ArtiSynth Reference Manual

John Lloyd
July 8, 2014

Contents

Artisynth is a mechanical modeling system that allows users to combine finite element method (FEM) components with multibody systems, constraints, and collision handling. It is implemented in Java, and provides a graphical interface for interactive model editing and simulation control.

At present, most Artisynth models are created with Java code, using the Artisynth API. The programmatic aspects of Artisynth are the focus of this manual.

Component Hierarchy

Model Components

Artsynth models are created from a hierarchy of components. Each component is an instance of ModelComponent, which contains a number of methods used to maintain the component hierarchy. These include methods for naming and numbering components:

```
// get the name for this component
String getName();

// set the name for this component
void setName (String name);

// get the number of this component
int getNumber();

// set the number of this component (for internal use only)
void setNumber (int num);
```

Each component can be assigned a name, which can be any sequence of characters that does not begin with a digit, does not contain the characters '.', '/', ':', '*', or '?', and does not equal the string `"null"`. For components which are not assigned a name, getName() will return `null`.

Artisynth may also be configured so that components names must be unique for all components which are children of the same parent.

Even if a component does not have a name, it has a number, which identifies it with respect to its parent. Numbers are assigned automatically when a component is added to its parent, and persist unchanged until the component is removed from its parent. This persistence of numbers is important to ensure that components keep the same path name as long as they are connected to the hierarchy.

Names and/or numbers can be used to form a path name for each component that identifies its place in the hierarchy. If a component does not have a name, its number is used in the path instead. Some example path names look like:

```
models/jawmodel/axialSprings/lat

models/mech/models/tongue/bundles/2/elementDescs/12
```

`ModelComponent` contains a number of other methods for navigating and maintaining hierarchy structure:

```
// get the parent of this component
CompositeComponent getParent();

// sets the parent of this component (internal use only)
void setParent(CompositeComponent parent);

// called by the system after a component is added to a parent
void connectToHierarchy();

// called by the system before a component is removed from a parent
void disconnectFromHierarchy();

// get all hard references for this component (see below)
void getHardReferences (List<ModelComponent> refs);
```

```
// get all soft references for this component (see below)
void getSoftReferences (List<ModelComponent> refs);

// called to update the component when soft references are removed
void updateReferences (boolean undo, Deque<Object> undoInfo);

// notify the parent of a change in this component
void notifyParentOfChange (ComponentChangeEvent e);

// returns true if this component contains state information
boolean hasState();
```

getParent() returns the component's parent, which is a CompositeComponent (Section 1.3). Conversely, if `getParent()` returns null, the component is not attached to any parent and is not connected to the hierarchy unless it is the top-level `RootModel` component (Section 2.3).

When a model component is added to a parent, its method connectToHierarchy() is called, and when it is removed from its parent, its method disconnectFromHierarchy() is called. When either of these methods are called, the component's parent (as returned by getParent()) will be valid, and so hierarchy-dependent initialization can be performed, like setting (or removing) back pointers to references, etc.:

```
connectToHierarchy () {
   ... perform hierarchy-dependent initialization ...
}

disconnectFromHierarchy (CompositeComponent parent) {
   ... undo hierarchy-dependent initialization ...
}
```

The methods `getHardReferences()` and `getSoftReferences()` are described in Section 1.2.

It is also necessary to notify components in the hierarchy when there are changes in structure or component properties, so that the necessary adjustments can be made, including the clearing of cached data. Notification is done using the method notifyParentOfChange(), which propagates an appropriate change event up the hierarchy. It will typically do this by calling the componentChanged() method of the parent (see Section 1.3).

The method hasState() should return `true` if the component contains state information. This is always true if the component contains dynamic state information such as positions or velocities, but components may sometimes contain additional state information (such as contact state). Structural changes involving the addtion or removal of state-bearing components should be announced to the system by calling `notifyParentOfChange()` with a StructureChangeEvent for which stateIsChanged() returns `true`.

A `ModelComponent` also maintains a number of flags:

```
// returns true if a component is selected
boolean isSelected();

// sets whether or not a component is selected (system use only)
void setSelected (boolean selected);

// returns true if a component should not be removed from its parent
boolean isFixed();

// sets whether or not a component should be removed from its parent
void setFixed (boolean fixed);

// returns true if a component is marked
boolean isMarked();

// sets whether or not a component is marked
void setMarked (boolean marked);
```

All of these flags are `false` by default.

selected indicates whether or not a component is selected. Components can be selected using various selection mechanisms in the Artisynth interface, such as the navigation panel or the viewer. When selected, its isSelected() method will return `true`.

fixed indicates, if `true`, that a component should not be removed from its parent. It is used to fix required child components of composite components that contain otherwise removable children (Section 1.4).

marked is available for use by graph-processing algorithms involving the component hierarchy, to indicate when a component has been visited or otherwise processed. This flag should be used with care to avoid side effects.

> **Important:**
> setSelected() should only be used by the SelectionManager, and should not be called by applications. Programmatic component selection should be performed by calling the `addSelected()` or `removeSelected()` methods of the `SelectionManager`.

Finally, all `ModelComponents` implement the interface Scannable, which provides methods for writing and scanning to and from persistent storage. Details are given in Section 3.

For convenience, ModelComponentBase provides a base implementations of all the `ModelComponent` methods. Most ArtiSynth components inherit from `ModelComponentBase`.

Component References

Model components can reference additional components outside of the parent-child relationships of the hierarchy. For example, a point-to-point spring contains two *references* to its end-points, which are themselves model components. As another example, components which implement the `ExcitationComponent` interface can maintain references to other `ExcitationComponents` to use as excitation sources. References can be considered to be either *hard* or *soft*. A `hard` reference is one which the component requires in order for its continued existence to be meaningful. The end-point references for a point-to-point spring are usually hard. A *soft* reference is one that the component can do without, such as the excitation source inputs mentioned above. The methods getHardReferences() and getSoftReferences() are used to report all hard and soft references held by a component.

> **Note:**
> `getHardReferences()` and `getSoftReferences()` should report only references held by the component itself, and *not* those held by any of its descendents.

The distinction between hard and soft references is used by the system when components in the hierarchy are deleted. A component that holds a hard reference to a deleted component is generally deleted as well. However, when only soft references are deleted, then the updateReferences() method of the referring component is called to update the component's internal structures. `updateReferences()` should also store information about the update, to allow the update to be undone in case the method is called later with its `undo` argument set to `true`. A typical implementation pattern for `updateReferences()` is shown by the following example, in which `maspack.util.ListRemove` is used to remove selected items from a list of soft references, and store information needed to undo this later:

```
ArrayList<ModelComponent> mySoftRefs;
...
void updateReferences (boolean undo, Deque<Object> undoInfo) {
   super.updateReferences (undo, undoInfo);
   if (undo) {
      // undo the update
      Object obj = undoInfo.getFirst();
      if (obj != ModelComponentBase.NULL_OBJ) {
         ((ListRemove<ModelComponent>)obj).undo();
      }
   }
   else {
      // remove soft references which aren't in the hierarchy any more:
ListRemove<ModelComponent> remove = null;
      for (int i=0; i<mySoftRefs.size(); i++) {
```

```
            if (!ComponentUtils.isConnected (this, mySoftRefs.get(i)) {
         // reference isn't in the hierarchy; request its removal
         if (remove == null) {
                  remove = new ListRemove<ModelComponent>(mySoftRefs);
               }
         remove.requestRemove (i);
            }
         }
         if (remove != null) {
            remove.remove();
      undoInfo.addLast (remove);
         }
         else {
      undoInfo.addLast (ModelComponentBase.NULL_OBJ);
         }
      }
   }
```

When updating, the method uses ComponentUtils.isConnected() to determine which soft references have been deleted from the hierarchy. A `ListRemove` object is used to assemble the remove requests and then perform the remove all at once and store information about what was removed for possible later undoing. The remove object is appended to the end of `undoInfo`. If no undo was needed, then `NULL_OBJ` is stored instead because `Deque` objects don't accept `null` arguments. Undo information is stored at the end of the deque and removed from the front. This allows multiple updates, including that for the super class, to be performed in sequence.

