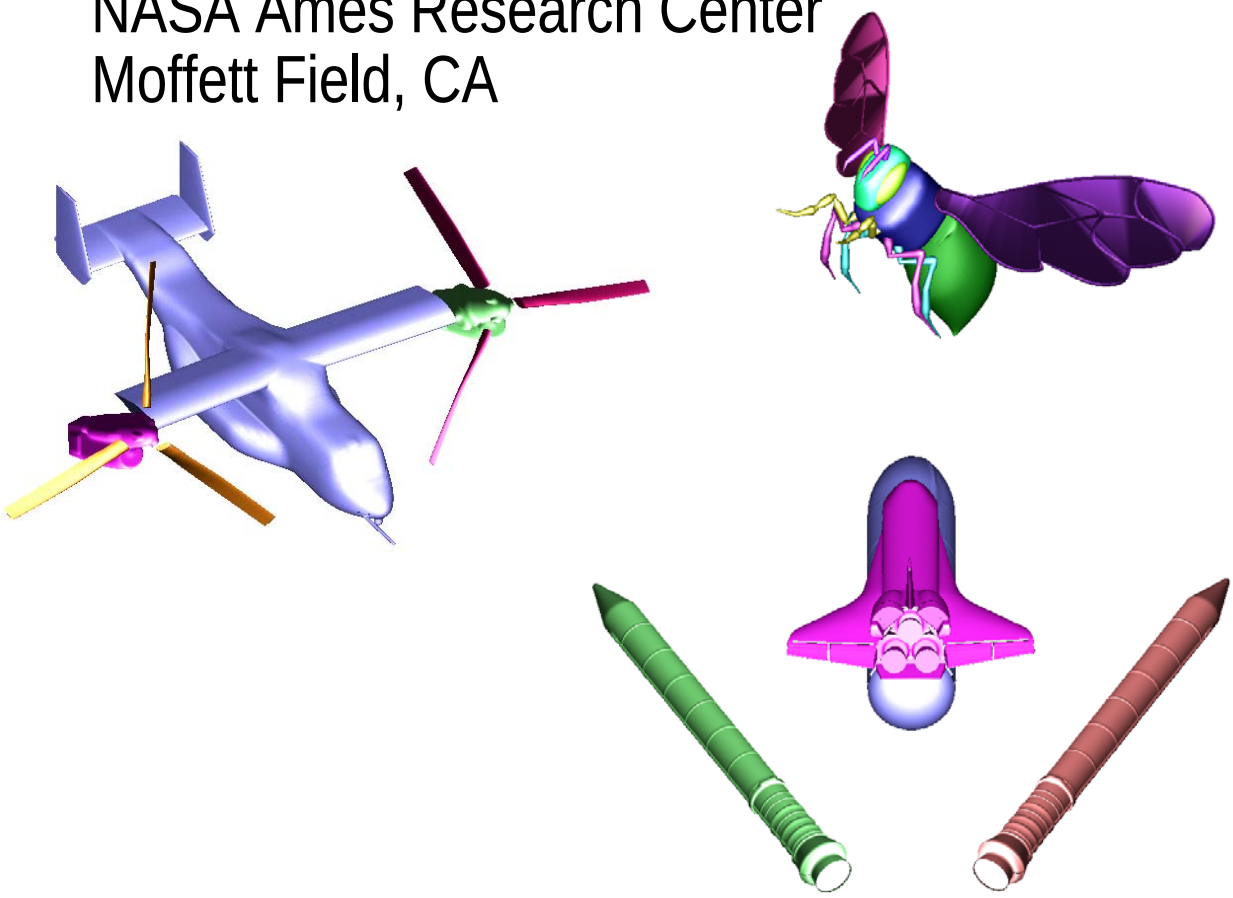
AIAA-2003-1237

An Interface for Specifying Rigid-Body Motions for CFD Applications

Scott M. Murman, William M. Chan
Michael J. Aftosmis, Robert L. Meakin
NASA Ames Research Center
Moffett Field, CA



41st AIAA Aerospace Sciences Meeting

January 6-9, 2003 / Reno, NV

AIAA-2003-1237

An Interface for Specifying Rigid-Body Motions for CFD Applications

Scott M. Murman,* William M. Chan,†
Michael J. Aftosmis,† and Robert L. Meakin‡
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

An interface for specifying rigid-body motions for CFD applications is presented. This interface provides a means of describing a component hierarchy in a geometric configuration, as well as the motion (prescribed or six-degree-of-freedom) associated with any component. The interface consists of a general set of datatypes, along with rules for their interaction, and is designed to be flexible in order to evolve as future needs dictate. The specification is currently implemented with an XML file format which is portable across platforms and applications. The motion specification is capable of describing general rigid body motions, and eliminates the need to write and compile new code within the application software for each dynamic configuration, allowing client software to automate dynamic simulations. The interface is integrated with a GUI tool which allows rigid body motions to be prescribed and verified interactively, promoting access to non-expert users. Illustrative examples, as well as the raw XML source of the file specifications, are included.

1 Introduction

It is common practice in CFD applications to compute a parameter study using static configurations. For example, a single (usually steady-state) simulation can be computed for various flap deflection angles, freestream Mach numbers, and angles of attack. In this manner a matrix of static "snapshots" of the flowfield can be easily generated, and interrogated to discern trends. This is possible, in part, because the inputs required by a CFD flow solver to perform a static simulation (those controlling the choice of scheme, timestep, etc. aside) are usually only the geometry filename (typically a character string) and freestream conditions (scalars). If some means of varying these input parameters can be devised, powerful automated tools for generating large "databases" of static simulation results can be built (cf. Refs. [1, 2]). When working with dynamic simulations however, where it is desired that the geometry move in some manner *during* the computation, a simple, yet general, means of describing the required motion is unavailable. Common methods of specifying a moving geometry for a CFD application include limiting the allowable motions, such as only providing a constant rotation rate about a Cartesian axis, or requiring the user to prescribe the motion by writing code that can be called from within the application. The former of these is not general enough for complex motions, while the latter does not lend itself to automation, and requires an "expert" user to implement. A method for describing geometric configurations and their dynamic motions which is general, can be automated, and is readily accessible to non-expert users is desired.

