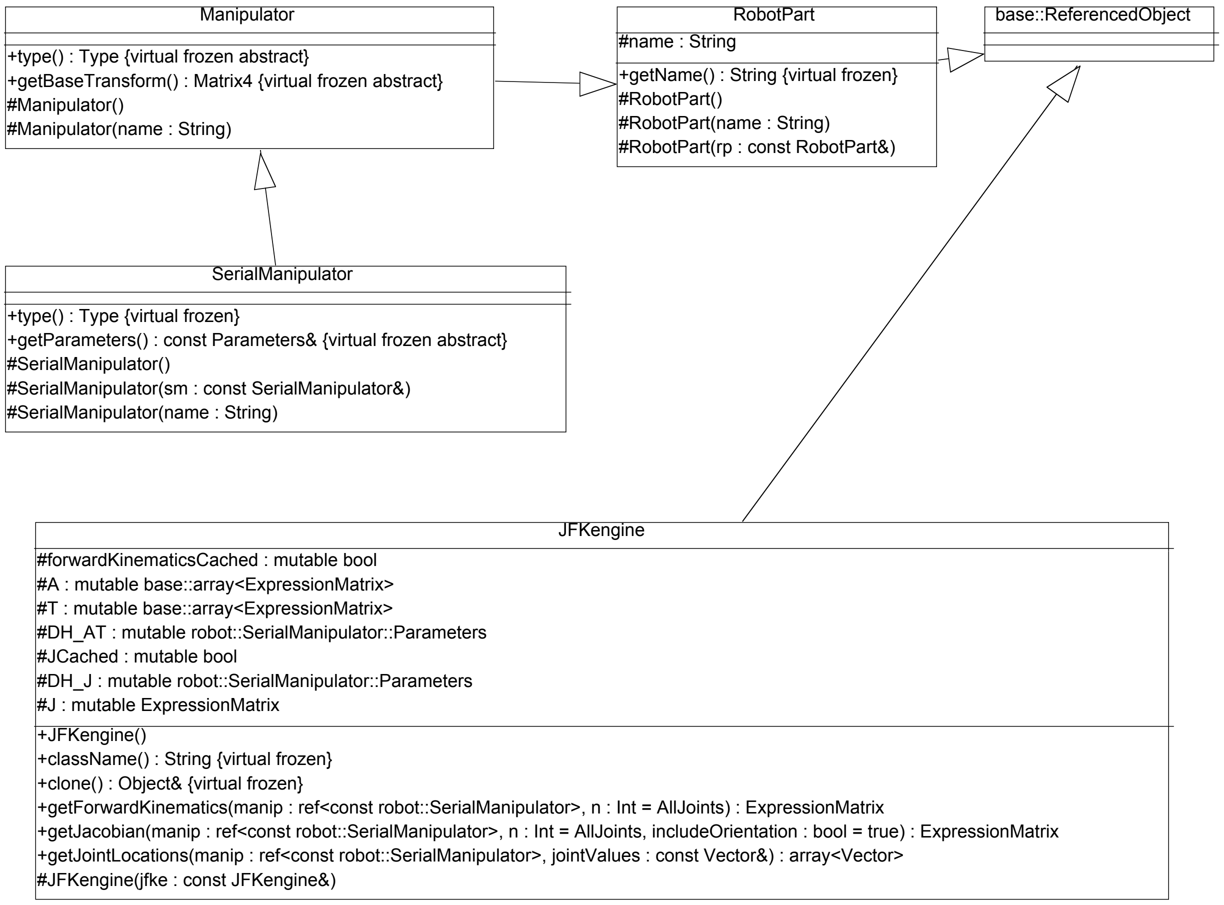

T
array
-_a : T*
-_size : size_type
-_capacity : size_type
-_owner : bool
+array()
+array(n : size_type)
+array(n : size_type, initial_capacity : size_type)
+array(n : size_type, a[] : T, by_reference : bool = true)
+array(a : const array&)
+~array()
+operator=(a : const array&) : array&
+operator==(a : const array&) : bool {frozen}
+operator!=(a : const array&) : bool {frozen}
+operator[](i : difference_type) : reference
+operator[](i : difference_type) : const_reference {frozen}
+operator()(i : difference_type) : reference
+operator()(i : difference_type) : const_reference {frozen}
+at(i : difference_type) : reference
+size() : size_type {frozen}
+capacity() : size_type {frozen}
+clear() : array&
+empty() : bool
+swap(a : array&) : void
+front() : reference
+front() : const_reference {frozen}
+back() : reference
+back() : const_reference {frozen}
+push_back(e : const T&) : void
+pop_back() : void
+begin() : iterator
+begin() : const_iterator {frozen}
+end() : iterator
+end() : const_iterator {frozen}
+operator T*() : array&
+extend(new_min_capacity : size_type) : void
+trim() : array&
+resize(newsize : size_type) : array&
+destructive_resize(newsize : size_type) : array&
+c_array() : T* {frozen}
-init(size : size_type, cap : size_type) : void
-copy(a : const array&) : void
Cloneable
+clone() : Object& {virtual frozen abstract}