Composite Components

CompositeComponent is a subinterface of ModelComponent which can contain children. Its main methods include:

```
// returns the number of child components
int numComponents();

// gets a child component by name
ModelComponent get (String name);

// gets a child component by index
ModelComponent get (int idx);

// gets a child component by number
ModelComponent getByNumber (int num);

// returns the index of a child component
int indexOf (ModelComponent c);

// finds a descendent component with a specified path relative to this component
ModelComponent findComponent (String name);

// called when a change occurs in one of the descendants.
void componentChanged (ComponentChangeEvent e);
```

Most of the above methods are self-explanatory. It is important to note the difference between indices and numbers when identifying child components. An *index* is the location of the child within the list of children, starting from 0, and can change as children are added or removed. A *number*, on the other hand, as described above, is assigned automatically to a child when it is added to the parent and persists as long as it remains.

The `componentChanged()` method is called to indicate structure or property changes. Appropriate actions may include clearing cached data, and propogating the event further up the hierarchy (using `notifyParentOfChange()`).

MutableCompositeComponent is a subinterface of CompositeComponent which allows child components to be added and removed by an ArtiSynth application. It is a generic class parameterized by a class type `C` which must be an extension of `ModelComponent`. Its definition is:

```
public interface MutableCompositeComponent<C extends ModelComponent>
```

```
extends CompositeComponent {

   // add a component; return false if not possible
   public boolean add (C comp);

   // add a set of components at specified index locations
   // (mainly for internal system use)
   public void addComponents (ModelComponent[] comps, int[] indices, int num);

   // remove a component; return false if not found
   public boolean remove (C comp);

   // removes a set of components and stores their original
   // index locations (mainly for internal system use)
   public void removeComponents (ModelComponent[] comps, int[] indices, int num);
}
```

CompositeComponentBase, ComponentList, and ComponentListImpl

A default implementation of CompositeComponent is provided by CompositeComponentBase. It is a non-generic class that provides a base for composite components whose composition is created at construction time and is not intended to change during the running of an ArtiSynth application.

ComponentList is a much more flexible class which implements MutableCompositeComponent and provides for collections of components whose composition may be built and changed by an application. `ComponentList` is used widely to store the many lists of components that comprise a working ArtiSynth model.

[]

In particular, MechModel and FemModel3d, the primary ArtiSynth classes for implementing mechanical and finite element models, are themselves subclasses of `ComponentList` which contain lists of mechanical components (such as particles, rigid bodies, and force effectors for `MechModel`, and nodes, elements, and geometry for `FemModel3d`).

In the case of `MechModel`, applications can create and add their own component lists to the model itself:

```
   MechModel mech;

   ....

   ComponentList<Particle> bigParticles =
      new ComponentList<Particle>(Particle.class, "big");

   ComponentList<Particle> smallParticles =
      new ComponentList<Particle>(Particle.class, "small");

   mech.add (bigParticles);
   mech.add (smallParticles);
```

> By default, child components that belong to a MutableCompositeComponent (which includes `ComponentList`) may be selected by the ArtiSynth application for deletion. This may be undesirable, particularly if internal structures depend on certain child components. Components that should not be removed from their parents should have their *fixed* flag set to `true` in the composite component constructor, either by calling setFixed(), or by adding the component using the addFixed() method of `ComponentList`.

The class ComponentListImpl is available as an internal implementation class for constructing instances of either CompositeComponent or MutableCompositeComponent. It provides most of the implementation methods needed for a mutable component list, which can be exposed in the client class using delegate methods. Components implementing only `CompositeComponent` may choose to expose only some of these methods. For details, one should consult the source code for CompositeComponentBase or ComponentList.

Models

Models and State

A Model is a specific ModelComponent that can contain state and be advanced forward in time.

The methods associated with time advancement are:

```
// initialize the model for time t
void initialize (double t)

// prepare to advance the model from time t0 to t1
StepAdjustment preadvance (double t0, double t1);

// advance the model from time t0 to t1
StepAdjustment advance (double t0, double t1);

// gets the maximum step size for advancement (or -1 if undefined)
double getMaxStepSize();
```

`initialize()` is called to initialize the model for a particular time. It is called at the beginning of a simulation (with time t = 0), when the model is moved to a state and time defined by a WayPoint, and when a step is repeated during adaptive stepping (Section 2.5).

`preadvance()` is called to prepare the model for advancement from time `t0` to `t1`. Often this method does nothing; it is supplied for situtations where the model needs to perform computation *before* the application of controllers or input probes (Section 2.2), such as evolving internal state in some way. The method can optionally return a StepAdjustment object to request a change in step size (Section 2.5).

`advance()` is called to advance the model from time `t0` to `t1`. This is the main driver method for simulation, and typically involves solving an ordinary differential equation (ODE) associated with an underlying mechanical system, for which the model employs an internal physics solver. The method can optionally return a StepAdjustment object to request a change in step size (Section 2.5).

A very basic simulation might proceed as follows:

```
t = 0;
model.initialize (t);
while (simulating) {
  model.preadvance (t, t+h);
  model.advance (t, t+h);
  t = t+h;
}
```

The rate of advancement (`h` in the above example) is limited by the model's *effective step size*, which is nominally the maximum step size of the root model (Section 2.3). The model can override this by providing its own maximum step size (via `getMaxStepSize()`) that is less than that of the root model. Advance intervals can be smaller than the effective step size, if required by other time events imposed by WayPoints, rendering, or output probes. The effective step size may also be reduced when adaptive stepping is employed (Section 2.5).

A model can contain state, which is defined to be all information needed to deterministically advance it forward in time.

Models can contain state, as supported by the following methods:

```
// creates an appropriate state object for storing model state
ComponentState createState ();

// gets the current state for this model
void getState (ComponentState state);

// sets the current state for this model
void setState (ComponentState state);
```

If a model actually maintains state, then its `hasState()` method (inherited from `ModelComponent`) should return `true`, and `createState()` should create an appropriate object for saving and restoring the state using using `getState()` and `setState()`.

The state of a model should contain all the internal information required to advance it forward in time. In particular, in the code fragment,

```
model.getState (state);  // save state
model.preadvance (t1, t2);
model.advance (t1, t2);
model.setState (state);  // restore state and
model.initialize (t1);   // reinitialize to time t1
model.preadvance (t1, t2);
model.advance (t1, t2);
```

the model should have the exact same state and appearance after both the first and seconds calls to `advance()` (the call to `initialize()` is used to reset time-dependent quantities, such as time-dependent forces). For mechanical systems, the most prominent state quantities are the positions and velocities of the dynamic components, but there can be other quantities as well, such as contact state and viscoelastic state for FEM models.

Model Agents

As models are advanced, auxiliary agents can be employed to control the inputs and observe the outputs of the model. These include *probes*, *controllers*, and *monitors*.