Towards this end, this work presents a protocol for specifying geometric hierarchies and their rigid-body motions. This protocol takes the form of a general set of datatypes and rules which can be implemented through any desired syntax, with the current choice being the Extensible Markup

*ELORET. Member AIAA
†Senior Member AIAA
‡U.S. Army AFDD (AMCOM). Senior Member AIAA

Language (XML)[3]. This low-level XML implementation is then "wrapped" with an Application Programming Interface (API). With this interface between the geometry motion and the application tools (such as the CFD flow solver), it is possible to build automated tools for performing dynamic simulations, such as would be required to compute a matrix of dynamic stability derivatives (cf. Refs. [4–6]). The specification is suitable for simple analytic prescribed motions, as well as complex N-body problems with collisions and controller feedback. A fixed specification for the geometry motion allows multiple application programs, such as visualization tools, flow solvers, and post-processing tools, to be built upon a common interface. The geometry motion can be stored in a single repository, and shared among distributed applications, which minimizes errors due to duplication. Efforts to extend the specification to include geometry states and non-rigid bodies are underway, and will be discussed at the end of this article. Illustrative examples are used throughout this article to describe the specification, and the entire XML description for these examples is included in the appendixes for reference.

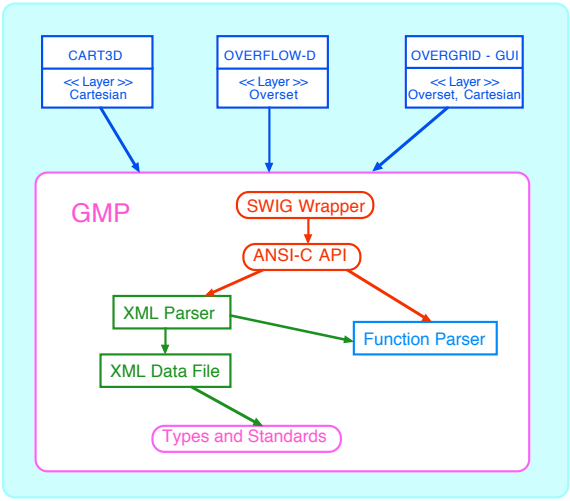


Figure 1: Schematic of GMP implementation components. The core is a set of datatypes and standards, which are implemented with an XML file specification. The XML file parser and analytic function parser provide low-level functionality. An ANSI-C API is built on top of these, and is extended using SWIG[8] to provide an interface for all common interpreted languages. High-level applications implement a customized middleware layer on top of the API provided by GMP.

2 Design Goals

The collection of datatypes and standards for the current geometry specification, along with the API, file parsers, and other auxiliary packages, is referred to as the Geometry Manipulation Protocol (GMP) (cf. Fig. 1). Reference to a protocol is inspired by Internet Protocols (IP). IPs are low-level conventions which enable data to be transferred between machines, and higher-level applications to be built upon a common standard. Similarly, GMP is a set of low-level conventions which enable geometry descriptions and manipulations to be shared and understood among various (higher-level) CFD applications and tools. Currently, the interface is implemented using an XML file format along with an analytic function parser, although the interface is not specific to XML. The function parser will be described in Sec. 4. The specification is implemented in a stand-alone library with an ANSI-C interface. This interface is extended using the Simplified Wrapper Interface Generator (SWIG)[7, 8] to support all popular scripting languages, including Perl, Python, Java, Tcl, etc. GMP is currently integrated within an inviscid Cartesian moving-body flow solver[9], the OVERFLOW-D structured, overset, viscous, dynamic flow solver[10], as well as the OVERGRID pre-processing Graphical User Interface (GUI)[11], along with several support applications.

Two of the primary goals during the development of the current specification were: **1)** that it easily allows higher-level application programs (or scripts) to modify the data for analyzing an entire parameter space of dynamic simulations, and **2)** that it also enables the use of GUI's for specifying the motion of rigid components. In order to satisfy the first item, it was decided that only a plain text (human-readable) file format could be used. There are many schemes that could be used for defining a plain text specification, however for the current application the syntax should allow variable definitions, comments, nested structures, and also the ability to insert the contents of another file (similar to the `#include` mechanism of the C programming language). XML satisfies all these criteria, and also provides several additional desirable features. By implementing the rigid-body motion specification using XML, it is possible to leverage the large amount of development work dedicated to XML in the web and database communities. Public-domain and commercial software packages exist for parsing, validating, displaying, generating

databases, and many other tasks for manipulating XML files. XML is not only portable across platforms, it also can be "understood" by many different applications, from web browsers to word processors. An example is the color-coded XML source included in the appendixes, which were generated by a web browser. One final attractive feature of XML for the current application is that the hierarchical structures that appear in many geometric configurations, and also their motions, are directly supported by the XML language.

The motion specification is intended to be simple and intuitive enough that it can be used for relatively simple motions, such as an oscillating airfoil, yet still be general enough to handle any arbitrary, complex motion. The specification allows for prescribed motions (either analytically or through a discrete table look-up), unconstrained 6-degree-of-freedom (6-DOF) rigid-body motion, as well as constrained (1-DOF, 2-DOF, ...) motion. Finally, it can describe what is referred to here as "controlled 6-DOF motion", as in a guided missile or aircraft flying under a control system. Detailed examples for describing a prescribed analytic motion and a constrained 6-DOF motion are presented in this paper.

One of the primary goals for the current development was that it support extensibility, so that it can be used in currently unanticipated roles. The major means to meet this goal was to provide a flexible structure that can be augmented in an almost arbitrary manner. The specification is independent of any application type, such as structured grid technology, or a particular CAD implementation. The API was also designed to be independent of any application. Rather than provide a complex API which attempts to be "all things for all people" (and usually fails), the API is kept very simple, and it is the responsibility of the application programmer to build the data structures, or complicated interfaces, which are apropos for their particular application. For example, the current GMP implementation is integrated within several distinct applications[9–11]. Each of these applications provides a layer of "middleware" between the interface datatypes, and the more complex (specialized) data structures used within each application code (cf. Fig. 1).

3 Typographical Conventions

As the current work is essentially a description of a set of datatypes, a consistent font system is used as an aid. All datatypes from the GMP are capitalized and displayed in sans serif font, e.g. Configuration. Most of the types have names which connote their intention, and hence are often used as a normal part of a sentence. Hierarchical datatypes are displayed with an indented list, such as

- Configuration
  - Component

where in this case Configuration is composed of lower-level Components. Types are provided with a parenthetical argument which describes whether the type is required, optional, etc. Types which are composed of base types, such as strings, scalars, etc. are shown with a bracketed argument containing the base types, for example Name [string] (required). Arguments which are typed in directly are shown in fixed-width font as in `10.0*sin(2*pi*t)`.

