CHAPTER VIII

DynaMechs: An Object Oriented Robot Simulation Package

8.1 Introduction

In the previous chapter, an efficient dynamic and hydrodynamic simulation algorithm was developed for general multiple chain robotic systems. In this chapter, the implementation of this algorithm is presented. Many simulation algorithms have been implemented using third generation languages like C and Pascal. The advantage of these languages over second generation languages is their support of structured programming techniques which emphasizes procedural decomposition of the programming task. This reduces the complexity of the resulting task because it allows for the independent development of sections (modules) of a large program. As the size of desired software continues to grow however this method breaks down because maintenance costs become very high when too many internal details and interactions between the modules must be tracked. Even feature-rich applications (like a simulator for a general class of robotic mechanisms) that are moderately large can also suffer from unnecessarily large maintenance costs.

In response to the complexity management problem, a concept called *information hiding* or *encapsulation* was invented which has spurred the development of a fourth generation of programming languages. These include object oriented programming languages (OOPLs) which include C++, Smalltalk, and Common Lisp Object System (CLOS), and object-based languages, like Ada [61]. Using OOPLs, the problem domain is decomposed in a top-down fashion into a hierarchy of component objects, also called a *part-of* decomposition. The structure of these objects are defined by class types that contain hidden data and a public interface in the form of methods (functions), whose implementation is also hidden, which act on this data. As a result, each object has a certain functionality, and now the emphasis in the top-down design focuses on the interaction of objects using only the method interfaces specified to achieve the programming goal. This significantly reduces the complexity of this task. It also removes the detailed implementation of the objects'

internal data and methods from the top level design decisions, and decouples their implementation from the rest of the programming task.

The greatest obstacle to overcome in realizing the benefits of object oriented programming is the need to change ones thinking about programs in terms of data and subroutines to the OO entities called objects [62]. With the proper object decomposition, a second bottom-up design process concentrates on the hidden implementation for each class of objects. This is accomplished with a *kind-of* classification of these objects which leads naturally to a second hierarchy called the class hierarchy. With their interface defined, the implementation of these classes are not dependent on (decoupled from) one another, the amount of information needed by the programmer at any one time to solve programming subtasks is reduced, and the complexity is significantly reduced. This method of programming also provides strong support for *code reuse*. As a result productivity is increased and maintenance of the code is much simpler [62].

For further discussion of OOD philosophy or methodologies, the reader is referred to the numerous texts and articles written on this topic (e.g., [61]). With this philosophy in mind, however, the purpose of this chapter is to present the resulting OO implementation of a dynamic and hydrodynamic simulation software package. The C++ language has been chosen because of its current popularity, availability of compilers, and its close ties with the third generation programming language, C. It is named **DynaMechs**[*], and is applicable to general multiple chain systems. The next two sections present the object and class hierarchies that have been developed to support simulation of a general class of robotic systems. At the same time a number of the OOD concepts that were used in this development are described. Then in Section 8.4, use of the resulting software package is presented. It describes initialization of the package, the implementation of the Articulated Body (AB) algorithm, and numerical integration. Also discussed is the implementation on a Silicon Graphics (SGI) workstation including runtime performance (since efficient implementation is important), and examples of graphical interfaces.

8.2 Object Hierarchy

Simply stated, an object is an entity that has a state (variables), behavior (methods or procedures) and identity (its name). The first step in the OOD process is to decompose the programming problem into such objects. These objects may or may not be decomposed further into more detailed objects. This procedure is repeated until the level of detail desired is reached. The resulting objects are organized into a hierarchy which represents the *part-of* decomposition. Figure 26 illustrates the

[*]Pronounced dynamics, and stands for dynamics of mechanisms.

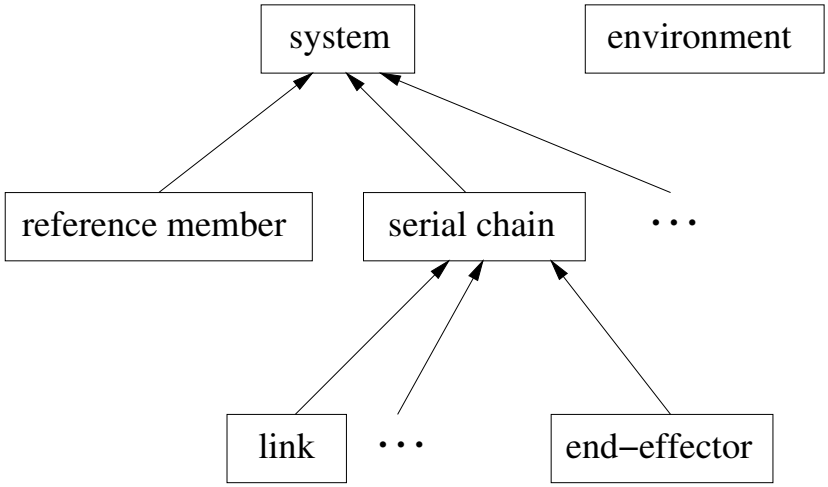

Figure 26: Object hierarchy for **DynaMechs**.

object hierarchy that has been developed for the simulation of a general class of robotic systems.

A robot-centric view is taken in the first step of the decomposition where the domain is divided into two parts, the robotic system and the surrounding environment, as indicated by the two top-level objects. The former contains all procedures and data related to the simulation of the robotic system and the latter contains the information and methods related to the external environment with which the system interacts such as gravity, fluid characteristics, and terrain models (not shown). The emphasis in this chapter is on the system object which must be capable of representing robots with varying topologies from single chain, fixed base manipulators to multiple chain, mobile base legged vehicles. Therefore, the system is a composite object and is decomposed further. All of these systems have a base (fixed or mobile) called the reference member and a number (an array in this implementation) of serial chains attached to it. The serial chain objects are further decomposed into a series (array) of link objects and an end effector object.

In terms of functionality, the system object provides the user with the interface to all of the needed methods such as initialization, computation of dynamics, integration, and rendering. The reference member and serial chain objects provide the system with the necessary methods to carry out their part of these tasks. In turn, the link and end effector objects provide the functions required by the serial chain objects. For the dynamics computation, the system controls the communication of data between the reference member and the serial chain objects. The reference member object contains the data and functions necessary to compute the quantities

associated with the reference member. The serial chain sets up the three recursions along the chain in the AB algorithm and passes data between the link and end effector objects. The link objects are responsible for computing the quantities associated with themselves (the equations in one iteration of the innermost loop of the dynamics algorithm), and finally, the end-effector object must detect contacts of the last link in the chain with the environment and compute contact forces, if present.