Probes

A Probe is an agent that sets model input data, or records model output data, over a specific window of time. Probes that set input data are *input probes* (InputProbe), while those that record output data are *output probes* (OutputProbe). Examples of input data include muscle excitation signals or external forces. Output data often includes items such as velocities, postions, or internal forces.

Input probes can be used to perform a function analagous to the “loading curves” used in FEM analysis.

A probe contains several principal methods:

```
// apply this probe at time t
apply (double t);

// returns the start time of this probe
double getStartTime();

// sets the start time
void setStartTime (double t);

// returns the stop time of this probe
double getStopTime();

// sets the stop time
void setStopTime (double t);

// returns the update interval of this probe (or -1 if undefined)
double getUpdateInterval();

// sets the update interval
void setUpdateInterval (double dt);

// returns true if this probe is active
boolean isActive();

// sets whether or not this probe is active
void setActive (boolean enable);
```

The task of applying input data or recording output data is performed in the `apply()` method, which is called periodically during the time window delimited by `getStartTime()` and `getStopTime()`.

The methods `isActive()` and `setActive()` control whether or not the probe is active. Inactive probes will not have their `apply()` method called by the system. Probe activity is exported as the property active, and allows probes to be enabled or disabled at run time.

For input probes, the `apply()` method is called between the the `preadvance()` and `advance()` methods of the model it is associated with (Section 2.2.3). For output probes, `apply()` is called after the model's `advance()` method, whenever the time advanced to equals an update time for the probe. Update times for an output probe are given by the start and stop times, plus any time that is an integer multiple of its *effective update interval*. The effective update interval is given by either the value returned by `getUpdateInterval()`, if it is not undefined (i.e., equal to -1), or the effective step size for the probe's associated model (Section 2.1).

Note that the start time, stop time, and update interval can also be observed and controlled via the properties `startTime`, `stopTime`, and `updateInterval`.

The most common types of probes used in ArtiSynth are NumericInputProbe and NumericOutputProbe, which are used to connect model properties to streams of numeric data which can be edited and observed on the system's timeline. Numeric probe data can also be saved to (or loaded from) external files.

Controllers and monitors

Controllers and monitors are other agents that can be used to control or observe a simulation. Controllers are called immediately before a model's `advance()` method (and after the `preadvance()` method and the application of any input probes) and are intended to compute control signals, while monitors are called immediately after the `advance()` method and are intended to record and process output data.

The primary method for both is

```
void apply (double t0, double t1)
```

which performs the work of the agent. The times `t0` and `t1` are the same times passed to the model's `advance()` method.

As with probes, controllers and monitors can be active or inactive, as determined the method `isActive()`. Controllers or monitors which are based on the default implementation classes ControllerBase or MonitorBase export also provide a `setActive()` method to control this setting, and export it as the property active. This allows controller and monitor activity to be controlled at run time.

Models associated with agents

As indicated above, model agents are typically associated with a specific model within the ArtiSynth structure, and are then applied either before or after the `advance()` method of that model. Agents which are not explicitly associated with a model are implicitly associated with the root model Section 2.3.

Methods to obtain and set the associated model include:

```
// returns the model (if any) associated with this agent
Model getModel();

// sets the model to be associated with this agent
void setModel (Model model);

// searchs for the model to be associated with this agent
void setModelFromComponent (ModelComponent comp);
```

`setModel()` sets the model directly, while `setModelFromComponent()` takes a subcomponet of a model and searches up the hierarchy to find the model itself. For example, when connecting a component property to a numeric probe, the system automatically determines the probe's model by calling `setModelFromComponent()` on the component hosting the property.

Model agent state

Like models, agents can also have state, and therefore implement the same methods `hasState()`, `createState()`, `getState()`, and `setState()` described for models in Section 2.1.

The base classes for probes, controllers, and monitors define stateless version of these methods (i.e., `hasState()` returns `false`, and `getState()` and `setState()` do nothing), so that agents which actually do contain state must override these methods.

In the context of agents, state can be thought of as the internal information that is required so that the agent's actions and effect on its associated model are always identical for a specific time and state. A common example of state in the context of a controller or monitor might be the time history used to filter a signal.

Note that the requirement "effect on its associated model" means that state is also needed for input probes to handle situations when the simulation is moved to a time and state defined by a WayPoint. That's because probes are applied to a model only over a specific time window, and so when the simulation is reset to a time outside that window, it is usually necessary to reset the model attributes controlled by the probe to their original values at that time. As a simple example, assume that the `apply()` method of an input probe sets a value x in a model to 10 over the time window [2,4], and that before that time, x has a value of 0. Now if time is advanced to t = 3, x will be set to 10, and if time is then reset to t = 1 (before the probe's window), x will need to be restored to 0. This must be done by restoring the probe's state, since the `apply()` method will not be called at t = 1.

At present, NumericInputProbe defines `getState()` and `setState()` to save and restore any model property values that it controls. No state is defined for NumericOutputProbe.

> If a controller has state, then it is important to implement `getState()` and `setState()` to ensure proper behavior with respect to both adaptive stepping Section 2.5 and WayPoints. If a probe or monitor has state, implementation of `getState()` and `setState()` is necessary to ensure its proper behavior with respect to WayPoints, but not adaptive stepping, since probe and monitor state is not changed during adaptive stepping).

The Root Model

All the models simulated by ArtiSynth at any given time are collected together within a root model (RootModel), which is the top-level model component in the hierarchy. Every system that is simulated by ArtiSynth is associated with a specific instance of a `RootModel`, and is typically created in code by subclassing `RootModel` and then creating and assembling the necessary components in the subclass's constructor. Alternately, a `RootModel` can be loaded from a file, in which case a generic `RootModel` is created and then populated with structures determined from the file.

A `RootModel` contains a list of all the system's models, probes, controllers, and monitors. Methods to add or remove these include:

```
addModel (Model model);
removeModel (Model model);
removeAllModels();

addInputProbe (InputProbe probe);
removeInputProbe (InputProbe probe);
removeAllInputProbes();

addController (Controller controller);
removeController (Controller controller);
removeAllControllers();

addMonitor (Monitor monitor);
removeMonitor (Monitor monitor);
removeAllMonitors();

addOutputProbe (OutputProbe probe);
removeOutputProbe (OutputProbe probe);
removeAllOutputProbes();
```

Advancing Models in Time

The root model is the top-level object seen by the ArtiSynth scheduler when running a simulation. As a simulation proceeds, the scheduler determines the next time to advance to and then calls the root model's `advance()` method:

```
double t0; // current time

while (simulating) {
   t1 = getNextAdvanceTime (t0);
   rootModel.advance (t0, t1);
}
```

In turn, the `advance()` method then individually advances all the models contained in the root model, as described below. The next advance time `t1`, computed by `getNextAdvanceTime()`, is determined mainly by the root model's maximum step size (returned by `RootModel.getMaxStepSize()`), along with other events such as WayPoint locations and the render update rate.

The root model's maximum step size is therefore the primary simulation step size. It can be set using the method `RootModel.setMaxStepSize()`, and is exposed as the `RootModel` property `maxStepSize`. It is also coupled to the "step" display in the ArtiSynth GUI, and can also be obtained or set using `Main.getMaxStep()` or `Main.setMaxStep()`. When a `RootModel` is created, if its maximum step size is not set explicitly (either in the constructor or in a file specification), then it is set to a default value which is either 0.01, or the value specified by the `-maxStep` command line argument.