4 Analytic Function Parser

The interface specification relies heavily upon a generic function parser which is capable of parsing and interpreting arbitrary analytic functions. These analytic functions can take an arbitrary number of arguments. The parser understands the common mathematical operators and precedence rules, such as "(), ^,∗, /, %, +, −", common constants such as π, and most commonly used functions such as "abs, log, sin, sqrt, tanh, ...". For example, pitch rate ($\alpha(t) = 10.0\sin(2\pi t)$) for an oscillating airfoil is expressed as `10.0*sin(2*pi*t)`, and can be evaluated at run-time by providing an appropriate scalar value to substitute for the variable `t`. This substitution mechanism is provided by the function parser API. Within the GMP, an analytic function which takes no arguments replaces the role of a scalar value, i.e. a single scalar, or numeric value, is not an explicit type. In this manner, it is possible to use variables and constants which are appropriate to the problem, making the interface easier to specify. For instance, an angle of rotation can be specified as `pi/4`, as opposed to `0.7854`, and the result will be evaluated at run-time by the application using the API for the function

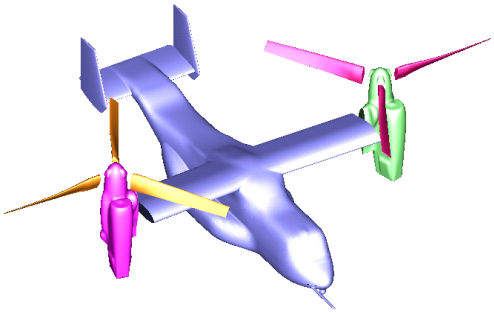
parser*. In the current specification, a nomenclature is adopted to describe the analytic function datatype, and optionally the arguments which are expected. All numeric fields are specified as an arbitrary function which takes no arguments, f(). If an analytic function datatype is expected to take an argument of time in the interface, it will be described using f(t). A vector of 3 numeric fields, such as is used to describe a position, is specified as vector: f().

5 Configuration Specification

The complete geometry which is being simulated is referred to here as a Configuration. Before a motion can be specified, it is necessary to describe the Configuration, so that a user can simply describe the motion of "the left rotor", as is intuitive, rather than being forced to refer to some application-specific geometry description. Instead of tightly coupling the Configuration information with a motion specification, the means of specifying a Configuration and the means of specifying its motion are separated. The motion specification is then built by referring to the Configuration. This allows different motions to easily refer to the same Configuration, as well as provides the ability to build separate tools which extend the Configuration.

Within GMP, the Configuration description is stored in an XML file typically named *Config.xml*. An example Configuration file for the V-22 tilt-rotor is included in Appendix A. A typical Configuration is often made up of lower-level pieces, referred to here as Components. A simplified representation of the V-22 tilt-rotor geometry is shown in Fig. 2, with the different Components highlighted by color. The V-22 is made up of many Components, such as the fuselage, wing, empennage, rotors, etc. Many of these Components can also be further broken into smaller pieces, for example the rotors can be broken down into a nacelle, hub, and blades. This suggests that the Configuration is composed of a hierarchy, or tree, of Components. One possible hierarchy structure for the V-22 is shown schematically in Fig. 2. Notice also that this hierarchy is not unique, for example the rotors might be considered as a lower-level Component of the wing, or on the same level as the wing. These different hierarchies

*It is still possible to use a simple scalar value, and it will evaluate to itself at run-time.



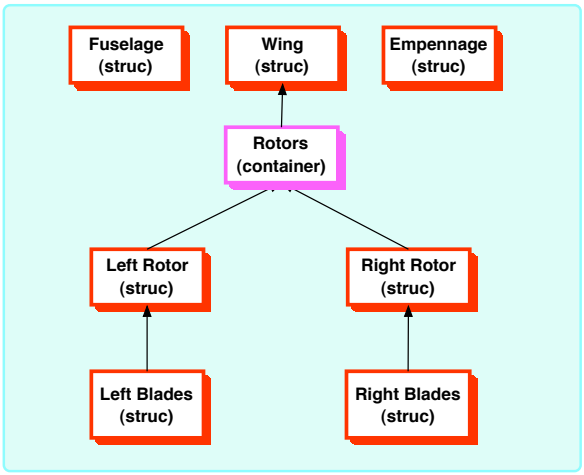


Figure 2: Example Configuration hierarchy for the V-22 tilt-rotor. Component types are specified by color and text. Solid lines represent a parent-child relationship, and dashed lines represent a source-clone relationship.

can become important when specifying the relative motion, however, as will be described in the next section.

While the abstract hierarchy description of a Configuration is helpful, at some level it must be associated with the actual geometry that is to be manipulated. This is accomplished by requiring that each Component specify its Type, and optionally include some type-dependent Data. In order to promote flexibility within the Configuration specification, each type of Component is considered equal, and can be utilized anywhere within the Configuration hierarchy. Further, the Component types form an open-ended list which can be extended by future applications as needed. In other words, it is up to the external applications to determine which type of Components they can work with and understand, not the specification, and similarly for the optional type-dependent Data. A small number of Component types have been developed in implementing

the GMP within the three application codes from Refs. [9–11]. The tree-diagram of Fig. 2 includes a label with the different Component types. These types are briefly described as

- `struc`: A set of structured grid (possibly overlapping) surface patches
- `tri`: A surface triangulation
- `container`: An agglomeration of lower-level Components
- `clone`: An duplicate of another (non-clone) Component

A Component of the Configuration hierarchy has the following complete list of attributes:

- Component:
  - Name [string] (required)
  - Type [string] (required)
  - Parent [string] (optional)
  - Data [arbitrary] (optional)
  - Source [string] (optional)
  - Transforms (optional)
    - Translate (optional)
      - Displacement [vector: f()] (required)
    - Rotate (optional)
      - Center [vector: f()] (required)
      - Axis [vector: f()] (required)
      - Angle [f()] (required)
    - Mirror $[x|y|z]$ (optional)

The Parent attribute is used to specify the tree structure of the Configuration. The motion of a Component is usually specified relative to its Parent. Root nodes of the tree have no parents (the interpretation being that the inertial reference frame is the parent for motion), and multiple root nodes are allowed. The Source tag specifies optional information, such as a filename or link, for the Component, so that the Configuration can potentially be built from a library of stored Components. The Transforms tag is used if the Component is to be translated, rotated, or mirrored into position within the Configuration, and is made up of sub-types for specifying the actual transformations. All coordinates used in the transformation are specified in the original, untransformed (natural) coordinate system of the appropriate geometry. The `clone` Type can represent an exact duplicate, although in most instances the original is copied and then Transformed to a new position. For example, the cloned Component can be Mirrored about the x, y, or $z = 0$ for Configurations with lateral symmetry. Similarly, a single turbine blade can be cloned and Rotated to form a set of blades around a hub. In this manner errors due to duplication are reduced, and a common set of methods can easily be extended to an arbitrary number of Components.