8.3 Class Hierarchy

In the previous discussion, the type of reference member (fixed or mobile) and link with an associated joint (revolute, prismatic, etc.) have not been specified. This is accomplished with the development of the class hierarchy which is the second step in the OOD process. Booch [61] states that "a class is a set of objects that share a common structure and a common behavior." Another way of describing the class-object relationship is that classes define the "types" of the objects that appear at the nodes of the object hierarchy. In this section, the class hierarchy developed for **DynaMechs** is presented. In addition, important OOD concepts that were used in this development including encapsulation, inheritance, and polymorphism are presented. To illustrate these concepts, examples are taken from the design of **DynaMechs**.

8.3.1 The C++ Class

As Booch's definition suggests, a class has two distinct features. It defines the *member variables* that store data or state, and *member functions* which correspond to a set of behaviors that operate on the member variables. As an example, Figure 27 shows the definition of a C++ class, `RigidBody`, that is used in **DynaMechs**. Examples of member variables for a rigid body include all of its mass and inertia properties, and hydrodynamic coefficients like the drag coefficients and the added mass matrix. Also note that a number of new types have been defined (using `typedef`) to make the declarations of dynamics variables clearer. For instance a `CartesianVector` is a one-dimensional, three element array of `float`s, `CartesianTensor`s are 3×3 arrays, `SpatialVector`s are six-vectors, and `SpatialTensor`s are 6×6 arrays.

As illustrated in this figure, C++ also allows the designer to control access to the member variables and functions with the use of the `private`, `protected`, and `public` keywords. The `private` and `protected` sections contain the variables and functions that cannot be accessed/invoked from outside of the class. The difference between them is made clear in the discussion on inheritance below. The `public`

```
class RigidBody
{
protected:
    float           mass;
    CartesianVector cgPos;         // position of center of gravity.
    CartesianVector h;             // first mass moment (mass*cgPos).
    CartesianTensor Ibar;          // 3x3 inertia tensor about body axes.
    SpatialTensor   SpatialInertia;

    int             axis;          // drag parameters.
    float           Cd, length, radius;
    float           dispFluidVol;  // volume of fluid displaced by body.
    float           dispFluidMass; // mass of fluid displaced by body
    CartesianVector cbPos;         // position of center of buoyancy.
    SpatialTensor   AddedMass;     // 6x6 added mass matrix
    SpatialTensor   HydrodynamicInertia;

public:
    RigidBody(FILE *cfg_ptr);
    ~RigidBody();
    void Compute_Drag(SpatialVector vrel, SpatialVector fd);
};
```

Figure 27: `RigidBody` class.

section contains the user accessible variables and functions. Normally all variables and some functions are private or protected, and it is through calls to the public functions that the variables are manipulated.

These mechanisms reinforce the concept of *information hiding* or *encapsulation* which is the key to the benefits of complexity management with OOD. As a result, the emphasis is moved from the implementation of the variables or functions, which can be deferred until later, to the interface that is provided by the class. Therefore, the user only has to worry about the interface in order to use the class, and the implementation within the class is the class designer's responsibility. This emphasizes the need for early and complete interface specifications in the design process. In fact, as long as the interface specifications do not change the underlying implementation can undergo arbitrary modification without affecting other portions of the software system that use this class.

In the `RigidBody` example, three functions are provided in the public section. The first two are special functions required by C++ and are called the constructor (which is given the same name as the class) and the destructor which is preceded by a tilde. The constructor is called when the class is instantiated for an object and performs the appropriate functions to initialize the member variables. In this example, the constructor expects a pointer to a file that contains the data with which to initialize the variables. For example, an object called `anvil` would be instantiated as follows:

```
RigidBody anvil(cfg_ptr);
```

where `cfg_ptr` points to a file that has been opened and contains the object's parameters. The destructor is automatically called when the object goes out of scope or the program exits and it performs specific shutdown procedures that may be required by the class.

The third function in this example, `Compute_Drag`, is a normal member function that is provided by the designer for the users of the class. For this function, the user supplies the object's relative velocity, and the function accesses the class's private variables to perform the computation of the drag force. This function is accessed in a manner similar to accessing an element of a structure as follows:

```
anvil.Compute_Drag(vrel, fD);
```

Should the implementation of the drag force computation change at a later date, users would not be required to change their portion of the code provided the interface (the parameters and name of the function) does not change.

8.3.2 Inheritance

Another goal in the design of the classes is to increase code reuse through a mechanism called *inheritance*. The desire in this case is to move variables and methods that are shared by more than one class into another, more general, class which is then inherited by the classes requiring them. The new class is called a *superclass* and its structure and behavior are inherited by its *subclasses*. This terminology indicates a hierarchical relationship between classes and is the motivation behind the development of the class hierarchy.

Since links and mobile bases are both examples of rigid bodies in robotic systems, the `RigidBody` class is an example of a superclass that must be inherited by both `Link` and `MobileBase` subclasses. An example inheritance in the definition of the `Link` class is shown in Figure 28. The first line, contains the command to inherit the `RigidBody` class definition. The `public` keyword in this line indicates how access

```
class Link: public RigidBody
{
protected:
    RotationMatrix  _R;      // rotation matrix from inboard link
    CartesianVector _p;      // position wrt inboard link.

public:
    Link(FILE *cfg_ptr);

    // Link-to-link spatial transformation functions:
    void   tx_To_Inboard(SpatialVector curr, SpatialVector prev);
    void tx_From_Inboard(SpatialVector prev, SpatialVector curr);
};
```

Figure 28: `Link` class.

to inherited variables and methods are determined. With public inheritance, public attributes in the superclass will remain public in the subclass, protected attributes in the superclass become similar to private attributes of the subclass, and private variables in the superclass will be inaccessible to the subclass.

In this implementation, access to superclass variables is desired within the subclasses; therefore, the `protected` keyword is used most often. When a `Link` object is instantiated, the constructor for the `RigidBody` class is called as well as the one for the `Link` class. The resulting object contains all of the variables and methods from both classes. In other words, it will have the variables associated with a rigid body (mass, inertia, etc.) as well as a link in a serial chain (position and orientation with respect to its inboard neighbor). Methods from both classes will also be available so that the drag force on the link can be computed as well as the transformation of spatial vectors to and from the coordinate system of the inboard neighbor.