A `RootModel` advances each of its models in sequence, using a procedure called `advanceModel()`. Because a model may have a maximum step size (as returned by `getMaxStepSize()`) that is less than that of the root model, or some of its output probes may have events that preceed `t1`, each model is advanced using a series of sub-advances, with `advanceModel()` taking the form of a loop:

```
advanceModel (model, t0, t1) {
   ta = t0;
   while (ta < t1) {
      tb = getNextAdvanceTime (model, ta, t1);
model.preadvance (ta, tb);
      applyInputProbes (model, tb);
      applyControllers (model, ta, tb);
      model.advance (ta, tb);
      applyMonitors (model, ta, tb);
      applyOutputProbes (model, tb);
      ta = tb;
   }
}
```

Each time through the loop, `getNextAdvanceTime()` determines the next appropriate sub-advance time `tb`, and then calls the model's `advance()` method, surrounded by the application of any probes, controllers or monitors that are associated with it. The `preadvance()` method is called first, followed by input probes and controllers. Then `advance()` is called, monitors are applied, and output probes are applied if their next update time is equal to `tb`.

> The apply time for input probes is not the time `ta` at the beginning of the time step, but rather the time `tb` corresponding to its end. This might seem counterintuitive, but makes sense when one considers that input probes are generally used to provide *targets* for the advance process, and we typically want targets specified for the end of the time step. If the target is a force, then this is also consistent with implicit integration methods (used most commonly by ArtiSynth) which solve for the system forces at the end of the time step.

The `RootModel advance()` method in turn calls `advanceModel()` for all models, surrounded by the application of any probes, controllers and monitors which do not have specific models of their own and are therefore considered to be "owned" by the root model. Because some of these output probes may have event times that preceed the desired advance time of `t1`, this process is also done in a loop:

```
advance (t0, t1) {
   ta = t0;
   while (ta < t1) {
```

```
      tb = getNextAdvanceTime (root, ta, t1);
      applyInputProbes (root, tb);
      applyControllers (root, ta, tb);
      for (each model m) {
         advanceModel (m, ta, tb);
      }
      applyMonitors (root, ta, tb);
      applyOutputProbes (root, tb);
      ta = tb;
   }
}
```

Note that at present, the `preadvance()` method for a root model does nothing and is not called.

Adaptive Stepping

It is possible for models to request adaptive time stepping, which may be necessary if the model determines that a requested time step is too large for stable simulation. The model can indicate this by having either its `preadvance()` or `advance()` methods return a StepAdjustment object, which contains a recommended scaling for the step size via its `scaling` attribute. Then, if adaptive stepping is enabled in the root model, it will reduce the effective time step for the model and redo the advance. If `preadvance()` or `advance()` return `null`, then it is assumed that the step size should remain unchanged (which is equivalent to returning a `StepAdjustment` with a scaling of 1).

Adaptive time stepping can be enabled or disabled using the `adaptiveStepping` property of `RootModel`, or by using the `RootModel` methods

```
boolean getAdaptiveStepping();

void setAdaptiveStepping (boolean enable);
```

When adaptive stepping is enabled, the inner loop of the `advanceModel()` procedure described above is modified to the following:

```
   while (ta < t1) {
      tb = getNextAdvanceTime (model, ta, s);
      model.getState (state);
      do {
   s = GET_SCALING (model.preadvance (ta, tb));
   if (s >= 1) {
            applyInputProbes (model, tb);
            applyControllers (model, ta, tb);
            s = GET_SCALING (model.advance (ta, tb));
         }
         if (s < 1) {
            tb = reduceAdvanceTime (model, ta, tb, s);
            model.setState (state);
            model.initialize (ta);
         }
      }
      while (s < 1);
      applyMonitors (model, ta, tb);
      applyOutputProbes (model, tb);
      ta = tb;
   }
```

where `GET_SCALING()` returns 1 if `preadvance()` or `advance()` returns null, or the value of `StepAdjustment.getScaling()` otherwise.

At the beginning of the loop, the model's state is saved in case a retry is necessary. Then if `preadvance` or `advance()` recommend scaling the step by $s < 1$, the advance time is reduced (by reducing the model's effective step size), the state is restored to what is was at the beginning of the step, and the step is retried. After a step succeeds, the root model will incrementally try to increase the effective step size, up to its nominal value.

The exact interpretation of the scaling value s is as follows:

s = 0: Advance unsuccessful; no recommendation as to how much to reduce the next step.

0 < s < 1: Advance unsuccessful; recommend trying to reduce the step by s*(tb-ta).

s = 1: Advance successful, no recommendation as to how much to increase the step by.

s > 1: Advance successful; recommend trying to increase the step by s*(tb-ta).

When requesting a step size reduction, models may provide a string message indicating the reason via the `message` attribute of `StepAdjustment`. The system will abort if the effective step size falls below the minimum value specified by the `RootModel` property `getMinStepSize`. At present, recommended increases in step size are ignored and treating simply as s = 1.

> Models are of course free to implement adaptive stepping internally, in a way that is invisible to the root model. However, the saving and restoring of state, along with the algorithms for step size adjustment, are sufficiently intricate that it is generally convenient to use the adaptive stepping provided by `RootModel`.

Writing and Scanning Components

ArtiSynth model components have the ability to save and restore themselves from persistent storage. They do this by implementing the write() and scan() methods of maspack.util.Scannable, and the postscan() method of ModelComponent:

```
// write this component to a PrintWriter:
void write (PrintWriter pw, NumberFormat fmt, Object ref) throws IOException

// scan the component from a ReaderTokenizer:
void scan (ReaderTokenizer rtok, Object ref) throws IOException

// perform the post-scan pass for this component:
void postscan (Deque<ScanToken> tokens, CompositeComponent ancestor);

// normally returns true but can be overridden to return false
// if for some reason a component should be written to secondary storage
boolean isWritable();
```

The operation and implementation of these methods will now be described in detail. A summary of the key points is given in Section 3.5.

Writing components

The write() method writes information about the component to a `PrintWriter`, using NumberFormat to format floating point numbers where appropriate. The `ref` argument is used to provide additional context information for generating the output, and is specifically used to generate path names for other components that are referenced by the component being written (Section 3.1.1).

In general, each component writes out its attributes as a list of name/value pairs, each of the form

```
<name>=<value>
```

with the list itself enclosed between square brackets '`[ ]`' which serve as *begin* and *end* delimiters. The value associated with each attribute name may itself be a quantity (such as a vector, matrix, or another component) delimited by square brackets. For example, the output for a Particle component may look like this:

```
[ name="primary"
  position=[ 15.0 0.0 10.0 ]
  mass=20.0
  dynamic=false
]
```

The output begins with an openning square bracket, followed by four attribute/value pairs and a closing square bracket. The position attribute is a 3-vector also enclosed between square brackets.

Within the `write()` method, the above output could be produced with code like this:

```
import maspack.util.*;

void write (PrintWriter pw, NumberFormat fmt, Object ref) throws IOException {
   pw.print ("[ ");
   IndentingPrintWriter.addIndentation (2);
   pw.println ("name=" + Write.getQuotedString(myName));
   pw.println ("position=[" + myPosition.toString (fmt) + "]");
   pw.println ("mass=" + fmt.format(myMass));
   pw.println ("dynamic=" + myDynamicP;
   IndentingPrintWriter.addIndentation (-2);
   pw.println ("]");
}
```

Output indentation can be controlled by using an IndentingPrintWriter, and IndentingPrintWriter.addIndentation() will increase (or decrease) output indentation if `pw` is an instance of `IndentingPrintWriter`.