6 Motion Specification

Each motion specification refers to the Configuration description outlined in the previous section. The specific Components which are in motion are referred to by their Name attribute. The motion, or sequence of motions, is described by what is referred to as a Scenario, and is specified in an XML file named *Scenario.xml*. Scenarios are parameterized by time (t), starting at $t = 0$, with the units of time dependent upon the application. A Scenario is characterized by a number of actions, each occurring at a specific time, and for a specific duration. Currently, two types of actions can be specified; Prescribed motions, and Aero6DOF motions. These two types of motions are considered as distinct types as there is little commonality between them.

6.1 Prescribed Motion

The Prescribed motion can be specified as an arbitrary analytic function of time, or through a discrete table look-up. The analytic functions of time are parsed and evaluated by the stand-alone function parser described in Sec. 4. The time is interpreted as relative to the Start time of the current Prescribed motion, and a substitution of this current relative time is performed whenever the analytic functions are evaluated. This allows a Prescribed motion to be used multiple times within the same Scenario without modification. In order to specify the motion, either the position of a Component must be specified, or its velocity and initial position, though both position and velocity are usually needed by most CFD flow solvers. Since the initial position is available from the description of the Configuration, and it is easier to numerically integrate a function accurately than it is to differ-

entiate, the analytic motion is Prescribed by providing the translational and angular velocities over the time period. The exception to this is the table look-up mode of operation, where the flexibility to specify only the position is allowed.

Motions are usually Prescribed relative to the parent of the Component within the Configuration hierarchy, and this is the default behavior. Options are discussed with the hovering bee example in Sec. 6.3. The motion is specified in the initial coordinate system of the input geometry (after any required Transforms have been applied within the Configuration specification).

Prescribed motions have the following required and optional attributes:

- Prescribed
  - Component [string] (required)
  - Start [f()] (required)
  - Duration [f()] (optional)
  - InitialPosition (optional)
    - Translate (optional)
      - Displacement [vector: f()] (required)
    - Rotate (optional)
      - Center [vector: f()] (required)
      - Axis [vector: f()] (required)
      - Angle [f()] (required)
    - Mirror $[x|y|z]$ (optional)
  - Translate (optional)
    - Velocity [vector: f()] (required)
    - Frame [string] (optional)
  - Rotate (optional)
    - Center [vector: f()] (required)
    - Axis [vector: f()] (required)
    - Speed [f()] (required)
    - Frame [string] (optional)

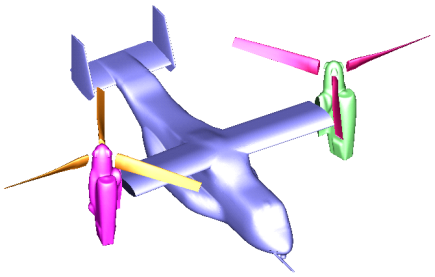
Start and Duration refer to the starting time and duration of the action. If the Duration is not specified the action is considered to be continued indefinitely. Prescribed motions are allowed to overlap in time intervals, and are ordered by their Start times. InitialPosition allows the orientation of the Components within a dynamic simulation to be transformed after a Configuration has been "built". This allows a general Configuration to be described, and then specialized if necessary for a dynamic simulation, i.e. it further decouples the Configuration and motion specification. The time level $t = 0$ is assumed to refer to the position of the body after the (optional) InitialPosition transforms have been applied. The Translate and Rotate commands specify the translational velocity of the center of mass of the component, and the rotation rate about an arbitrary axis through the center of rotation respectively. These commands are specified in the coordinates of the axis system specified by the Frame type. Choices for Frame are `body` or `parent`, with the default being `parent`. Multiple Translate and Rotate commands can be combined within a single Prescribed action, and are applied in the order they are specified within the XML file.

6.2 V-22 Example

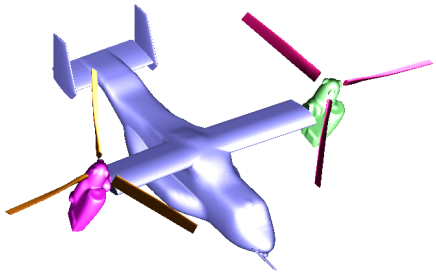
The first example Prescribed motion is a specification of the V-22 tilt-rotor where the V-22 rotors are transitioning from the vertical to horizontal positions (by rotating about a wing chord line), and the blades are continuously rotating about an axis through the rotor hub (cf. Fig. 3). The left and right sets of blades are counter-rotating. The complete Scenario specification is included in Appendix A. All of these motions are relative to their Parent in the Configuration hierarchy. Note that since the rotor blades counter-rotate, i.e. move differently relative to their parent in the Configuration hierarchy, it is not possible to simply clone one of the rotors, even though the geometries involved are simply mirror images, as this would imply that all of their sub-Components moved in exactly the same manner. The Configuration is thus both an abstract topology as well as a concrete means of manipulating geometry.

6.3 Hovering Bee Example

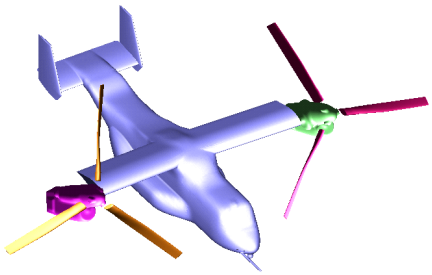
The second example Prescribed motion is of a bee flapping its wings to hover (cf. Figs. 4 and 5), and is also included in Appendix B. The Configuration for the bee is a more complex example, including cloned Components and several levels of hierarchy. The analytic formulation for the wing motion is based on the observations of Ellington[12]. As opposed to the V-22 tilt-rotor, where the compound motion is a superposition of the motion of various Components relative to their Parents, here the compound motion is of a single Component performing



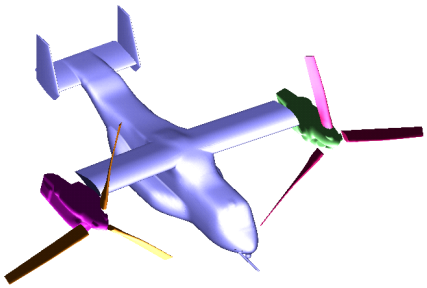
(a) time = 0.0



(b) time = 0.35



(c) time = 0.70



(d) time = 1.05

Figure 3: Snapshots of V-22 rotors transitioning from the vertical to horizontal positions, while the blades continuously rotate about the rotor hub. The GMP XML specification for this motion is included in Appendix A.

an ordered series of actions. In the V-22 example, all of the motions are specified in the parent coordinate frame, which is the default frame. When working with a complex motion of a single component, it is desirable to specify actions relative to the parent or relative to the continually moving body frame. In the flapping wing example, the motion of the wing is specified as a stroke and flapping about axes in the parent system, and a pitch about a wing span axis. Setting up this (relatively) complicated motion with the aid of the OVERGRID GUI and current motion specification infrastructure required approximately 15 min.

