View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

A GENERIC MULTIBODY SIMULATION

by

K. A. Hopping, Principal Engineer
W. Kohn, Advanced Systems Engineering Specialist

Lockheed Engineering and Management Services Company, Inc.
Houston, Texas

ABSTRACT

This paper describes a dynamic simulation package which can be configured for orbital test scenarios involving multiple bodies. The rotational and translational state integration methods are selectable for each individual body and may be changed during a run if necessary. Characteristics of the bodies are determined by assigning components consisting of mass properties, forces, and moments, which are the outputs of user-defined environment models. Generic model implementation is facilitated by a transformation processor which performs coordinate frame conversions. Transformations are defined in the initialization file as part of the simulation configuration. The simulation package includes an initialization processor, which consists of a command line preprocessor, a general purpose grammar, and a syntax scanner. These permit specification of the bodies, their interrelationships, and their initial states in a format that is not dependent upon a particular test scenario.

Keywords: generic simulation, multiple bodies, state propagation, initialization processor, data structure, data pointer, coordinate transformation

RECEIVED
A.I.A.A.

LEMSCO-23365

(NASA-CR-182949) A GENERIC MULTIBODY SIMULATION (Lockheed Engineering and Management Services Co.) 27 p CSCL 09B

N88-24198

Unclas
G3/61 0146670

1. INTRODUCTION

In general, simulations are designed and built to fixed specifications. With the advancement of the space program, there will be multiple and varied requirements for simulation. Each simulation will have a specific purpose, such as traffic control around the Space Station, control system analysis of auxiliary vehicles, systems evaluation for the Space Station, and construction and manufacturing in space. Each of these applications will have a unique set of requirements to be met by simulation, but common to all will be a requirement to simulate the physical phenomena of multiple bodies in Earth orbit and all their interactions.

This paper describes a generic multibody simulation that allows the user to construct a test scenario consisting of the specifications for a number of bodies and their interrelationships. The simulation is designed to handle different test scenarios without major software revisions. An initialization processor builds the required simulation data structures based on the user's initialization file. Changes to the test scenario, such as the addition of another body, affect the initialization file rather than the simulation code itself.

The primary task of the simulation is to propagate the translational and rotational states for each body. A key observation is that for orbital motion all bodies obey the same physical laws (Newton-Euler). Therefore, up to parameters, the software required for the characterization of the dynamic behavior of all bodies is the same and is based on recursive integration of the corresponding equations of motion.

The central data structure of the simulation is the free body, which is composed of the following elements.

1. State
2. Mass properties
3. Forces and moments

These elements are themselves composite data structures. For example, the state includes translation (position and velocity) variables, rotation (angular position and velocity) variables, and the integration method (rectangular, trapezoidal, etc.) used

for updating the variables. A detailed description of free body state propagation is given in section 4.

While the characteristics of each body are reflected in the free body data structure, the interrelationships between bodies are represented by constructing links between the different free body data structures. These links control the flow of computation in the simulation and are important for maintaining logically consistent state data. A discussion of the free body hierarchy is given in section 3.

From a purely computational point of view, all bodies are identical. However, some means must be provided to incorporate their different physical characteristics into the calculations. Section 5 describes the approach used to associate different mass properties, forces, and moments with each free body.

One of the most difficult problems in the construction of a generic simulation is the need to deal with scenario-dependent coordinate transformations. Simulation math models are often concerned with relationships between bodies. Since the initialization file contains the information specifying which bodies are in a given test scenario, the definition of interbody coordinate transformations must also be given there. Section 6 contains a discussion of generic coordinate transformations and how they fit into the flow of computation.

The advantages of a generic simulation are lost if the initialization calculations have to be hard coded for each scenario. One approach to circumventing this input bottleneck is the use of a general purpose syntax scanner and a flexible grammar for describing the initial state data. A discussion of these features is given in section 7, and an example is presented in section 8.

The simulation is coded in the C programming language. The choice of C language was made because of the capability for defining compound data structures and the ease of generating and manipulating data pointers. In most cases, the generic capability of the simulation is achieved by converting the specifications in the test scenario into some type of linked data structure. Simulation calculations can then be executed as recursive function calls independent of the particular test scenario.