In practice, components do not generally need to provide explicit code to write out all their attribute values. In particular, any information that is associated with a Property (see the Property section of the Maspack Reference Manual) can be written out automatically using a code fragment of the form:

```
getAllPropertyInfo().writeNonDefaultProps (this, pw, fmt);
```

In addition, any attribute information contained in a component's superclass will usually be written by that superclass. The default `write()` definition for a model component is usually looks something like this:

```
public void write (PrintWriter pw, NumberFormat fmt, Object ref)
throws IOException {

   dowrite (pw, fmt, ref);
}

protected void dowrite (PrintWriter pw, NumberFormat fmt, Object ref)
throws IOException {

   CompositeComponent ancestor = ComponentUtils.castRefToAncestor(ref);
   IndentingPrintWriter.printOpening (pw, "[ ");
   IndentingPrintWriter.addIndentation (pw, 2);
   writeItems (pw, fmt, ancestor);
   IndentingPrintWriter.addIndentation (pw, -2);
   pw.println ("]");
}

protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
throws IOException {

   getAllPropertyInfo().writeNonDefaultProps (this, pw, fmt);
}
```

The `write()` and `dowrite()` methods take care of writting the square brackets and setting up the initial indentation. Then a call to `writeItems()` prints out all necessary property values. The `ancestor` argument obtained from `ref` will be discussed in Section 3.1.1.

If all a component's attribute information is associated with property values, then it is usually not necessary to provide any component-specific code for writing the component: the default implementations of `write()` and `writeItems()` will handle it. If a component *does* have attribute information that is not associated with a property, then it is usually sufficient to handle this by overriding `writeItems()`. For example, if a component has a non-property "centroid" attribute, it can be written by an override of `writeItems()` constructed like this:

```
protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
```

```
   throws IOException {

      super.writeItems (pw, fmt, ref);
      pw.println ("centroid=[" + getCentroid().toString (fmt) + "]");
   }
```

Writing references

ArtiSynth components often contain references to other components that are not part of their ancestor hierarchy. For example, a two-point spring will contain references to its two end-point particles, and a FemElement3d will contain references to its nodes. The set of all references refered to by a component is returned by the combination of the component's getHardReferences() getSoftReferences() methods.

Because they generally reside outside a component's immediate ancestor hierarchy, information about each reference's location needs to be explicitly written and scanned as part of writing and scanning a component. The location information is stored using the component's path with respect to some known *ancestor*. This ancestor is passed to the component's `write()` method through the `ref` argument, and is cast explicitly to `CompositeComponent` and passed to `writeItems()` as the `ancestor` argument. A component can then use ComponentUtils.getWritePathName() to obtain the path name of each reference with respect to the ancestor, and write this to the output.

As an example, here is a possible implementation of `writeItems()` for a two-point spring:

```
   protected void writeItems (
   PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
   throws IOException {

      super.writeItems (pw, fmt, ancestor);
      pw.println ("point0=" + ComponentUtils.getWritePathName(ancestor,myPnt0));
      pw.println ("point1=" + ComponentUtils.getWritePathName(ancestor,myPnt1));
   }
```

This will produce an output like this,

```
      point0="models/points/0"
      point1="models/points/1"
```

where `models/points/n` gives the path name of the n-th point with respect to the ancestor. Alternatively, if a component has a variable number of references, they can be written out as a list between square brackets:

```
   protected void writeItems (
   PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
   throws IOException {

      super.writeItems (pw, fmt, ancestor);
      pw.println ("points=[");
      IndentingPrintWriter.addIndentation (pw, 2);
      for (ModelComponent pnt : myPnts) {
         pw.println (ComponentUtils.getWritePathName(ancestor,pnt));
      }
      IndentingPrintWriter.addIndentation (pw, -2);
      pw.println ("]");
   }
```

The above code will produce output like this:

```
      points=[
        "models/points/0"
        "models/points/1"
        "models/points/2"
      ]
```

The ancestor used for reading and writing references will always be a common ancestor of both the references and the refering component. This may sometimes be the root model (i.e., the top of the hierarchy), but more typically it will be the first common ancestor for which hierarchyContainsReferences() returns `true` (implying that all references are contained within the ancestors's descendants). This allows paths to be written more compactly. Most Model components presently enforce the `hierarchyContainsReferences()` condition.

Writing child components

If a component is a `CompositeComponent`, then it also needs to write out its child components along with its attribute information. This can be done by recursively calling the children's `write()` methods.

If the child component configuration is fixed (i.e., the component does not implement MutableCompositeComponent) and the children are created in the composite's constructor, then attribute names can be used to delimit each child. For example, suppose a component contains two children: a list of particles and a list of springs. This component could then be written using a code construction such as:

```
protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
throws IOException {

   super.writeItems (pw, fmt, ref);
   pw.print ("particles=");
   myParticles.write (pw, fmt, ref);
   pw.print ("springs=");
   mySprings.write (pw, fmt, ref);
}
```

If the composite component has been implemented internally using an instance of ComponentListImpl (Section 1.4), then the above can be written using the latter's writeComponentsByName() method, which writes each child, using its component name as an attribute name:

```
ComponentListImpl myComponents;

protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
throws IOException {

   super.writeItems (pw, fmt, ref);
   myComponents.writeComponentsByName (pw, fmt, ancestor);
}
```

One the other hand, composite components which are instances of MutableCompositeComponent may not have predetermined component arrangements and so these components cannot be identified by an attribute name. Instead, the components must simply be printed out in sequence. For example, the `writeItems()` method for a list of particles could be programmed like this:

```
protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
throws IOException {

   super.writeItems (pw, fmt, ref);
   for (int i=0; i<particles.size(); i++) {
      particles.get(i).write (pw, fmt, ref);
   }
}
```

This would produce output such as:

```
[ position=[ 10.0 0.0 20.0 ]
  mass=20.0
]
[ position=[ 5.0 0.0 10.0 ]
```

```
  mass=15.0
]
[ position=[ 15.0 0.0 10.0 ]
  mass=20.0
]
[ position=[ 10.0 0.0 0.0 ]
  mass=15.0
]
```

However, when scanning a MutableCompositeComponent, the `scan()` method (discussed below) needs to create and scan new child components as it encounters them in the input. It is therefore necessary for `scan()` to know what class of component to create. Typically, the composite component has a default component type; for example, the default type for `ComponentList<Particle>` is Particle. However, in some cases the components may be subclasses of the default type, or the default type may be an interface or abstract class and hence not instantiable. In such instances, the output needs to be augmented with class type information, which is placed before the openning '`[`' of the component output. The details of how to do this are beyond the scope of this document. However, if the composite component has been implemented internally using an instance of ComponentListImpl, then one can use the latter's writeComponents() method to automatically write out all the components, along with the necessary class information:

```
ComponentListImpl myComponents;

protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)
throws IOException {

   super.writeItems (pw, fmt, ref);
   myComponents.writeComponents (pw, fmt, ancestor);
}
```

The result may look something like this:

```
[ position=[ 10.0 0.0 20.0 ]
  mass=20.0
]
artisynth.core.mechmodels.SpecialParticle [
  position=[ 5.0 0.0 10.0 ]
  mass=15.0
]
[ position=[ 15.0 0.0 10.0 ]
  mass=20.0
]
user.projects.CustomParticle [
  position=[ 10.0 0.0 0.0 ]
  mass=15.0
]
```

In this example, the second and fourth particles have class types that differ from the default and so class information for each is prepended to the output.