6.4 6-DOF Motion

Along with Prescribed motions, CFD applications often simulate 6-DOF motions where the rigid body is free to move under the influence of aerodynamic loads. With the exception of Start and Duration times, the specification of Prescribed and Aero6DOF motions have little in common, and hence are treated as separate types. A component cannot be specified as having both Prescribed and Aero6DOF motions overlapping in time. Once a Component has been specified to have an Aero6DOF motion, it is no longer considered to be a child of its Parent (if it had one), and becomes a root node, i.e. the Configuration specification becomes dynamic when Aero6DOF motions are considered.

Aero6DOF motions contain the same Name, Start, Duration, and InitialPosition types as Prescribed motions, but also contain sub-types for InertialProperties, AppliedLoads, Constraints, and Controllers. These latter are treated as sub-types of an Aero6DOF type, as opposed to types of their own, in order to make them more general. For example, if the AppliedLoad was a type then it would need to refer to the Aero6DOF motion it applied to in some manner. The AppliedLoad type would then need to be modified each time it was applied to a different Component. By making AppliedLoad a sub-type, it is implicit which Component it applies to, and it is also possible to use the same AppliedLoad with multiple Components without modification. For example, if a store ejector is modeled, this ejector can be tested with different store geometries simply by referencing the appropriate XML code within the specification. Similar arguments apply to Constraints and Controllers, as AppliedLoad. AppliedLoad, Constraints, and Controllers can be thought of as "modifiers" for the Aero6DOF

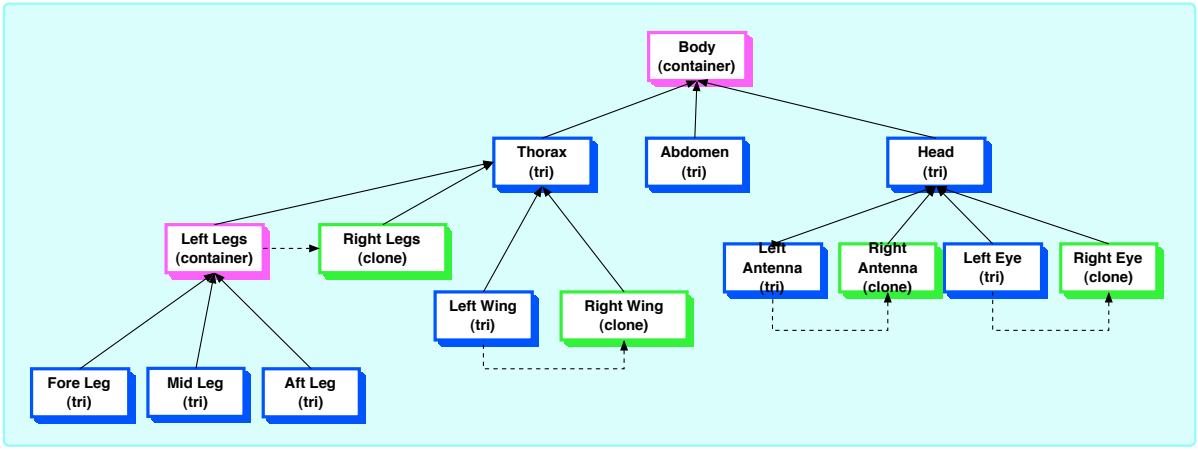
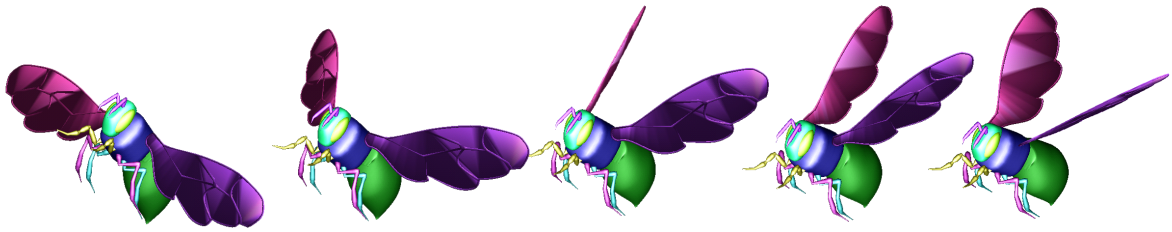


Figure 4: Configuration hierarchy for the bee in Fig. 5. Component types are specified by color and text. Solid lines represent a parent-child relationship, and dashed lines represent a source-clone relationship. The GMP XML specification for this Configuration is included in Appendix B.

(a) Downstroke (left to right)



(b) Upstroke (left to right)

Figure 5: Snapshots of a bee flapping its wings in hover. The Configuration hierarchy for this example is in Fig. 4, and the GMP XML specification for this motion is included in Appendix B.

type. In this manner it is possible to build a library of ejector models, feedback systems, etc., which can then be used within different simulations without modification.

The complete type map for an Aero6DOF motion is

- Aero6DOF
  - Component [string] (required)
  - Start [f()] (required)
  - Duration [f()] (optional)
  - InitialPosition (optional)
    - Translate (optional)
      - Displacement [vector: f()] (required)
    - Rotate (optional)
      - Center [vector: f()] (required)
      - Axis [vector: f()] (required)
      - Angle [f()] (required)
    - Mirror $[x|y|z]$ (optional)
  - InertialProperties (required)
    - Mass [f(t)] (required)
    - CenterOfMass [vector: f(t)] (required)