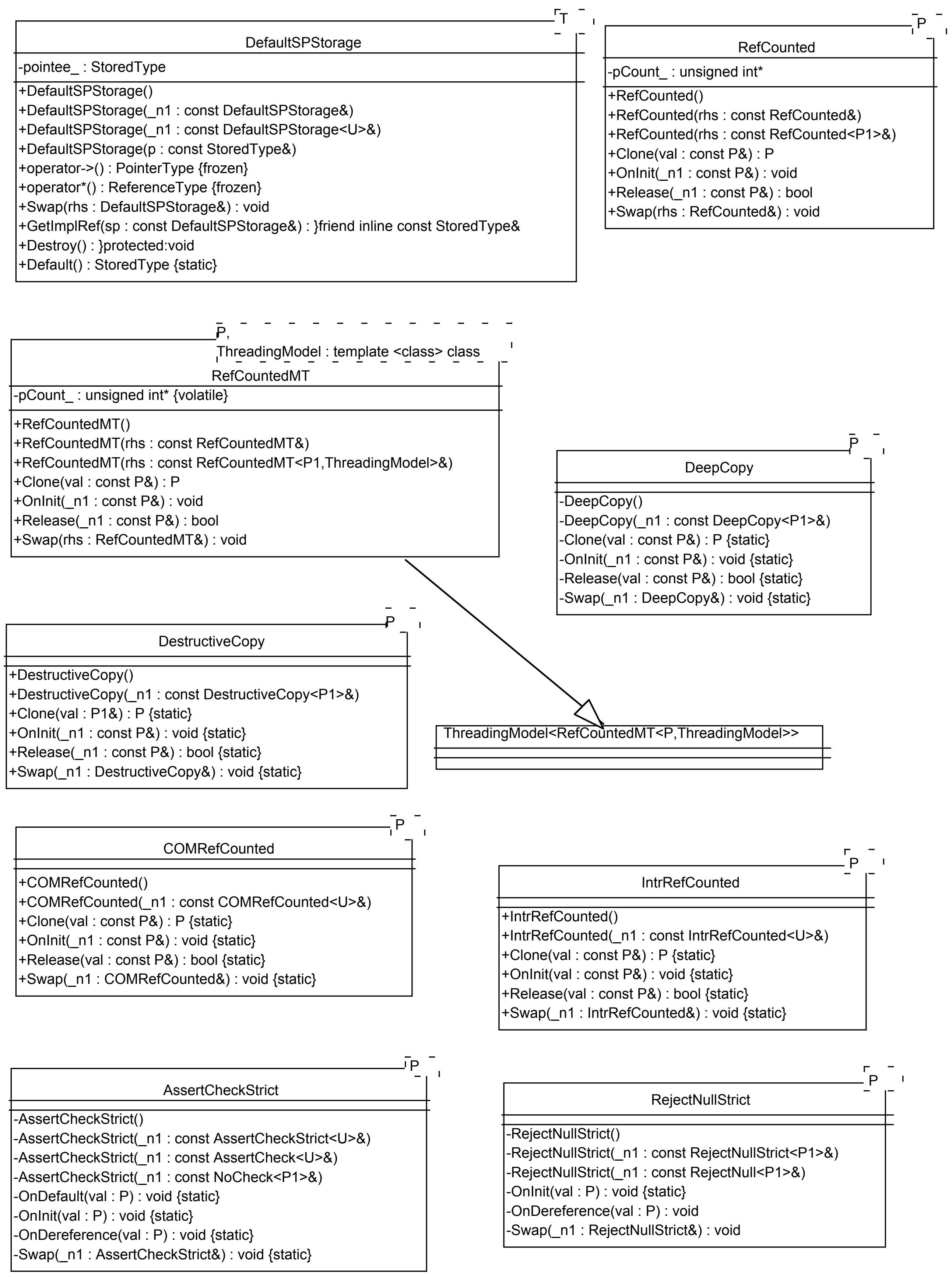

Object
+Object()
+~Object() {virtual}
+isSameKindAs(_n1 : const Object&) : bool {virtual frozen}
+className() : String {virtual frozen abstract}
#Object(_n1 : Object&)
#operator=(_n1 : const Object&) : Object&
Referenced
#_refCount : mutable int
#onUnreferenceEnabled : bool
#_markedForDestruction : mutable bool
+Referenced()
+Referenced(c : const Referenced&)
+reference() : void {inline frozen}
+unreference() : bool {inline frozen}
+referenceCount() : const int {inline frozen}
+enableOnUnreferenceCall(enabled : bool) : void
+onUnreference() : void {virtual frozen}
+~Referenced() {virtual}
<<virtual>>
ReferencedObject
+ReferencedObject()
+isSameKindAs(_n1 : const ReferencedObject&) : bool {virtual frozen}
+unreference() : void {inline frozen}
#~ReferencedObject() {virtual}
-ReferencedObject(_n1 : ReferencedObject&)
-operator=(_n1 : const ReferencedObject&) : ReferencedObject&
ExpressionNode
#valueCached : mutable bool
#value : mutable Real
+ExpressionNode()
+type() : NodeType {virtual frozen abstract}
+isOperator() : bool {frozen}
+isUnaryOp() : bool {frozen}
+isBinaryOp() : bool {frozen}
+opType() : NodeType {frozen}
+evaluate(params : const Vector&) : Real {frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen abstract}
+toString() : String {virtual frozen abstract}
#cacheValue(params : const Vector&) : void {virtual frozen abstract}
#resetCache() : void {virtual frozen abstract}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen abstract}
Expression
+p : VariableIndexer {static}
+operator/ : Expression& = (const Expression& e)
#expr : ref<ExpressionNode>
+Expression()
+Expression(constant : Real)
+Expression(e : const Expression&)
+className() : String {virtual frozen}
+evaluate(params : const Vector&) : Real {frozen}
+differentiate(withRespectTo : Expression) : Expression {frozen}
+simplify() : void
+operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {frozen}
+toString() : String {frozen}
+operator=(e : const Expression&) : Expression&
+operator+=(e : const Expression&) : Expression&
+operator-=(e : const Expression&) : Expression&
+operator*=(e : const Expression&) : Expression&
+negate() : Expression&
+sin() : Expression&
+cos() : Expression&
#Expression(expr : ref<ExpressionNode>)
#simplifyConstantExpressions(expr : ref<ExpressionNode>) : ref<ExpressionNode>