Scanning components

The scan() method reads the component in from a token stream provided by a ReaderTokenizer. This translates the input into a stream of tokens, including words, numbers, and special token characters (such as '`[`', '`]`', and '`=`'), which are then used to parse the input. Authors implementing scanning code should have some familiarity with `ReaderTokenizer`. A description is beyond the scope of this document but good documentation is available in the ReaderTokenizer class header.

The main code inside the default `scan()` method for a model component looks roughly like this:

```
public void scan (ReaderTokenizer rtok, Object ref)
throws IOException {

   rtok.scanToken ('[');
```

```
      while (rtok.nextToken() != ']') {
         rtok.pushBack();
         if (!scanItem (rtok, tokens)) {
            throw new IOException ("Unexpected token: " + rtok);
         }
      }
   }
```

The method looks for and scans the initial '[' character (and will throw an `IOException` if this is not found). It then reads other tokens (using nextToken()) until the terminating ']' character is found. After each token is inspected, it is pushed back into the token stream using pushBack() and `scanItem()` is called to try and read an individual attribute or subcomponent from the input. If `scanItem()` cannot match the input to any attributes or child components it returns `false`.

> Note: ReaderTokenizer allows one token of look-ahead, so that any read token can be pushed back once. In particular, in the following sequence, `t1` and `t2` should be the same:
>
> ```
> t1 = rtok.nextToken();
> rtok.pushBack();
> t2 = rtok.nextToken();
> ```

The default implementation of `scanItem()` provides code for reading property values and looks something like this:

```
   protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
   throws IOException {

      rtok.nextToken();
      // if attribute name is a property name, scan that property:
      if (ScanWriteUtils.scanProperty (rtok, this)) {
         return true;
      }
      rtok.pushBack();
      return false;
   }
```

The method begins by getting the next token and then calling ScanWriteUtils.scanProperty() to see if the token is a word matching the name of one of the component's properties. If so, then `scanProperty()` scans and sets the property value and returns `true`, and `scanItem()` itself returns `true`. Otherwise, `scanItem()` returns `false` indicating that it was unable to find a match for the input. The `tokens` argument is used to store information whose processing must be deferred until the post-scan step, as discussed in Section 3.2.1.

Typically, component implementations will not need to override `scan()` unless the scanning procedure calls for pre- or post-processing, as in:

```
   public void scan (ReaderTokenizer rtok, Object ref)
   throws IOException {

      ... do pre-processing here ...
      super.scan (rtok, ref);
      ... do post-processing here ...
   }
```

Note that if a class makes use of the post-scan step (Section 3.2.1), then it may be necessary to do the post-processing in an override of `postscan()` instead.

Component implementations often *will* need to override `scanItem()` to scan additional attribute information. For example:

```
   protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
   throws IOException {

      rtok.nextToken();
```

```
      if (rtok.ttype == ReaderTokenizer.TT_WORD) {
         if (rtok.sval.equals ("attributeXXX") {
      rtok.scanToken ('=');
      ... scan information for attribute XXX ...
            return true;
         }
      }
      rtok.pushBack();
      return super.scanItem (rtok, tokens);
   }
```

First, the method gets the next token. Then it checks if its type (`ttype`) corresponds to a `WORD` token and if the word's string value (`sval`) equals the attribute name `attributeXXX`. If so, then it scans the '=' character following attribute name, scans whatever information is associated with the attribute, and returns `true`. Otherwise, if no expected attribute name is matched, the current token is pushed back, and the superclass method is called to see if it can match the current input.

Most implementations of `ModelComponent` provide the convenience method `scanAttributeName(rtok, name)` which allows the code fragment

```
      if (rtok.ttype == ReaderTokenizer.TT_WORD) {
         if (rtok.sval.equals ("attributeXXX") {
      rtok.scanToken ('=');
```

to be replaced with

```
       if (scanAttributeName (rtok, "attributeXXX")) {
```

Employing this in a larger example, we have

```
   protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
   throws IOException {

      rtok.nextToken();
      if (scanAttributeName (rtok, "name"))
         myName = rtok.scanQuotedString();
         return true;
      }
      else if (scanAttributeName (rtok, "position")) {
         myPosition.scan (rtok);
         return true;
      }
      else if (scanAttributeName (rtok, "mass")) {
         myMass = rtok.scanNumber();
         return true;
      }
      else if (scanAttributeName (rtok, "dynamic")) {
         myDynamicP = rtok.scanBoolean();
         return true;
      }
      rtok.pushBack();
      return super.scanItem (rtok, tokens);
   }
```

Here, if any of the attribute names `name`, `position`, `mass`, or `dynamic` are matched, then the corresponding string, vector, numeric, or boolean attribute values are scanned using scanQuotedString(), the vector's own `scan()` method, scanNumber(), or scanBoolean(). Each of these will throw an `IOException` if the input token sequence does not match what is expected.

Scanning references and post-scanning

When scanning a component that contains references, the path for each reference is used to locate the referenced component within the component hierarchy. However, this poses a problem: because components are created only as the

component hierarchy is recursively scanned, it is possible that some references may not yet exist at the time when the component is scanned. For example, if the points referenced by a two-point spring belong to part of the hierarchy further "to the right" of the spring components, then when the spring is scanned the points won't yet exist and the scanning method will be unable to find them.

The solution to this problem is to employ a two-step scanning process in which the initial scan is followed by a secondary "post-scan" which can be used to resolve references. Each reference path found during the initial scan is saved for later use in the post-scan step, by which time all components are guaranteed to have been created. Reference information, along with any other information needed for the post-scan step, is saved in a queue of ScanTokens supplied to the `scan()` method through the `ref` argument. Several different types of `ScanTokens` allow different types of information to be stored: StringTokens are use to store attribute names and reference paths; ObjectTokens are use to store object pointers; and special marker tokens, `ScanToken.BEGIN` and `ScanToken.END`, can be used as delimiters.

At a minimum, scanning each component causes `BEGIN` and `END` tokens to be added to the token queue, with additional tokens added in between as necessary. Revisiting the basic `scan()` method code shown at the top of Section 3.2, we show the additional code that is needed to handle this:

```
public void scan (ReaderTokenizer rtok, Object ref)
throws IOException {

   Deque<ScanToken> tokens = (Deque<ScanToken>)ref;
   tokens.offer (ScanToken.BEGIN);
   rtok.scanToken ('[');
   while (rtok.nextToken() != ']') {
      rtok.pushBack();
      if (!scanItem (rtok, tokens)) {
         throw new IOException ("Unexpected token: " + rtok);
      }
   }
   tokens.offer (ScanToken.END);
}
```

The token queue itself, called `tokens`, is obtained from the `ref` argument via an explicit cast. `BEGIN` and `END` tokens are added at the beginning and end of the scan. In between, the token queue is passed to `scanItem()`, which adds addtional tokens when necessary.

Within `scanItem()`, tokens are added to provide whatever information is needed for the post-scan step. This information is often provided in the form of two or more tokens comprising an attribute name/value pair, so that the post-scan step is not sensitive to input ordering. In this sense, the information stored in the token queue will reflect the same structure as the tokens in the original input.