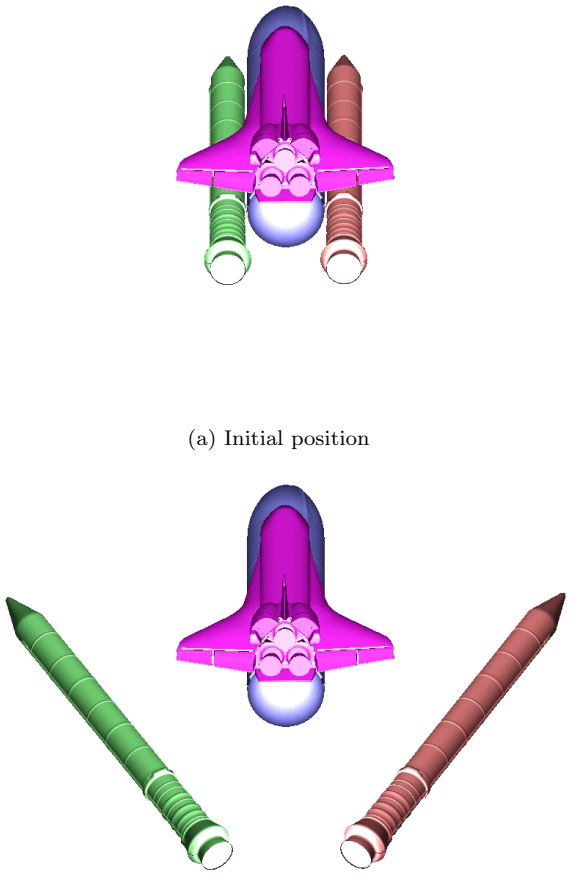
    - PrincipalMomentsOfInertia [vector: f(t)] (required)
    - PrincipalAxesOrientation (required)
      - Axis [vector: f()] (required)
      - Angle [f()] (required)
  - AppliedLoads (optional)
    - Start [f()] (required)
    - Duration[f()] (optional)
    - Frame [string] (required)
    - Force [vector: f(t)] (optional)
    - Moment [vector: f(t)] (optional)
  - Constraint (optional)
    - Start [f()] (required)
    - Duration [f()] (optional)
    - Translate [vector: f(t)] (optional)
    - Rotate [vector: f(t)] (optional)
  - Controller (optional)

The initial translational and rotational velocities are either zero if no Prescribed motions were in effect previously, or are equal to the Prescribed values. it is assumed the origin of the PrincipalAxes corresponds to the CenterOfMass location at the beginning of the Aero6DOF motion. The InertialProperties are allowed to be general functions of time, as is necessary to model a rocket burning fuel, or a satellite deploying an arm. it is the responsibility of the application to implement a suitable model for solving the 6-DOF equations under these conditions.

The AppliedLoads can be specified in 3 different coordinate frames; `body`, `parent`, and `inertial`. An example of a parent frame would be a pylon ejector force for a store separation. A constant thrust could be modeled using an AppliedLoad in the body frame. Constraints on the other hand are always assumed to be relative to the parent frame. The Constraint is specified as either a Translate constraint, Rotate constraint, or both. The numerical inputs are bounded by 0 and 1, with 1 corresponding to unconstrained motion and 0 for no allowed motion relative to the parent system. The three components of the Constraint vector are the x, y, z components of translation or rotation. Arbitrary functions of time can be specified for the Constraints and AppliedLoads. Controller types are specified as modifiers to the Aero6DOF motion, however they are currently left vague until more experience is gained with controlled, 6-DOF motions.

6.5 Space Shuttle Example

An example specification for an Aero6DOF motion is the Space Shuttle ejecting its two Solid Rocket Boosters (SRB) after burnout (cf. Fig. 6). The SRBs are free to move under the influence of aerodynamic forces, and an additional external ejector force is applied over the first time unit.



(a) Initial position

(b) Later time level

Figure 6: Snapshots of Space Shuttle SRBs releasing after burnout. The GMP XML specification for this Configuration and its motion is included in Appendix C.

7 Implementation

The current work specifies a set of datatypes and rules for their interaction, without enforcing any particular implementation model. The implementation is left to the particular applications, as discussed in Sec. 2. In fact, it is assumed that the applications will only implement a subset of the

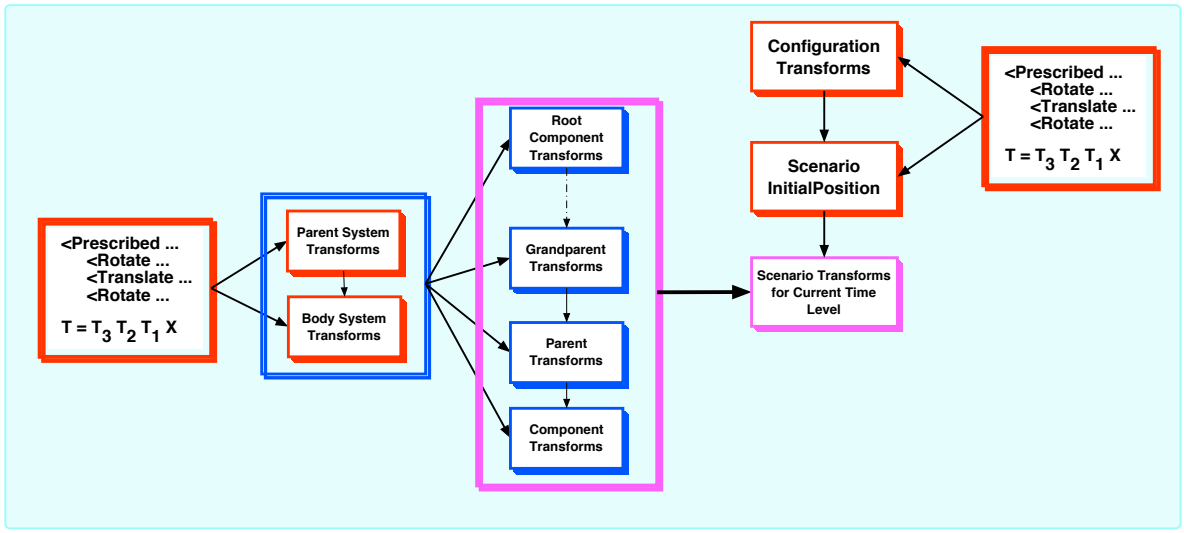


Figure 7: Required transformations to place a moving Component at current time level. The transforms are applied from the top down each vertical arrow. Horizontal arrows represent transformations that are composed of multiple parts. Each Prescribed action is an accumulation of the individual commands, in both the parent and body systems. The similar transformations from each Component in the hierarchy is applied to each of its children. This is done after each Component has been initialized in the Configuration, and placed in its InitialPosition by the Scenario.

specification. For example, the middleware for the Cartesian package[9] is customized for unstructured triangulated surfaces, while the overset solver[10] is customized for structured surface patches.* Further, applications may choose to ignore complicated or seldom-used features. For example, implementing the clone Component type adds a layer of complexity for the implementation and may not be necessary for all environments. Similarly, implementing a 6-DOF model which can handle variable mass systems may not be necessary, etc. These decisions are left to the application environment.