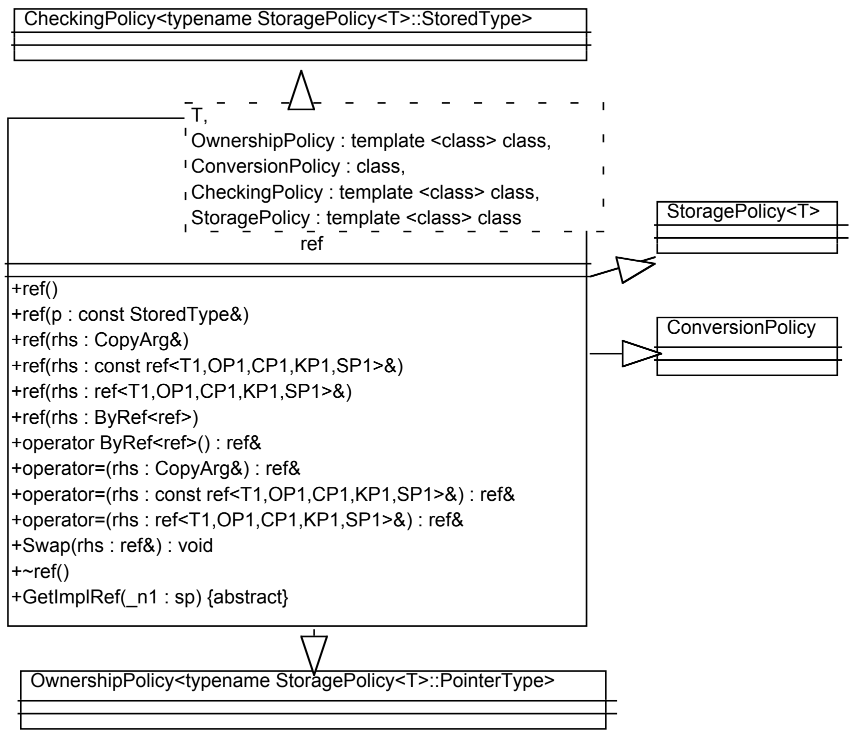

SumExpression
+SumExpression(left : ref<ExpressionNode>, right : ref<ExpressionNode>)
+SumExpression(e : const SumExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
DifferenceExpression
+DifferenceExpression(left : ref<ExpressionNode>, right : ref<ExpressionNode>)
+DifferenceExpression(e : const DifferenceExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
BinaryOpExpression
#leftArg : ref<ExpressionNode>
#rightArg : ref<ExpressionNode>
+BinaryOpExpression(left : ref<ExpressionNode>, right : ref<ExpressionNode>)
+BinaryOpExpression(e : const BinaryOpExpression&)
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen abstract}
#cacheValue(params : const Vector&) : void {virtual frozen abstract}
#resetCache() : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
ExpressionNode
ProductExpression
+ProductExpression(left : ref<ExpressionNode>, right : ref<ExpressionNode>)
+ProductExpression(e : const ProductExpression&)
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+type() : NodeType {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
QuotientExpression
+QuotientExpression(left : ref<ExpressionNode>, right : ref<ExpressionNode>)
+QuotientExpression(e : const QuotientExpression&)
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+type() : NodeType {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}

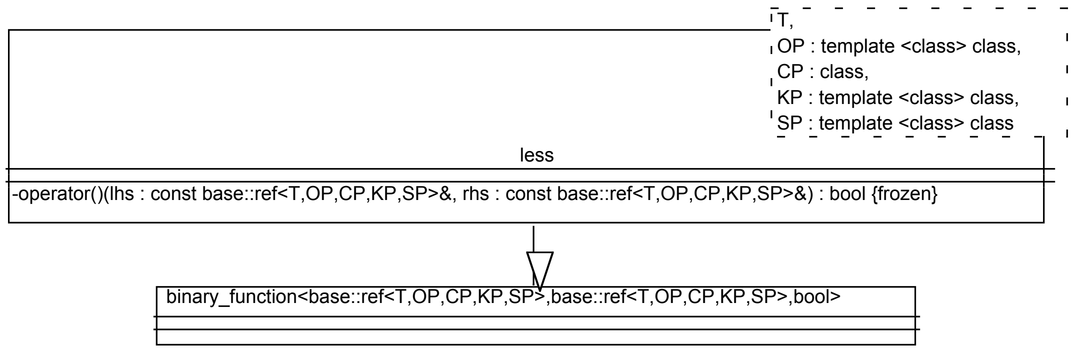

NegateExpression
+NegateExpression(arg : ref<ExpressionNode>)
+NegateExpression(e : const NegateExpression&)
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+type() : NodeType {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
ReferencedObject
ExpressionNode
#valueCached : mutable bool
#value : mutable Real
+ExpressionNode()
+type() : NodeType {virtual frozen abstract}
+isOperator() : bool {frozen}
+isUnaryOp() : bool {frozen}
+isBinaryOp() : bool {frozen}
+opType() : NodeType {frozen}
+evaluate(params : const Vector&) : Real {frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen abstract}
+toString() : String {virtual frozen abstract}
#cacheValue(params : const Vector&) : void {virtual frozen abstract}
#resetCache() : void {virtual frozen abstract}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen abstract}
UnaryOpExpression
#arg : ref<ExpressionNode>
+UnaryOpExpression(arg : ref<ExpressionNode>)
+UnaryOpExpression(e : const UnaryOpExpression&)
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen abstract}
#cacheValue(params : const Vector&) : void {virtual frozen abstract}
#resetCache() : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
SinExpression
+SinExpression(arg : ref<ExpressionNode>)
+SinExpression(e : const SinExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
CosExpression
+CosExpression(arg : ref<ExpressionNode>)
+CosExpression(e : const CosExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}