Inheritance from more than one superclass is also possible. This is desired with the `MobileBase` class. Like a link, a mobile base has the characteristics of a rigid body so it should inherit the definition of the `RigidBody` class, but it also has the added characteristics of a reference member. Because fixed and mobile base objects share common attributes a `RefMember` superclass is defined to hold these which is then inherited by the corresponding classes. An example of a `RefMember` superclass is shown in the top part of Figure 29. In contains the position and orientation relative to an inertial coordinate system (ICS), and it also provides the user with a pair of functions to transform vectors between the ICS and the reference member's

```
class RefMember
{
protected:
    CartesianVector pos;   // position wrt inertial coord. sys. (ICS)
    EulerAngles     pose;  // [phi theta psi]' Euler angles

public:
    RefMember(FILE *cfg_ptr);

    void Tx_From_ICS(CartesianVector pICS, CartesianVector pRef);
    void   Tx_To_ICS(CartesianVector pRef, CartesianVector pICS);
};


class MobileBase: public RigidBody, public RefMember
{
private:
    SpatialVector vel;      // velocity and accel wrt ICS.
    SpatialVector acc;

public:
    MobileBase(FILE *cfg_ptr);

    void Update_State();
};
```

Figure 29: `RefMember` and `MobileBase` classes.

coordinate system.

In contrast to a fixed base, a mobile base is also capable of motion with respect to the ICS. Therefore, the corresponding `MobileBase` class includes variables for its spatial velocity and acceleration with respect to the ICS as shown in the bottom part of Figure 29. In addition, it could provide a member function, like `Update_State`, which is responsible for numerically integrating the velocity and accelerations to obtain new values for the position, orientation, and velocity of the reference member.

Note that the first line of the `MobileBase` class definition specifies more than one class to be inherited, which is called, quite appropriately, *multiple inheritance*. Therefore, objects that are instances of the `MobileBase` class will also contain the

member variables and functions of both the `RigidBody` and `RefMember` classes. Multiple inheritance can also be achieved by inheriting classes that are themselves subclasses. Examples of this are given with the further refinement of the `Link` class in the next subsection.

These examples show how inheritance can be used to achieve some level of code reuse. The key to successful decomposition is to move variables and functions that are common to more than one class to a superclass which is then inherited by these subclasses. The result is that attributes are moved to the most general level possible. The advantage is that the amount of code reuse by derived classes is maximized because these variables and functions need only to be implemented once. This reduces the amount of coding and testing that is required of the programmer and therefore efficiency in code development is increased.

8.3.3 Virtual Functions and Abstract Classes

Polymorphism is another important concept to be considered during the development of the class hierarchy. This is defined as the ability to refer to methods of different classes by a common name, and through this name objects of various classes can "respond to a common set of operations in different ways" [61]. One obvious example in the simulation of robotic mechanisms where this is desirable is in the implementation of dynamics computation at the link level. Different joint types (e.g., revolute and prismatic) require different efficient implementations of the equations. With a third generation language this would require conditional or case statements at the serial chain level to execute the proper code depending on the type of joint involved. This code would also have to be rewritten each time a new joint type is to be modeled. However, with polymorphism this becomes unnecessary because the dependence on the type of joint is eliminated from the serial chain code.

In this section, further refinement of the `Link` class, to pair a link with either a revolute or a prismatic joint, is used to illustrate how this is accomplished. A function to perform the efficient congruence transformation of the reflected Articulated Body (AB) inertia matrices, $\mathbf{N}_i$, is given as an example since it is shown, in Appendix A, to depend on the type of joint. The first step is to expand the class hierarchy related to the `Link` class which will serve as the superclass for all of the link subclasses which are *specialized* for different joint types. The resulting classes are illustrated in Figure 30. Note that an intermediate class, `MDHLink`, is also defined which contains Modified Denavit-Hartenberg (MDH) parameters and related quantities that are common to both revolute and prismatic joints. The reason that these terms are not included in the `Link` class is because multiple degree of freedom joints cannot be modeled using MDH parameters, but to support polymorphism they will also have to inherit the `Link` class (see `BallnSocketLink` in the next section).

```
class Link: public RigidBody
{
public:
    Link(FILE *cfg_ptr);

    virtual void CongTx_To_Inboard_Refl(SpatialTensor N,
                                        SpatialTensor Nib) = 0;
};

class MDHLink: public Link
{
protected:
    float aMDH, alphaMDH, dMDH, thetaMDH;    // Modified DH parameters

public:
    MDHLink(FILE *cfg_ptr);

    virtual void CongTx_To_Inboard_Refl(SpatialTensor N,
                                        SpatialTensor Nib) = 0;
};

class RevoluteLink: public MDHLink
{
public:
    RevoluteLink(FILE *cfg_ptr);

    void CongTx_To_Inboard_Refl(SpatialTensor N, SpatialTensor Nib);
};

class PrismaticLink: public MDHLink
{
public:
    PrismaticLink(FILE *cfg_ptr);

    void CongTx_To_Inboard_Refl(SpatialTensor N, SpatialTensor Nib);
};
```

Figure 30: Polymorphism in the link classes.

```
1    Link *link[2];
2    SpatialTensor N[3];
3
4    link[0] = new RevoluteLink(cfg_ptr);
5    link[1] = new PrismaticLink(cfg_ptr);

     ⋮

6    for (i=2; i>0; i--)
7    {
8        link[i-1]->Congtx_To_Inboard_Refl(N[i], N[i-1]);
9    }

     ⋮
```

Figure 31: An example using polymorphism.

To achieve polymorphism, a uniform interface must be provided in the `Link` superclass for each function that will be specialized according to the joint type in its subclasses. In C++, this is accomplished with the `virtual` keyword. This is used in both the `Link` and `MDHLink` classes for the `Congtx_To_Inboard_Refl()` function. The "`= 0`" at the end of these declarations indicates that these are *pure virtual functions*. This indicates that the definition of this function is deferred until one of the subclasses defines it. Because it is undefined, these classes cannot be used to directly instantiate an object (that is, no objects exist that are just `Link` or `MDHLink` objects). This type of class is called an *abstract* class and can only be used in inheritance in base classes. In contrast, classes that can be used to instantiate objects are called *concrete* classes such as the `RevoluteLink` and `PrismaticLink` classes.