A discussion of some features of the implementation used in [9–11] is presented in order to provide further understanding. One basic requirement of any implementation is the ability to easily transform between the body and inertial coordinate systems. The aforementioned applications use homogeneous transformation matrices (cf. van Arsdale[13]) to represent the transforms, which are capable of uniformly representing translations, rotations, mirroring, dilation, etc. The net effect of any Transform or Prescribed command is then a cumulative matrix product of the individual transformations, applied in order (cf. Fig. 7). When applying the Prescribed commands, a further step is necessary in order to account for the motion of the Configuration hierarchy, as any transformation of the Parent component affects the position of the child. The Prescribed command processing is handled in two stages. First, a homogeneous transformation matrix is constructed by considering each Component in isolation, then the matrices from the Configuration hierarchy are applied by traversing the Configuration tree from top to bottom. The transformations required to place a body in position for a Prescribed command at an arbitrary time level are shown schematically Fig. 7.

*Both CFD solvers can understand the same motion Scenarios however, as long as the Configurations are similar.

8 Summary and Future Work

The GMP package implements a low-level specification for describing geometric configurations and their arbitrary rigid-body motions. Higher-level applications, such as visualization tools, automated post-processing environments, and CFD flow solvers, are built on top of the low-level protocol. The specification is intended for either interactive use through a GUI, or modification by application control scripts as part of an automated process. The protocol reduces the information required for describing and manipulating geometry to an XML file which is portable between different operating systems and different application programs.

This single repository for the specification reduces errors due to duplication, and also provides a self-documenting capability.

As more experience is gained with the GMP specification it will continue to evolve. The flexibility to handle this evolution has been incorporated into the specification wherever possible. The stand-alone Configuration specification has many potential uses beyond providing a means to specify rigid-body motions. Some application areas which are currently in development include: integrating the Configuration specification with post-processing tools for calculating integrated forces and moments, providing a means for specifying a Configuration "space" (ConfigSpace) for use when generating a matrix of static simulations with different geometric settings, and extending the Configuration to include non-rigid bodies. Deformable bodies are required in order to morph geometry within a geometric optimization package or perform aeroelastic simulations. These types of low-level descriptions are required in order to build reliable automated tools for CFD simulations.

Within the current specification, the Controller modifier for Aero6DOF motions has been left intentionally sparse until more experience is gained with controlled simulations. One outstanding item is the ability to repeatedly execute a command, or series of commands, optionally in a loop. This ability is currently being added and tested within the specification by generalizing the Start time, and will be supported in the future.

References

[1] Yarrow, M., McCann, K.M., DeVivo, A. and Tejnil, E., "Production-Level Distributed Parametric Study Capabilities for the Grid," in *Proceedings of Grid 2001 2nd International Conference on Grid Computing*, 2001.

[2] Murman, S.M., Chaderjian, N.M., and Pandya, S. A., "Automation of a Navier-Stokes S&C Database Generation for the Harrier in Ground Effect," AIAA Paper 2002-0259, Jan. 2002.

[3] Elliotte Rusty Harold and W. Scott Means, *XML in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Inc., 2001.

[4] Park, M. A. and Green, L. L., "Steady-state Computation of Constant Rotational Rate Dynamic Stability Derivatives," AIAA Paper 2000-4321, June 2000.

[5] Oktay, E. and Akay, H. U., "CFD Predicitiions of Dynamic Derivatives for Missiles," AIAA Paper 2002-0276, Jan. 2002.

[6] Murman, S.M., Aftosmis, M.J., and Berger, M.J., "Numerical Simulation of Rolling-Airframes Using a Multi-Level Cartesian Method," AIAA Paper 2002-2798, June 2002.

[7] Beazley, D.M., "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pp. 129–139, 1996.

[8] "Simplified Wrapper and Interface Generator." http://www.swig.org.

[9] Murman, S.M., Aftosmis, M.J., and Berger, M.J., "Implicit Approaches for Moving Boundaries in a 3-D Cartesian Method," AIAA Paper 2003-1119, Jan. 2003.

[10] Chan, W., Meakin, R., and Potsdam, M., "CHSSI Software for Geometrically Complex Unsteady Aerodynamic Applications," AIAA Paper 2001-0539, Jan. 2001.

[11] Chan, W. M., "The OVERGRID Interface for Computational Simulations on Overset Grids," AIAA Paper 2002-3188, June 2002.

[12] Ellington, C.P., "The Aerodynamics of Hovering Insect Flight. III. Kinematics," *Phil. Transactions of the Royal Society of London B*, 305:41–78, 1984.

[13] van Arsdale, D., "Homogeneous Transformation Matrices for Computer Graphics," *Computers & Graphics*, 18(2):177–191, 1994.

Appendix

A V-22 Example Specification

This specifies a possible Component hierarchy for the V-22 tilt-rotor shown in Fig. 3.

```
<?xml version='1.0' encoding='utf-8'?>
<Configuration>

   <Component Name="Starboard Nacelle" Parent="Nacelles"
              Type="struc">
      <Data> Grid List=82-97 </Data>
   </Component>

   <Component Name="Port Nacelle" Parent="Nacelles" Type="struc">
      <Data> Grid List=66-81 </Data>
   </Component>

   <Component Name="Nacelles" Parent="Main Body" Type="container"/>

   <Component Name="Main Body" Type="struc">
      <Data> Grid List=1-65 </Data>
   </Component>

   <Component Name="Starboard Blades" Parent="Starboard Nacelle"
              Type="struc">
      <Data> Grid List=107-115 </Data>
   </Component>

   <Component Name="Port Blades" Parent="Port Nacelle"
               Type="struc">
      <Data> Grid List=98-106 </Data>
   </Component>

</Configuration>
```

This specifies the motion of the V-22 tilt-rotor Configuration show in Fig. 3. The rotors transition from the vertical to horizontal positions, and the blades are continuously rotating about the rotor hub. The left and right sets of blades are counter-rotating. Figure 3 contains snapshots of the motion at 4 instances during the transition of the rotors.

```
<?xml version='1.0' encoding='utf-8'?>
<Scenario>

<Prescribed Component="Nacelles" Start="0" Duration="1" >
   <Rotate Center="0.86775, 0, 0.3742" Axis="0, -1, 0"
           Speed=" 0.5*pi" />
</Prescribed>

<Prescribed Component="Starboard Blades" Start="0" >
   <Rotate Center="0.903614, 0.602761, 0.662562" Axis="0, 0, 1"
           Speed="2.0 * pi" />
</Prescribed>

<Prescribed Component="Port Blades" Start="0" >
   <Rotate Center="0.903614, -0.602761, 0.662562" Axis="0, 0, -1"
           Speed="2.0 * pi" />
</Prescribed>

</Scenario>
```