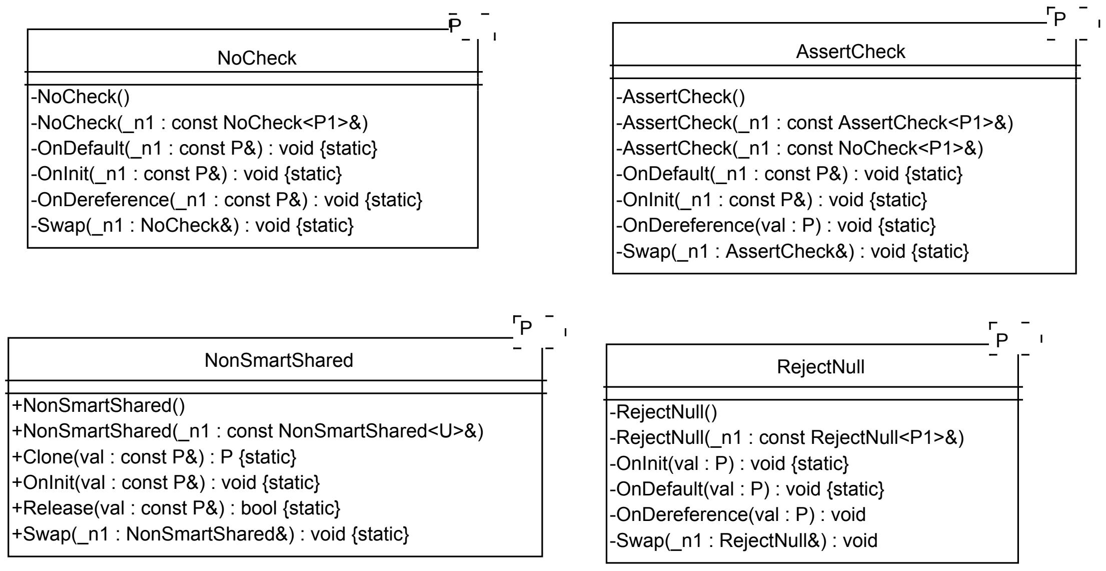

ConstantExpression
#constValue : Real
+ConstantExpression(constantValue : Real)
+ConstantExpression(e : const ConstantExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#resetCache() : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}
ReferencedObject
ExpressionNode
#valueCached : mutable bool
#value : mutable Real
+ExpressionNode()
+type() : NodeType {virtual frozen abstract}
+isOperator() : bool {frozen}
+isUnaryOp() : bool {frozen}
+isBinaryOp() : bool {frozen}
+opType() : NodeType {frozen}
+evaluate(params : const Vector&) : Real {frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen abstract}
+toString() : String {virtual frozen abstract}
#cacheValue(params : const Vector&) : void {virtual frozen abstract}
#resetCache() : void {virtual frozen abstract}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen abstract}
VariableExpression
#index : Int
+VariableExpression(paramsIndex : Int)
+VariableExpression(e : const VariableExpression&)
+type() : NodeType {virtual frozen}
+clone() : Object& {virtual frozen}
+className() : String {virtual frozen}
+differentiate(withRespectToIndex : Int) : ref<ExpressionNode> {virtual frozen}
+toString() : String {virtual frozen}
#cacheValue(params : const Vector&) : void {virtual frozen}
#resetCache() : void {virtual frozen}
#operationCounts(addsub : Int&, multdiv : Int&, trig : Int&) : void {virtual frozen}