With that in mind, Figure 31 lists some C++ code that could be used in the `SerialChain` class and illustrates how the virtual functions are used to achieve polymorphism. In this example, an array of two pointers to `Link` objects are used (line 1). Since these are pointers and not `Link` objects themselves nothing is actually instantiated so that the rule against instantiation of an abstract class is not violated. This provides the common name through which the function is called. Lines 4 and 5 are then used to instantiate a `RevoluteLink` and `PrismaticLink` object. The `new` command allocates memory for the specified class, calls the constructor with the specified parameter list, and returns the address of the object. Although the

returned values are pointers to `RevoluteLink` and `PrismaticLink` classes and are assigned to a `Link` pointer, the compiler does not complain about a type mismatch because `Link` is a superclass of the others.

With the common name, called `Link`, and the uniform function interface, called `Congtx_To_Inboard_Refl`, polymorphism can now be used. This is illustrated in line 8 of the example where the congruence transformation of `N[2]` is performed across both links in a backward recursion. Regardless of the actual link type, the function call is the same. Internally, however, `PrismaticLink`'s function was executed for `link[1]` and `RevoluteLink`'s function for `link[0]`. This is also called *dynamic binding* where the particular function to be executed is not known until runtime (lines 4 and 5). With polymorphism no case statements or conditionals are needed in this code to determine which function to call. Other than the instantiation of the link objects themselves, this portion of the code does not require knowledge about the various types of links that may be defined. This is also advantageous if different link subclasses are added to the software system at a later date because this section of the code would require no modification.

8.3.4 The Hierarchy

The structure implied by the inheritance processes described in the previous three subsections leads to a hierarchy in which the superclasses are placed above the subclasses. In the resulting tree, the more "general" classes appear at the top where characteristics are common to more of the subclasses. As the tree is traversed downward, the classes define more specialized objects. This hierarchy is often referred to as the *kind-of* hierarchy[†]. The hierarchy that has been developed for the **DynaMechs** software is shown in Figure 32. Many of the classes have been used in the examples in the previous discussions, and the abstract classes are indicated by dashed boxes.

Reexamining the object hierarchy in Figure 26 in light of this class hierarchy, a number of comments can now be made about the software implementation. The `System` class is a *simple* class (that is, no inheritance and no virtual functions). Its member variables include a pointer to a `RefMember` object and an array of pointers to `SerialChain` objects. The array of pointers to the `SerialChain` objects is dynamically allocated in the constructor when a `System` object is instantiated. This allows for any number of serial chains attached to the reference member to be simulated.

The `RefMember` class contains member variables for the position and orientation of the reference member with respect to the inertial coordinate system and provides

[†]This comes from the statement, "a *child class* is a kind of *parent class*," which should hold true for each parent-child pair in the tree. For example, a `Link` is a kind of `RigidBody`

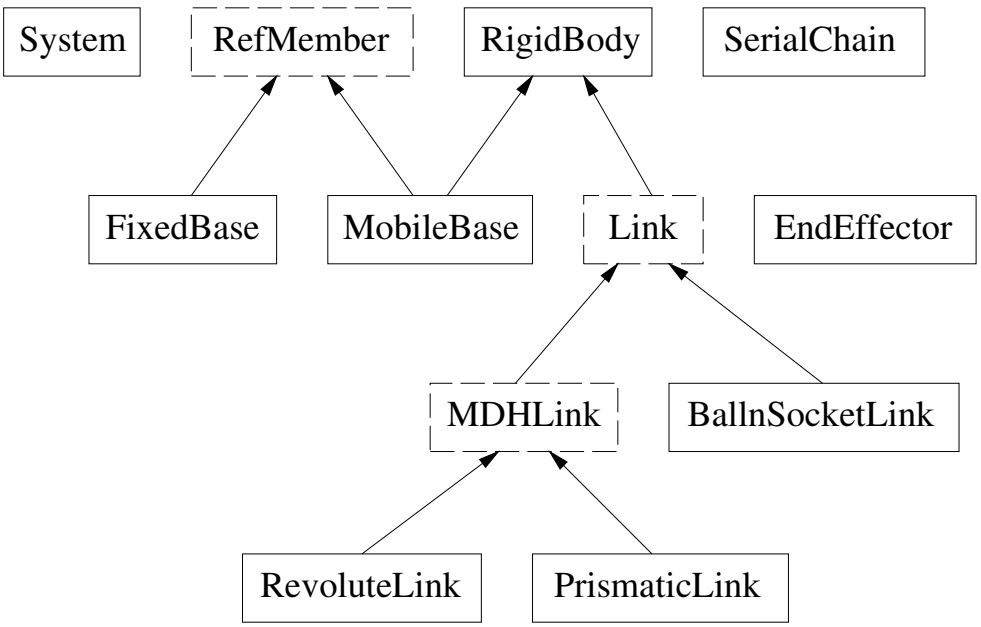

Figure 32: Class hierarchy for **DynaMechs**.

functions for transformations between this and the reference member's coordinate system. It is an abstract class because it also contains a number of pure virtual functions (not shown in Figure 29) to provide `System` objects with a uniform interface for functions whose implementation depends on whether the reference member is a `FixedBase` or `MobileBase` object. This allows for dynamic binding of the proper code at runtime so that either fixed or mobile base systems can be simulated. One example would be the function to compute the acceleration of the reference member. For the `MobileBase` function the dynamics developed in the previous chapter would be implemented, while the `FixedBase` function would simply return zero acceleration without any computation.

The `SerialChain` class is another simple class. Its member variables include an array of pointers to `Link` objects and a pointer to an `EndEffector` object. The array is also dynamically allocated in the constructor when a `SerialChain` object is instantiated, so that a serial chain with an arbitrary number of links can be simulated by this software. The `EndEffector` class is also a simple class and currently contains code for modeling compliant contacts at the end of each serial chain and computing external tip forces as described in [7]. A pointer is provided in the `SerialChain` class to support future specialization under the `EndEffector` class to support the specific functions of grippers, foot pads, sensors, and even cameras.

The structure of the `Link` class and its subclasses were discussed in some detail above. The only addition to this part of the hierarchy is the `BallnSocketLink` class which is used to efficiently model the kinematics and dynamics of the footpads on the end of each leg of Aquarobot which contain unpowered ball-in-socket joints. This class is an example of how to include new types of links. Its kinematics cannot be modeled using MDH parameters, but rather a position vector and a set of Euler angles is most efficient. The joint axis vector, ϕ_i, is 6×3, hence the joint position, velocity, acceleration, and torque are now represented by 3-vectors (not scalars as in the case of `MDHLink`s). These differences are all hidden in the implementation of the `BallnSocketLink` class. The primary responsibility of this class definition is to provide the definitions of the virtual functions exactly as they appear in the `Link` class.