Consider the first example in 3.1.1 where the reference information for a two-point spring was output as:

```
   point0="models/points/0"
   point1="models/points/1"
```

To process this inside `scanItem()`, we check for the attribute names `point0` and `point1` and if either is found, we store both the attribute name and the reference path in the token queue using `StringTokens`. For `point0`, the corresponding code looks like

```
   if (scanAttributeName (rtok, "point0")) {
      String refpath = rtok.scanWordOrQuotedString ('"');
tokens.offer (new StringToken ("point0"));
tokens.offer (new StringToken (refpath));
return true;
   }
```

Most implementations of `ModelComponent` provide the convenience method `scanAndStoreReference (rtok, name, tokens)` which allows this to be compressed into

```
   if (scanAndStoreReference (rtok, "point0", tokens)) {
return true;
   }
```

One may also use ScanWriteUtils.scanAndStoreReference() for the same purpose. The `scanItem()` method for a two-point spring can then be written as:

```
protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
throws IOException {

   rtok.nextToken();
   if (scanAndStoreReference (rtok, "point0", tokens)) {
      return true;
   }
   else if (scanAndStoreReference (rtok, "point1", tokens)) {
      return true;
   }
   rtok.pushBack();
   return super.scanItem (rtok, tokens);
}
```

Alternatively, if we have an attribute followed by a list of references enclosed in square brackets, such as

```
   points=[
     "models/points/0"
     "models/points/1"
     "models/points/2"
   ]
```

then we want to store a sequence of tokens consisting of the attribute name, a BEGIN token, the reference paths, and an END token:

```
   "points"
   BEGIN
   "models/points/0"
   "models/points/1"
   "models/points/2"
   END
```

That can be done by a code sequence that looks like

```
   if (scanAttributeName (rotk, "points", tokens)) {
      rtok.scanToken ('[');
      tokens.offer (new StringToken ("points"));
      tokens.offer (ScanToken.BEGIN);
      while (rtok.nextToken() != ']') {
   if (rtok.tokenIsWordOrQuotedString ()) {
      tokens.offer (rtok.sval);
         }
         else {
      throw new IOException ("Error: reference path expected");
         }
      }
      tokens.offer (ScanToken.END);
      return true;
   }
```

and which is available in most ModelComponent implementations via the convenience method `scanAndStoreReferences()`:

```
   if (scanAndStoreReferences (rtok, "point", tokens)) {
return true;
   }
```

One may also use ScanWriteUtils.scanAndStoreReferences() for the same purpose.

Scanning child components

In addition to their attributes, composite components need to scan in their child components. This can be done by recursively calling the children's `scan()` methods.

If the child component configuration is fixed (i.e., the component does not implement MutableCompositeComponent), and the children are created in the composite's constructor and written out using attribute names as delimiters, then these attributes names can be used to drive the scanning. The example composite from Section 3.1.2, comprising a list of particles and a list of springs, could be scanned in using code such as:

```
protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
throws IOException {

   rtok.nextToken();
   if (scanAttributeName (rtok, "particles"))
      tokens.offer (new ObjectToken (myParticles));
      myParticles.scan (rtok, tokens);
      return true;
   }
   else if (scanAttributeName (rtok, "springs")) {
      tokens.offer (new ObjectToken (mySprings));
      mySprings.scan (rtok);
      return true;
   }
   rtok.pushBack();
   return super.scanItem (rtok, tokens);
}
```

Here, an `ObjectToken()` identifying each scanned componet is stored on the token queue for later use in the post-scan step. If the composite component has been implemented internally using an instance of ComponentListImpl (Section 1.4), then the above can be written more succinctly using the latter's scanAndStoreComponentByName() method:

```
ComponentListImpl myComponents;

protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
throws IOException {

   rtok.nextToken();
   if (myComponents.scanAndStoreComponentByName (rtok, tokens)) {
      return true;
   }
   rtok.pushBack();
   return super.scanItem (rtok, tokens);
}
```

On the other hand, composite components which are instances of MutableCompositeComponent are written out in sequence, without using attribute names but with possible prefixed information giving information about the component's class. When scanning in these children, `scanItem()` must determine the class for the child, create an instance of the child, and then scan it in. The code required for these steps is beyond the scope of this document. However, if the mutable composite has been implemented internally using an instance of ComponentListImpl, then one can use the latter's methods scanBegin(), scanAndStoreComponent(), and scanEnd() to handle the scanning.

First, `scanBegin()` is called in an override of `scan()`:

```
ComponentListImpl myComponents;

public void scan(ReaderTokenizer rtok, Object ref) throws IOException {
   myComponents.scanBegin();
   super.scan (rtok, ref);
}
```

`scanAndStoreComponent()` can then be called in `scanItem()` to handle scanning of individual components:

```
protected boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
throws IOException {

   if (super.scanItem (rtok, tokens)) {
      return true;
   }
   rtok.nextToken();
```

```
      return myComponents.scanAndStoreComponent (rtok, tokens);
   }
```

The method checks to see if the input contains child component information, and if so, scans the information, creates an instance of the component if necessary, stores a copy of the component in the token queue for use in post-scanning, and returns `true`. Note that in this case one should call `scanAndStoreComponent()` *after* the `super` method to avoid confusing attribute names with class information.

> scanAndStoreComponent() will not create a new component if a *fixed* component (Section 1.4) of the appropriate class already exists at the current list position. Instead, the existing component will be scanned "in place".

Finally, `scanEnd()` is called in an override of `postscan()` method (discussed below):

```
   public void postscan (
   Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

      super.postscan (tokens, ancestor);
      myComponents.scanEnd();
   }
```

Post-scanning implementation

Once the token queue has been built by the `scan()` methods, it is processed in the post-scan step. This is done by each component using a `postscan()` method that takes as arguments the token queue and the ancestor with respect to which reference paths should be evaluated. The default `postscan()` method for most components looks something like this:

```
   public void postscan (
   Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

      ScanWriteUtils.postscanComponentBegin (tokens, this);
      while (tokens.peek() != ScanToken.END) {
         if (!postscanItem (tokens, ancestor)) {
            throw new IOException (
               "Unexpected token " + tokens.poll());
         }
      }
      tokens.poll(); // consume the END token
   }
```

postscanComponentBegin() gets the next token on the queue, checks that it is a `BEGIN` token, and throws an exception if this is not the case. Then the method simply calls `postScanItem()`, which does the actual token handling work, until a terminating `END` token is found.

As is the case with `scan()`, subclasses typically do not need to override `postscan()`. The exception to this is when post-processing is required after the scan process:

```
   public void postscan (
   Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

      super.postscan (tokens, ancestor);
      ... do post processing ...
   }
```

However, any component which adds tokens in its `scanItem()` method *will* need to process those tokens in an override of `postscanItem()`. Tokens can be removed from the queue using the queue's `poll()` method, and can be examined (without removing them) using the queue's `peek()` method. More usefully, the utility class ScanWriteUtils provides a number of methods for token processing, including:

```
   boolean postscanAttributeName (tokens, name);

   C postscanReference (tokens, clazz, ancestor);

   C[] postscanReferences (tokens, refs, clazz, ancestor);
```

ModelComponentBase also makes convenience wrappers for these directly available within the class. postscanAttributeName() checks if the next token in the queue is a StringToken matching name, and if it is, consumes that token and returns true. postscanReference() checks that the next token is a StringToken containing a path reference, finds the component referenced by that path relative to ancestor, checks that it is an instance of clazz, and returns it. postscanReferences() obtains a set of component references described by a sequence of StringTokens located between BEGIN and END tokens, and returns the referenced components in an array. These methods will throw an IOException if they encounter unexpected tokens or if referenced components cannot be found.