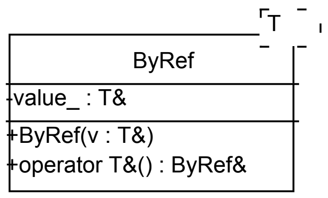

Manipulator
+type() : Type {virtual frozen abstract}
+getBaseTransform() : Matrix4 {virtual frozen abstract}
#Manipulator()
#Manipulator(name : String)
RobotPart
#name : String
+getName() : String {virtual frozen}
#RobotPart()
#RobotPart(name : String)
#RobotPart(rp : const RobotPart&)
base::ReferencedObject
SerialManipulator
+type() : Type {virtual frozen}
+getParameters() : const Parameters& {virtual frozen abstract}
#SerialManipulator()
#SerialManipulator(sm : const SerialManipulator&)
#SerialManipulator(name : String)
JFKengine
#forwardKinematicsCached : mutable bool
#A : mutable base::array<ExpressionMatrix>
#T : mutable base::array<ExpressionMatrix>
#DH_AT : mutable robot::SerialManipulator::Parameters
#JCached : mutable bool
#DH_J : mutable robot::SerialManipulator::Parameters
#J : mutable ExpressionMatrix
+JFKengine()
+className() : String {virtual frozen}
+clone() : Object& {virtual frozen}
+getForwardKinematics(manip : ref<const robot::SerialManipulator>, n : Int = AllJoints) : ExpressionMatrix
+getJacobian(manip : ref<const robot::SerialManipulator>, n : Int = AllJoints, includeOrientation : bool = true) : ExpressionMatrix
+getJointLocations(manip : ref<const robot::SerialManipulator>, jointValues : const Vector&) : array<Vector>
#JFKengine(jfke : const JFKengine&)

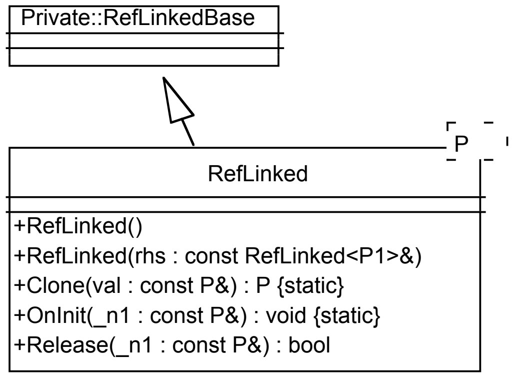

T
DefaultSPStorage
-pointee_ : StoredType
+DefaultSPStorage()
+DefaultSPStorage(_n1 : const DefaultSPStorage&)
+DefaultSPStorage(_n1 : const DefaultSPStorage<U>&)
+DefaultSPStorage(p : const StoredType&)
+operator->() : PointerType {frozen}
+operator*() : ReferenceType {frozen}
+Swap(rhs : DefaultSPStorage&) : void
+GetImplRef(sp : const DefaultSPStorage&) : }friend inline const StoredType&
+Destroy() : }protected:void
+Default() : StoredType {static}
P
RefCounted
-pCount_ : unsigned int*
+RefCounted()
+RefCounted(rhs : const RefCounted&)
+RefCounted(rhs : const RefCounted<P1>&)
+Clone(val : const P&) : P
+OnInit(_n1 : const P&) : void
+Release(_n1 : const P&) : bool
+Swap(rhs : RefCounted&) : void
P,
ThreadingModel : template <class> class
RefCountedMT
-pCount_ : unsigned int* {volatile}
+RefCountedMT()
+RefCountedMT(rhs : const RefCountedMT&)
+RefCountedMT(rhs : const RefCountedMT<P1,ThreadingModel>&)
+Clone(val : const P&) : P
+OnInit(_n1 : const P&) : void
+Release(_n1 : const P&) : bool
+Swap(rhs : RefCountedMT&) : void
P
DeepCopy
-DeepCopy()
-DeepCopy(_n1 : const DeepCopy<P1>&)
-Clone(val : const P&) : P {static}
-OnInit(_n1 : const P&) : void {static}
-Release(val : const P&) : bool {static}
-Swap(_n1 : DeepCopy&) : void {static}
P
DestructiveCopy
+DestructiveCopy()
+DestructiveCopy(_n1 : const DestructiveCopy<P1>&)
+Clone(val : P1&) : P {static}
+OnInit(_n1 : const P&) : void {static}
+Release(_n1 : const P&) : bool {static}
+Swap(_n1 : DestructiveCopy&) : void {static}
ThreadingModel<RefCountedMT<P,ThreadingModel>>
P
COMRefCounted
+COMRefCounted()
+COMRefCounted(_n1 : const COMRefCounted<U>&)
+Clone(val : const P&) : P {static}
+OnInit(_n1 : const P&) : void {static}
+Release(val : const P&) : bool {static}
+Swap(_n1 : COMRefCounted&) : void {static}
P
IntrRefCounted
+IntrRefCounted()
+IntrRefCounted(_n1 : const IntrRefCounted<U>&)
+Clone(val : const P&) : P {static}
+OnInit(val : const P&) : void {static}
+Release(val : const P&) : bool {static}
+Swap(_n1 : IntrRefCounted&) : void {static}
P
AssertCheckStrict
-AssertCheckStrict()
-AssertCheckStrict(_n1 : const AssertCheckStrict<U>&)
-AssertCheckStrict(_n1 : const AssertCheck<U>&)
-AssertCheckStrict(_n1 : const NoCheck<P1>&)
-OnDefault(val : P) : void {static}
-OnInit(val : P) : void {static}
-OnDereference(val : P) : void {static}
-Swap(_n1 : AssertCheckStrict&) : void {static}
P
RejectNullStrict
-RejectNullStrict()
-RejectNullStrict(_n1 : const RejectNullStrict<P1>&)
-RejectNullStrict(_n1 : const RejectNull<P1>&)
-OnInit(val : P) : void {static}
-OnDereference(val : P) : void
-Swap(_n1 : RejectNullStrict&) : void