In summary, the programming problem was decomposed into smaller tasks with the development of the object hierarchy. The class hierarchy developed in this section can be considered orthogonal to the object decomposition, and its development further decomposes the programming tasks. Through techniques such as inheritance, encapsulation, and polymorphism, the overall complexity of the problem is significantly reduced because interactions between code segments are limited and tightly controlled. This also significantly reduces the amount of time taken to maintain and modify such code.

8.4 Using DynaMechs

Now that the object and class hierarchies have been presented, the use of this software package is discussed in this section. The steps for performing the dynamic simulation of a robotic system using this package are initialization of the composite `System` object, execution of the dynamics algorithm, numerical integration, and, if desired, graphical output. Each of these steps is discussed in this section. An example of the graphical output obtained on a Silicon Graphics (SGI) workstation is presented along with performance results on this platform.

8.4.1 Initialization

Before the dynamics algorithm can be implemented, an instance of the `System` class must be created. Just like a variable, it can be given any name, and for this discussion it is called `robot`. When instantiated, the constructor for the `System` class requires, as arguments, a pointer to the file containing all of the necessary kinematic, dynamic and hydrodynamic parameters for the entire robotic system, and a set of variables which will contain the state, accelerations, and inputs of the system. This is accomplished with the code shown in Figure 33.

```
// state variables.
     EulerAngles     refPose;
     CartesianVector refPos;
     SpatialVector   refVel;
     float ***jointPos, ***jointVel;

// accelerations.
     SpatialVector refAcc;
     float ***jointAcc;

// force/torque inputs
     SpatialVector refForce;
     float ***jointForce;

// a robotic system.
     System robot(cfg_ptr, refPose, refPos, refVel,
                  &jointPos, &jointVel, &jointAcc, &jointForce);
```

Figure 33: Initialization of robotic system and input/output variables.

The rest of the initialization process is hidden from the user. The `System` constructor extracts the information it needs from the file and then uses it to initialize its member variables. Two pieces of data specify the type of reference member (fixed or mobile) and the number of serial chains the system has. These are then used to direct the `System`'s initialization of its component objects, an instance of the `RefMember` class and an array of `SerialChain` instances. This is accomplished by passing the file pointer when the constructor for these objects are called. During the initialization of the `SerialChain` objects the file pointer is also passed on to the individual `Link` constructors and each `EndEffector` constructor. In this manner, the data for the entire robotic system can be contained in a single input data file.

At the same time, the state variables that are passed to the `System` constructor are then passed along to the appropriate constructors of component objects which will set these to the initial conditions specified in the input file. Since the number of serial chains and the number of links in each chain are dynamically allocated at runtime, the corresponding state, acceleration, and input variables (`jointPos, jointVel, jointAcc, jointForce`) are also dynamically allocated during this initialization. The resulting variables appear to the user as three dimensional arrays. For instance, the torque for the fourth joint of the second serial chain of the system

would be set as follows:

```
jointForce[1][3][0] = torque;
```

where C++ arrays begin indexing with zero so the first link of the first chain would be referenced with two zero indices. The third index in the example supports multiple degree of freedom joints. For revolute and prismatic joints this is always zero, and for ball-in-socket joints it ranges from 0 to 2. Once they are allocated, the `System` object receives input from and sends output to the user through these variables.

8.4.2 AB Dynamics Algorithm

The next step for the user is to compute the dynamics of the system using the AB algorithm developed in Chapter 7 (Tables 13–15). First, the user must set the inputs to the dynamics computation, `refForce` and `jointForce`. Then, these are passed as arguments to the `AB_Dynamics` member function of the `System` class as follows:

```
robot.AB_Dynamics(refForce, jointForce);
```

This function begins the computation the dynamics for the entire system and the accelerations that are computed will be stored in member variables of the respective objects that make up the system. Although the implementation of the algorithm is completely hidden from the user, this section presents an overview of the implementation of the AB dynamics algorithm, and to aid in this discussion, a graphical representation of this implementation is given in Figures 34 and 35.

In these figures, each box corresponds to one of the boxes of the object hierarchy. At the top of each, the class and the member function names are given. Pointing to each of these is the object name (which is a member variable of the encompassing box) which is used to reference them. The remainder of the contents of the box describes the computation that is performed in terms of the variable names that were used in Chapter 7. As before, the algorithm is divided into three parts: Forward Kinematics, Backward Dynamics, and Forward Accelerations, and the member functions are named accordingly.

In the Forward Kinematics step, the `robot` object begins by calling the reference member's (`ref`) `AB_Forward_Kinematics()` function to compute velocity, acceleration, and bias terms. When complete, it calls each serial chain's (`chain[k]`) `AB_Forward_Kinematics()` function. Each `chain[k]` object, in turn, computes velocity, acceleration, and $\boldsymbol{\zeta}_{k,0}$ terms, sets up the `for`-loop for the forward recursion along the chain, and calls each of its links' (`link[i]`) `AB_Forward_Kinematics()` functions. At the end of the recursion, the serial chain calls its end effector, `ee`, function to compute the contact force at the end of the chain, ${}^{N}\mathbf{f}_{k,N+1}$, if present.