Employing these methods to handle the point0, point1 reference example above, we obtain:

```
public boolean postscanItem (
Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

   if (postscanAttributeName (tokens, "point0")) {
      myPnt0 = postscanReference (tokens, Point.class, ancestor);
      return true;
   }
   else if (postscanAttributeName (tokens, "point1")) {
      myPnt1 = postscanReference (tokens, Point.class, ancestor);
      return true;
   }
   return super.postscanItem (tokens, ancestor);
}
```

Similarly, the points reference example can be handled as:

```
public boolean postscanItem (
Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

   if (postscanAttributeName (tokens, "points")) {
      Point[] pnts = postscanReferences (tokens, Point.class, ancestor);
      return true;
   }
   return super.postscanItem (tokens, ancestor);
}
```

Finally, for composite components, it is necessary to call postscan() for each of their children. For composites implemented using ComponentListImpl, this can be done by calling the latter's postscanComponent() method. For both CompositeComponent and MutableCompositeComponent implementations, the corresponding code looks like this:

```
protected boolean postscanItem (
   Deque<ScanToken> tokens, CompositeComponent ancestor)
   throws IOException {

   if (myComponents.postscanComponent (tokens, ancestor)) {
      return true;
   }
   return super.postscanItem (tokens, ancestor);
}
```

postscanComponent() checks to see if the next token is an ObjectToken containing a ModelComponent, and if it is, it removes that token, calls the component's postscan() method, and returns true.

Post-scanning property values

As described in 3.2, base implementations of scanItem() automatically read in and set property values for the component. However, since this is done during the first scan step, problems may arise if any properties depend on references having already been set. An example of this is the theta property for the joint component RevoluteJoint, which requires knowledge of the frames referenced by the joint.

The solution to this is to defer setting the values for such properties until the post-scan step. This is done by saving the scanned property values in the token queue, and then actually setting the properties during the post-scan. Two convenience methods, ScanWriteUtils.scanAndStorePropertyValues() and ScanWriteUtils.postscanPropertyValues() allow this to be done fairly easily:

```
// properties that must be set during post-scan:
String[] deferredProps = new String[] { "theta", "thetaRange" };

public boolean scanItem (ReaderTokenizer rtok, Deque<ScanToken> tokens)
throws IOException {

if (ScanWriteUtils.scanAndStorePropertyValues (
   rtok, this, deferredProps, tokens)) {
      return true;
   }
   else {
      ... remaining scanItem() implementation ...
   }
}

public boolean postscanItem (
Deque<ScanToken> tokens, CompositeComponent ancestor) throws IOException {

   if (ScanWriteUtils.postscanPropertyValues (tokens, this, deferredProps)) {
      return true;
   }
   else {
      ... remaining postscanItem() implementation ...
   }
}
```

Here, `defernedProps` is an array of the names of the properties whose values must be set in the post-scan step (`"theta"` and `"thetaRange"` are taken from the specific `RevoluteJoint` example).

To ensure that this works properly, it is also important that the reference information be written out *before* the property information, so that in the post-scan step, it will be set before the property values. That means that `writeItems()` should be structured as follows:

```
protected void writeItems (
PrintWriter pw, NumberFormat fmt, CompositeComponent ancestor)

throws IOException {

   ... write out reference information first ...
   super.writeItems (pw, fmt, ancestor);
}
```

Invoking the complete scan process

The two-step scanning process means that for the top-level invocation of `scan`, the application needs to create a token queue, call `scan()` with this queue as the `ref` argument, and then call `postscan()`:

```
   ArrayDeque<ScanToken> tokens = new ArrayDeque<ScanToken>();
   comp.scan (rtok, tokens);
   comp.postscan (tokens, ancestor);
```

For convenience, the above code fragment is encapsulated into the method ScanWriteUtils.scanfull().

If we are scanning an entire hierarchy from scratch, then `comp` will be the root component of the hierarchy and `ancestor` will equal `comp`. Otherwise, if we are scanning a new sub-hierarchy, then `comp` will be the root of the sub-hierarchy and `ancestor` may be some component higher in the existing hierarchy.

File and token structure

Because Artisynth components are responsible for writing and scanning themselves, there is no mandatory file structure imposed per-se. However, there is a structure that should be adhered to whenever possible. Expressed loosely as a production grammar, with '*' expressing repetition of zero or more times, this is:

```
component
        '[' componentItem* ']'

componentItem
        attributePair
        componentSpec

attributePair
        NAME '=' value

componentSpec
        CLASSINFO component
        component

value
        componentSpec
        list
        literal

list
        '[' listItem* ']'

listItem
        value
        attributePair

literal
        BOOLEAN
        INTEGER
        FLOAT
        STRING
        WORD
```

Here, NAME and WORD are identifiers that consist of alphanumerics, '$', or '_', and do not begin with a digit. CLASSINFO is the classname for a component, or an alias that can be mapped to a classname using ClassAliases.resolveClass().

Within an ArtiSynth file, '#' is a comment character, causing all remaining characters on the line to be discarded.

When tokens are saved for the post-scan step, they should arranged in a structure similiar to that used for the file itself:

```
componentTokens
BEGIN itemTokens* END

itemTokens
componentSpecTokens
attributePairTokens

attributePairTokens
NAME valueTokens

componentSpecTokens
      COMPONENT componentTokens
      componentTokens

valueTokens
componentSpecTokens
listTokens
literalTokens

listTokens
BEGIN <listItemTokens>* END

listItemTokens
valueTokens
attributePairTokens
```

```
literalTokens
STRING
OBJECT
```

Here, BEGIN and END are ScanToken.BEGIN and ScanToken.END, NAME is a StringToken with an attribute name as a value, COMPONENT is a ObjectToken with a reference to the object as a value, and STRING and OBJECT are StringToken and ObjectToken, respectively.

Debugging

Debugging write and scan methods is generally not too difficult because of the ascii nature of the data files. A good first test is to write components out, read them back in, and then write them out a second time and make sure that the second output equals the first. Scan methods will generally throw IOExceptions when unexpected input is encountered, and these usually provide the offending line number.

Problems that occur in post-scan can be slightly harder to solve because the token queue is not normally written out in any place where it can be inspected. To help with this, once can use ScanWriteUtils.setTokenPrinting() to enable the token queue produced by ScanWriteUtils.scanfull() to be printed to the standard output. It is also possible to print a token queue directly using ScanWriteUtils.printTokens().

Summary

The main points concerning component writing and scanning are as follows:

1. Writing and scanning are done using the component's write(), scan(), postscan() methods. These methods usually employ writeItems(), scanItem(), and postscanItem() to handle the writing and scanning of individual attributes and child components.
2. Where possible, the structure described in Section 3.3 should be adhered to.
3. Scanning is a two-step process, involving a scan step and a post-scan step. This is to accomodate the fact that some aspects of scanning (most importantly the evaluation of references) cannot be done until the entire component hierarchy has been constructed. Information needed for the post-scan step (such as reference path names) should be stored in the token queue passed to scan() and scanItems().
4. It is usually only necessary for a component implementation to override writeItems(), scanItem(), and postscanItem(). Property values are usually written and scanned automatically by the base implementations of writeItems() and scanItem(). If a component does not contain references or non-property attributes, it may not be necessary for the implementation to override any methods at all.
5. Composite components need to call write(), scan(), and postscan() for their child components. Composites implemented using ComponentListImpl can do this using methods supplied by that class, such as writeComponents(), scanAndStoreComponent(), and postscanComponent().
6. A complete scan operation involves creating a token queue and then calling both scan() and postscan() for the top-level component. This can be done using the convenience method ScanWriteUtils.scanfull().
7. The utility class ScanWriteUtils contains a large number of methods that facilitate writing and scanning.