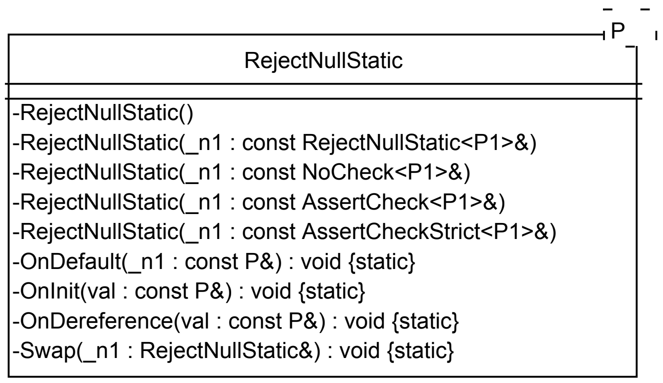

CheckingPolicy<typename StoragePolicy<T>::StoredType>
T,
OwnershipPolicy : template <class> class,
ConversionPolicy : class,
CheckingPolicy : template <class> class,
StoragePolicy : template <class> class
ref
+ref()
+ref(p : const StoredType&)
+ref(rhs : CopyArg&)
+ref(rhs : const ref<T1,OP1,CP1,KP1,SP1>&)
+ref(rhs : ref<T1,OP1,CP1,KP1,SP1>&)
+ref(rhs : ByRef<ref>)
+operator ByRef<ref>() : ref&
+operator=(rhs : CopyArg&) : ref&
+operator=(rhs : const ref<T1,OP1,CP1,KP1,SP1>&) : ref&
+operator=(rhs : ref<T1,OP1,CP1,KP1,SP1>&) : ref&
+Swap(rhs : ref&) : void
+~ref()
+GetImplRef(_n1 : sp) {abstract}
StoragePolicy<T>
ConversionPolicy
OwnershipPolicy<typename StoragePolicy<T>::PointerType>

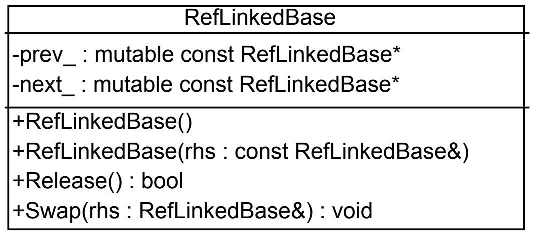

T,
OP : template <class> class,
CP : class,
KP : template <class> class,
SP : template <class> class
less
-operator()(lhs : const base::ref<T,OP,CP,KP,SP>&, rhs : const base::ref<T,OP,CP,KP,SP>&) : bool {frozen}
binary_function<base::ref<T,OP,CP,KP,SP>,base::ref<T,OP,CP,KP,SP>,bool>