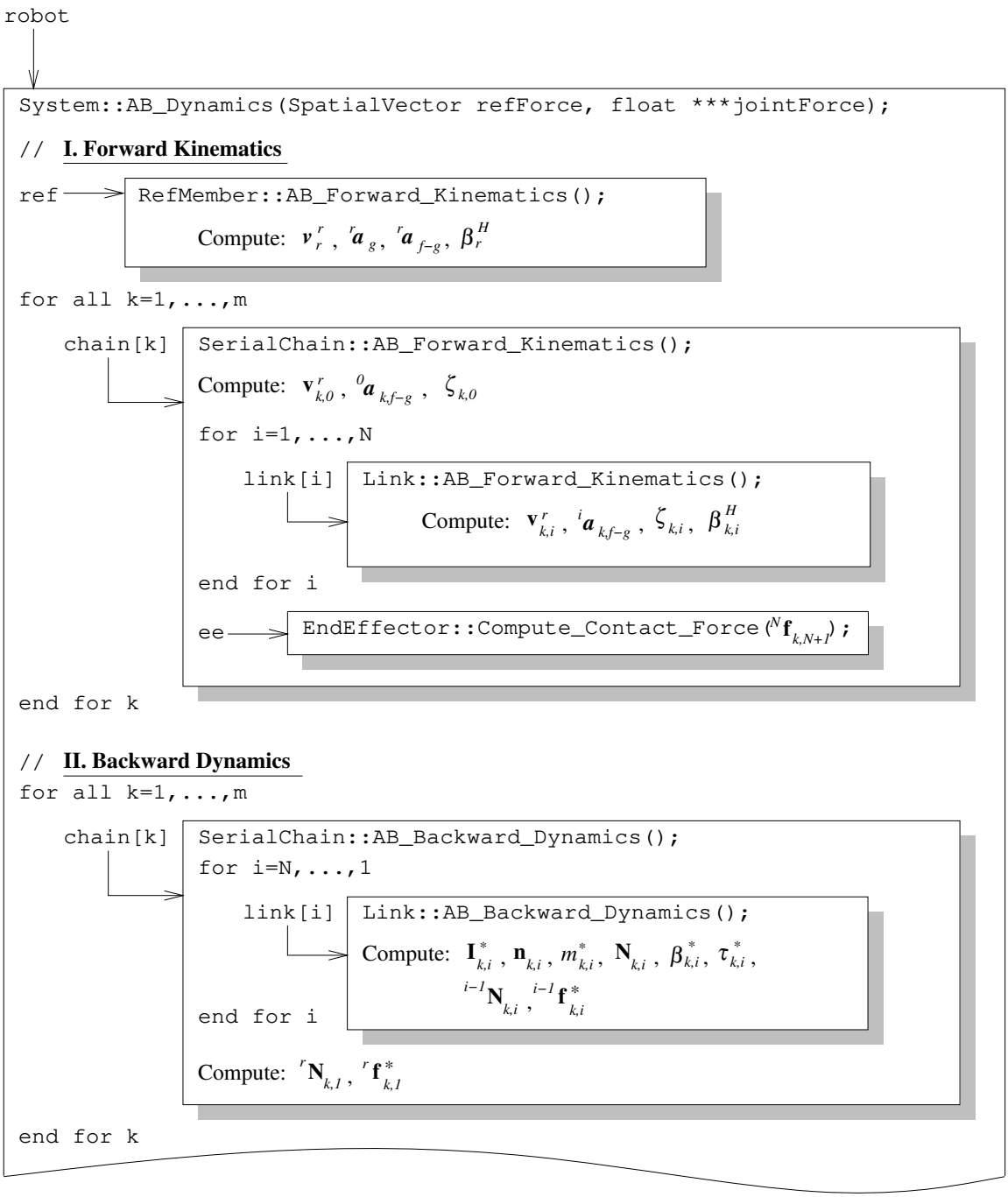

Figure 34: AB algorithm distribution among the **DynaMechs** classes.

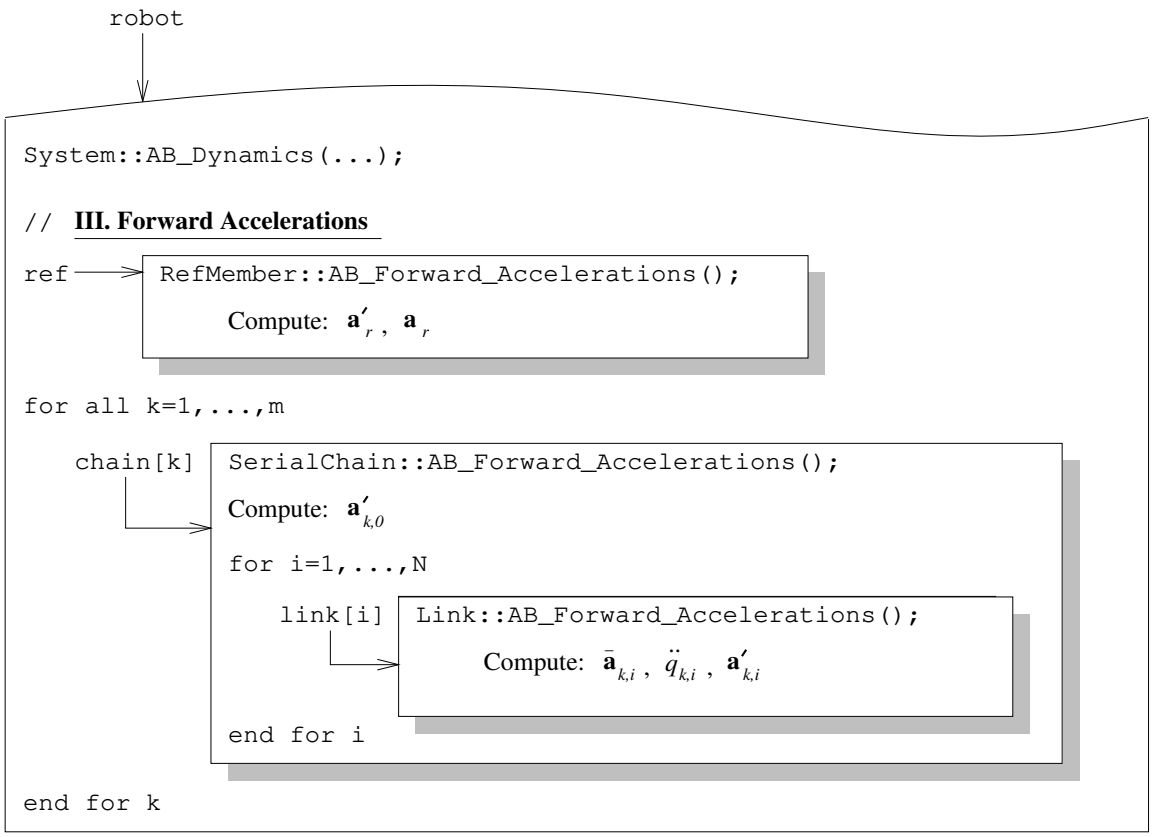

Figure 35: AB algorithm distribution among the **DynaMechs** classes (continued).

In the Backward Dynamics step, the `robot` object calls each of its chain's `AB_Backward_Dynamics()` functions. Each `chain[k]` object sets up the `for`-loop for the backward recursion, and calls each link's `AB_Backward_Dynamics()` function. At the end of this recursion, each `chain[k]` object computes ${}^r\mathbf{N}_{k,1}$ and ${}^r\mathbf{f}^*_{k,1}$ in preparation for the final step of the algorithm.