B Hovering Bee Example Specification

The following is a Configuration specification for the bee geometry shown in Fig. 4 expressed in XML syntax:

```
<?xml version='1.0' encoding='utf-8'?>

<Configuration Source="bee5.a.tri" AngleUnit="radian">

   <Component Name="Left Wing" Parent="Thorax" Type="tri">
      <Data> Face Label=9, 10 </Data>
   </Component>

   <Component Name="Right Wing" Parent="Thorax" Type="clone">
      <Transform>
         <Mirror Plane="y"/>
      </Transform>
      <Data> Original = "Left Wing" </Data>
   </Component>

   <Component Name="Left Legs" Parent="Legs" Type="tri">
      <Data> Face Label=3, 4, 5 </Data>
   </Component>

   <Component Name="Right Legs" Parent="Legs" Type="clone">
      <Transform>
         <Mirror Plane="y"/>
      </Transform>
      <Data> Original = "Left Legs" </Data>
   </Component>

   <Component Name="Legs" Parent="Thorax" Type="container">
   </Component>

   <Component Name="Left Antennae" Parent="Head" Type="tri">
      <Data> Face Label=2 </Data>
   </Component>

   <Component Name="Right Antennae" Parent="Head" Type="clone">
      <Transform>
         <Mirror Plane="y"/>
      </Transform>
      <Data> Original = "Left Antennae" </Data>
   </Component>

   <Component Name="Left Eye" Parent="Head" Type="tri">
      <Data> Face Label=7 </Data>
   </Component>

   <Component Name="Right Eye" Parent="Head" Type="clone">
      <Transform>
         <Mirror Plane="y"/>
      </Transform>
      <Data> Original = "Left Eye" </Data>
   </Component>

   <Component Name="Head" Parent="Body" Type="tri">
      <Data> Face Label=6 </Data>
   </Component>

   <Component Name="Thorax" Parent="Body" Type="tri">
      <Data> Face Label=1 </Data>
   </Component>

   <Component Name="Abdomen" Parent="Body" Type="tri">
      <Data> Face Label=8 </Data>
   </Component>

   <Component Name="Body" Parent="Bug" Type="container">
   </Component>

   <Component Name="Bug" Type="container">
   </Component>

</Configuration>
```

The following is a motion specification for the hovering bee shown in Fig. 5 expressed in XML syntax. The compound motion of the wings is an ordered series of rotations; two about body axes and one about a wing chord line.

```
<?xml version='1.0' encoding='utf-8'?>

<Scenario AngleUnit="radian">

<Prescribed Component="Left Wing" Start="0.25" Duration="0.50" >
   <Rotate Center="0, -48, 13.3" Axis="0, -0.999343, 0.0362456"
           Speed="-pi" Frame="body"/>
</Prescribed>

<Prescribed Component="Left Wing" Start="1.1" Duration="0.50" >
   <Rotate Center="0, -48, 13.3" Axis="0, -0.999343, 0.0362456"
           Speed="pi" Frame="body"/>
</Prescribed>

<Prescribed Component="Left Wing" Start="0" >
   <Rotate Center="0, -47, 0" Axis="0, 0, -1"
           Speed="pi/3.25 * pi *cos(pi*t + asin(pi/10.0))"/>
   <Rotate Center="0, 0, 0" Axis="1, 0, 0"
           Speed="0.25*pi*cos(pi*t)"/>
</Prescribed>

<Prescribed Component="Right Wing" Start="0.25" Duration="0.50" >
   <Rotate Center="0, 48, 13.3" Axis="0, 0.999343, 0.0362456"
           Speed="pi" Frame="body" />
</Prescribed>

<Prescribed Component="Right Wing" Start="1.1" Duration="0.50" >
   <Rotate Center="0, 48, 13.3" Axis="0, 0.999343, 0.0362456"
           Speed="-pi" Frame="body"/>
</Prescribed>

<Prescribed Component="Right Wing" Start="0" >
   <Rotate Center="0, 47, 0" Axis="0, 0, 1"
           Speed="pi/3.25 * pi *cos(pi*t + asin(pi/10.0))"/>
   <Rotate Center="0, 0, 0" Axis="1, 0, 0"
           Speed="-0.25*pi*cos(pi*t)"/>
</Prescribed>

<Prescribed Component="Body" Start="0" >
   <Rotate Center="0, 0, 0" Axis="0, 1, 0"
           Speed="pi*(pi/24.0)*cos(pi*t)" />
</Prescribed>

<Prescribed Component="Legs" Start="0" >
   <Rotate Center="0, 0, 0" Axis="0, 1, 0"
           Speed="pi*(pi/36.0)*cos(pi*t)" />
</Prescribed>

</Scenario>
```

C Space Shuttle Example Specification

The following is a Configuration specification for the Space Shuttle shown in Fig. 6 expressed in XML syntax.

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE Configuration SYSTEM "Config.dtd">

<Configuration AngleUnit="radian">

   <Component Name="Orbiter" Type="struc">
      <Data> Grid List=6,32,47-57,59-61,63-86,90-92,94,96-105 </Data>
   </Component>

   <Component Name="Left SRB" Parent="External Tank" Type="struc">
      <Data> Grid List=16-21,107 </Data>
   </Component>

   <Component Name="Right SRB" Parent="External Tank" Type="struc">
      <Data> Grid List=22-27,106 </Data>
   </Component>

   <Component Name="External Tank" Parent="Orbiter" Type="struc">
      <Data> Grid List=1-5,7-15,28-31,33-46 </Data>
   </Component>

</Configuration>
```

The following is the Aero6DOF Scenario for the Space Shuttle SRB release expressed in XML syntax:

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE Scenario SYSTEM "Scenario.dtd">

<Scenario Name="Booster Separation" Gravity="0.0, 0.0, -32.2" AngleUnit="DEG">

<Aero6dof Component="Left SRB"
          Start="0.0">

<!-- the center for the principal axes is implicitly the c.m. -->
   <InertialProperties
          Mass="150E3"
          CenterOfMass="0.4, -0.25, 1.642"
          PrincipalMomentsOfInertia="422, 40711, 40711">
          <PrincipalAxesOrientation Axis="0.0, 0.0, 1.0" Angle="0.0"/>
   </InertialProperties>

   <AppliedLoad Start="0.0" Duration="1.0" Frame="parent" Force="0.0, -200E3, 0.0" />

</Aero6dof>

<Aero6dof Component="Right SRB"
          Start="0.0">

<!-- the center for the principal axes is implicitly the c.m. -->
   <InertialProperties
          Mass="150E3"
          CenterOfMass="0.4, 0.25, 1.642"
          PrincipalMomentsOfInertia="422, 40711, 40711">
          <PrincipalAxesOrientation Axis="0.0, 0.0, 1.0" Angle="0.0"/>
   </InertialProperties>

   <AppliedLoad Start="0.0" Duration="1.0" Frame="parent" Force="0.0, 200E3, 0.0" />

</Aero6dof>

</Scenario>
```