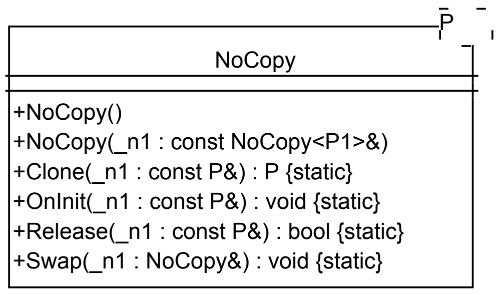

P
NoCheck
-NoCheck()
-NoCheck(_n1 : const NoCheck<P1>&)
-OnDefault(_n1 : const P&) : void {static}
-OnInit(_n1 : const P&) : void {static}
-OnDereference(_n1 : const P&) : void {static}
-Swap(_n1 : NoCheck&) : void {static}
P
AssertCheck
-AssertCheck()
-AssertCheck(_n1 : const AssertCheck<P1>&)
-AssertCheck(_n1 : const NoCheck<P1>&)
-OnDefault(_n1 : const P&) : void {static}
-OnInit(_n1 : const P&) : void {static}
-OnDereference(val : P) : void {static}
-Swap(_n1 : AssertCheck&) : void {static}
P
NonSmartShared
+NonSmartShared()
+NonSmartShared(_n1 : const NonSmartShared<U>&)
+Clone(val : const P&) : P {static}
+OnInit(val : const P&) : void {static}
+Release(val : const P&) : bool {static}
+Swap(_n1 : NonSmartShared&) : void {static}
P
RejectNull
-RejectNull()
-RejectNull(_n1 : const RejectNull<P1>&)
-OnInit(val : P) : void {static}
-OnDefault(val : P) : void {static}
-OnDereference(val : P) : void
-Swap(_n1 : RejectNull&) : void


T
ByRef
-value_ : T&
+ByRef(v : T&)
+operator T&() : ByRef&


Private::RefLinkedBase
P
RefLinked
+RefLinked()
+RefLinked(rhs : const RefLinked<P1>&)
+Clone(val : const P&) : P {static}
+OnInit(_n1 : const P&) : void {static}
+Release(_n1 : const P&) : bool


P
RejectNullStatic
-RejectNullStatic()
-RejectNullStatic(_n1 : const RejectNullStatic<P1>&)
-RejectNullStatic(_n1 : const NoCheck<P1>&)
-RejectNullStatic(_n1 : const AssertCheck<P1>&)
-RejectNullStatic(_n1 : const AssertCheckStrict<P1>&)
-OnDefault(_n1 : const P&) : void {static}
-OnInit(val : const P&) : void {static}
-OnDereference(val : const P&) : void {static}
-Swap(_n1 : RejectNullStatic&) : void {static}


RefLinkedBase
-prev_ : mutable const RefLinkedBase*
-next_ : mutable const RefLinkedBase*
+RefLinkedBase()
+RefLinkedBase(rhs : const RefLinkedBase&)
+Release() : bool
+Swap(rhs : RefLinkedBase&) : void


P
NoCopy
+NoCopy()
+NoCopy(_n1 : const NoCopy<P1>&)
+Clone(_n1 : const P&) : P {static}
+OnInit(_n1 : const P&) : void {static}
+Release(_n1 : const P&) : bool {static}
+Swap(_n1 : NoCopy&) : void {static}

INTERNAL DISTRIBUTION

1–5.	W. E. Dixon	25.	C. W. Nestor
6–10.	K. N. Fischer	26.	M. W. Noakes
11.	E. C. Fox	27–31.	F. G. Pin
12.	R. G. Gilliland	32.	R. W. Reid
13.	D. C. Haley	33.	R. B. Shelton
14.	W. F. Harris	34.	B. A. Worley
15.	D. M. Hetrick	35.	T. Zacharia
16–20.	D. L. Jung	36.	Central Research Library
21.	C. M. Kendrick	37.	ORNL Laboratory Records-RC
22.	A. L. King	38–39.	ORNL Laboratory Records-OSTI
23.	K. L. Kruse		
24.	L. J. Love		

EXTERNAL DISTRIBUTION

40–41. Arel Cordero, 3090 Potter St., Eugene, OR 97405

42. Teresa Fryberger, Director, Environmental Remediation Sciences Division, SC-75/Germantown Building, U.S. Department of Energy, 1000 Independence Ave., S.W., Washington, DC 20585-1290

43. Mark Gilbertson, Director, Office of Basic and Applied Research, EM-52/Forrestal Building, U.S. Department of Energy, 1000 Independence Ave., S.W., Washington, DC 20585

44. Roland Hirsch, SC-73/Germantown Building, U.S. Department of Energy, 1000 Independence Ave., S.W., Washington, DC 20585-1290

45. Ann Marie Phillips, Idaho National Engineering and Environmental Laboratory, Bechtel BWXT Idaho, LLC, P.O. Box 1625-3765, 2525 Fremont Ave., Idaho Falls, ID 83415-3765

46. Sylvia Wolfe, Oak Ridge Site Office Program Manager, U.S. Department of Energy, P.O. Box 2008, MS-6269, Oak Ridge, TN 37831-6269