In the final Forward Accelerations step, the `robot` object begins by calling the reference member's `AB_Forward_Accelerations()` function to compute the biased reference member acceleration, $\mathbf{a}'_r$, that is needed by the serial chains, and the actual reference member acceleration, $\mathbf{a}_r$, that is needed for the simulation. Then, it calls each serial chain's (`chain[k]`), `AB_Forward_Accelerations()` function. Each `chain[k]` object, in turn, computes some initial quantities, sets up the `for`-loop for the forward recursion, and calls the `AB_Forward_Accelerations()` function of each of its links (`link[i]`). At the end of this recursion, the reference member and link

accelerations remain stored in the respective objects' member variables.

These figures also indicate how this algorithm can be parallelized because the iterations of the `for all k` loops can be performed simultaneously. By instantiating the `SerialChain` objects on different processors and providing for synchronization and communication between them and the `System` object, spatial parallelism can be implemented like that shown in Chapter 3 for multiple closed chain manipulators. The form the parallelism is also nearly the same except this algorithm requires two stages of serial computation for the reference member forward kinematics and acceleration.

8.4.3 Numerical Integration

The final step of simulation is to numerically integrate the accelerations to update the state of the system (the positions and velocities of the joints and the reference member). At present, only Euler integration is implemented to support real-time evaluation of the resulting system. To perform this step, the user executes another member function of the `System` class as follows:

```
robot.Update_State(idt, refPose, refPos, refVel, refAcc,
                        jointPos, jointVel, jointAcc);
```

The first parameter, `idt`, is the integration stepsize which is provided by the user. To achieve real-time operation, the system clock is monitored and the amount of time since the last call to `Update_State` is provided. The rest of the parameters are output variables which supply the user with the accelerations that were computed in the last dynamics computation, and the updated state after the numerical integration is performed. This function is implemented in a distributed manner much like the dynamics algorithm explained in the previous section. The `System` function, calls the `RefMember` function to update its state, and then calls each of the `SerialChain` functions. The `SerialChain` functions call each of its `Link` functions to update the joint state.

8.4.4 Silicon Graphics Implementation

Another member function of the `System` class which graphically renders the resulting system is still under development at the Naval Postgraduate School (NPS) [63]. It is invoked as follows:

```
robot.draw();
```

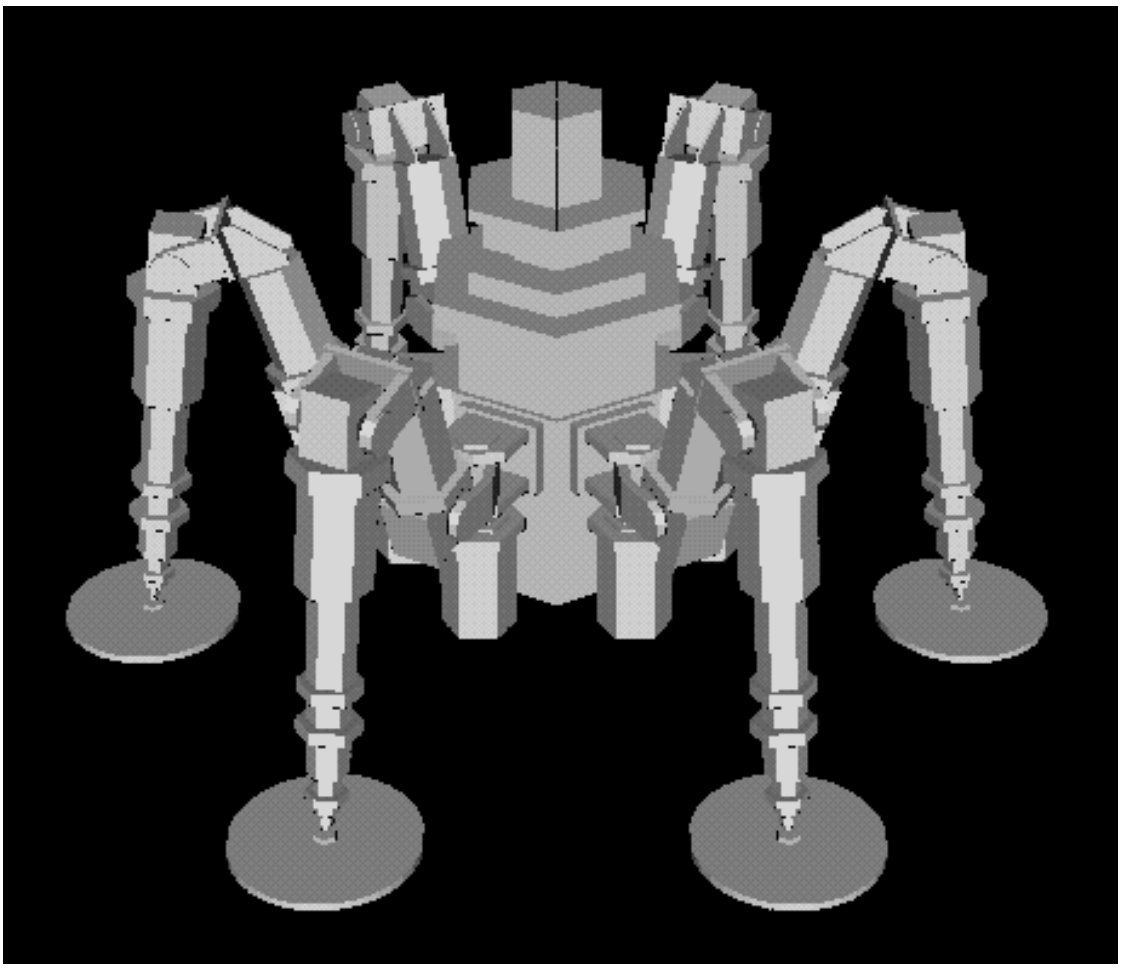

Figure 36: Scene from the *Aquarobot* simulation.

Rendering is distributed among the classes like the `Update_State()` function. The `System` function calls component objects' `draw()` functions, and the `SerialChain` function calls its component objects' `draw()` functions. At each step, the graphics state of the machine is modified using the state of the object to be rendered (reference member position and orientation, MDH parameters of the links, etc.). Thus the position and orientation of each object is defined as needed by the graphics system and it is then rendered. The specific command issued to render the object depends on the particular modeling package used. To date SGI's Inventor [64] and Performer [65] packages, as well as NPS's Graphics Description Language (GDL2) [66] have been implemented for this purpose. Figure 36 shows the *Aquarobot* which was rendered

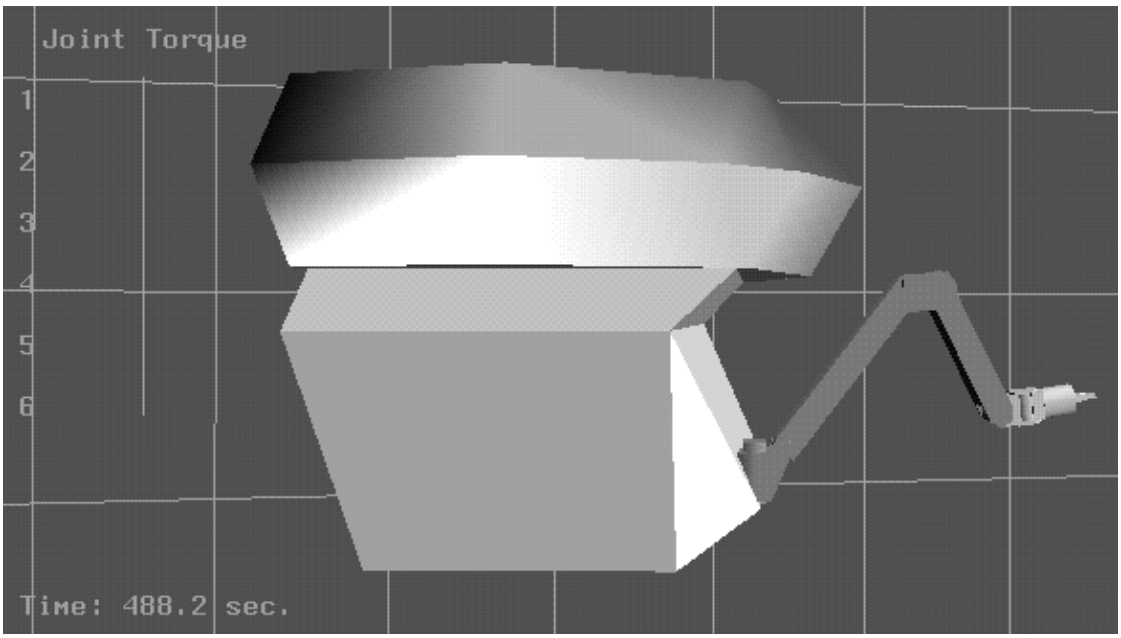

Figure 37: Scene from the *Tiburon* simulation.

on an Indigo^2 workstation with Extreme graphics using GDL2. Figure 37 shows *Tiburon* which was rendered using Inventor. It also contains a superimposed "heads up" display of the current joint torques as well as thruster output and the current control mode (not shown).

The runtime performance of the simulation algorithm has been measured on the SGI system with a 150MHz MIPS R4400 processor. Due to the coarse resolution of the timing functions, runtime averages were taken over a hundred iterations of the dynamics computation. The performance for both systems using single precision arithmetic functions are given in Table 20. The ROV system requires about 0.7 ms on the average to perform one iteration of the simulation which should be adequate for real-time performance during normal operation of the ROV without contact with the environment. One iteration of the *Aquarobot* simulation requires about 2.6 ms. In either case and depending on the stiffness of the compliant contacts with the environment, these may or may not be adequate for real-time performance, and this subject requires further study.

These performance results do not include rendering of the scenes themselves. The amount of time required to render is highly dependent on a number of factors: number and nature of the graphics elements (e.g., filled polygons) to be rendered, the graphics hardware available (e.g., Elan, Extreme, RealityEngine), and the software

Table 20: Runtime performance of simulations on SGI Indigo^2 workstation.

Robotic System	Stepsize Range (ms)	Stepsize Mean (ms)
Tiburon ($m = 1,\ N = 6$)	[0.68, 0.73]	0.7
Aquarobot ($m = 6,\ N = 4$)	[2.5, 2.8]	2.6

used to control the rendering process (GDL2, Inventor, Performer). The rendering process is faster with the more powerful graphics hardware. With an SGI RealityEngine, rendering is performed almost completely with the graphics hardware and does not affect the load on the CPU, and thus does not affect the performance of the simulation. As the capabilities of the graphics hardware decrease, significant CPU resources can be required which could significantly affect the performance of the resulting simulation. Multiprocessors and software packages like Performer can be used to distribute the graphics load, and Performer also has a feature to "cull" unseen graphics elements before rendering actually takes place. Both are being investigated at NPS to increase the performance of the resulting simulation system [63].

8.5 Summary and Conclusions

In this chapter, the implementation of the simulation algorithm was presented and the resulting software package, called **DynaMechs**[‡], has been described. A primary goal of this endeavor was to investigate object oriented design (OOD) techniques to implement the algorithm in C++ for a general class of robotic systems (star topologies) ranging from single chain systems with fixed bases to multiple chain systems with a mobile base. The first step was to decompose the system into its components and develop a general object hierarchy that is capable of describing the range of robotic systems just described. This forms the foundation for the next step in the OOD process which is the development of the class hierarchy which was used to define the types of objects that could be present in the system (e.g., mobile and fixed reference members, links paired with revolute and prismatic joints).

The two hierarchies provide the "road map" for decomposition and decoupling

[‡]This package is available via anonymous FTP from `taurus.cs.nps.navy.mil` in the `/pub/dynamechs/` directory.

of the programming task. This process leads to OOD's main advantage over programming languages from previous generations which is the increased ability to manage the complexity in very large programming tasks. Some of the techniques that were employed in this implementation were single and multiple inheritance, abstract classes to support polymorphism, and most importantly, encapsulation (information hiding) techniques. The result is a very efficient simulation algorithm that is capable of simulating different kinds of systems, containing many different kinds of links, and with and without hydrodynamics. With the later addition of the `BallnSocketLink` class, it has also proved its enhanced ability to easily extend the package. It has also proven this ability, by the ease with which various graphics packages have been included in the system for rendering the results.

The implementation of the algorithm is very efficient and yet it is still able to handle a wide variety of systems and link types. Runtime performance is very close to or exceeds real-time rates. Work is ongoing at NPS to achieve improved performance with different graphical packages and multiprocessors. For the latter, the parallelism discussed in Part I of this dissertation could be implemented on the multiprocessor SGI systems, and the speedups expected are comparable to those reported in Chapter 3. In addition, simplified dynamics models are also being investigated [67] to increase performance. The "full" dynamics implemented in this package will act as the baseline for measuring the viability of such approaches.