In many respects, the generic simulation bears a closer resemblance to an operating system than it does to the traditional simulation architecture.

2. SIMULATION DESIGN

A block diagram showing the general structure of the simulation is given in figure 1. The simulation is composed of five functional units: initialization; free body state propagation; environment, sensor, and controller models; output interface; and input interface.

The initialization unit reads the user's initialization file and configures the free bodies and math models according to the test scenario. A simulation update cycle begins by executing the environment models followed by integration of the free body equations of motion. The sensor models are then executed using the new state data. The controller models process the sensor data to generate inputs to the environment models. Finally, the output and input buffers are processed to complete the cycle.

The primary design goal was to provide a simulation architecture that could be adapted to the needs of several different projects. The first step in accomplishing this was to separate free body state propagation, which embodies invariant laws of physics, from the application-specific math models. Free bodies are treated as data objects which interact with the environment models by means of well-defined function calls. All free bodies have the same basic properties, and they may be reproduced in as many copies as required by the test scenario.

An object-oriented approach was also applied to the math models. Each model interacts with the overall simulation by means of the following operations.

1. Reset – Clear the internal state to the default value
2. Initialize – Read the initialization file to set the internal state
3. Update – Process the model inputs to the update state and generate new model outputs
4. Print – Output the model state to the print file
5. Input – Get the model inputs from the input buffer
6. Output – Put the model outputs into the output buffer

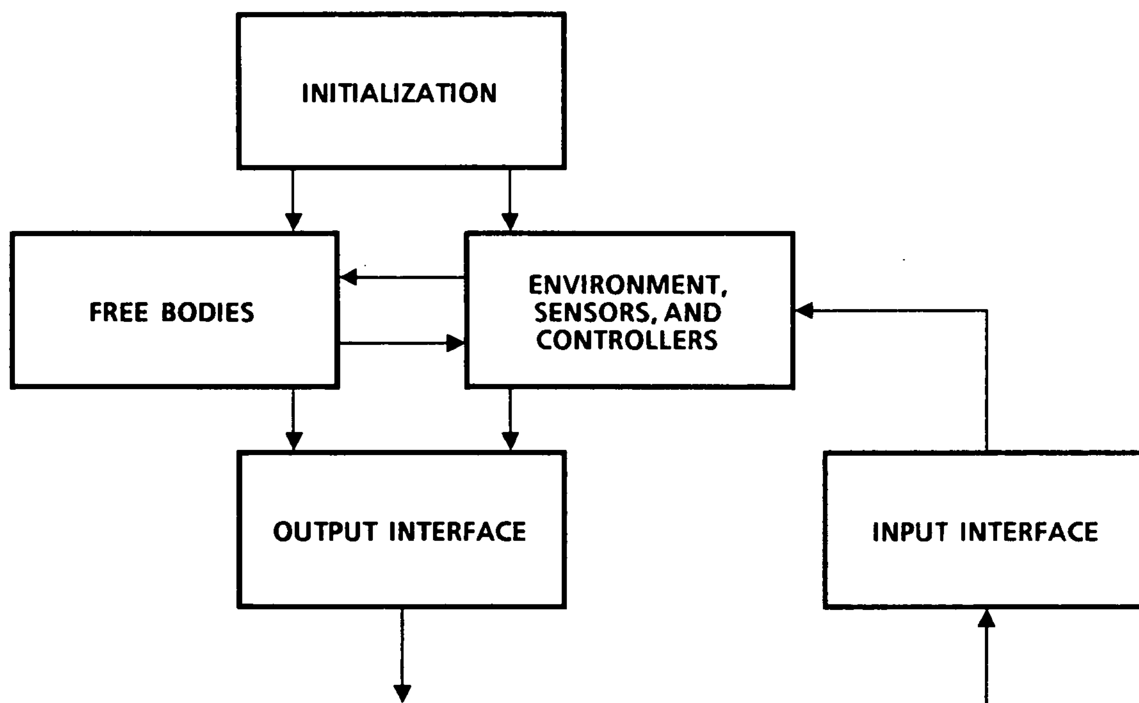

Figure 1.- Simulation block diagram.

The simulation executive executes these model functions during the operational cycle without regard to the internal workings of a particular model. Additional functions may be provided for interaction between models. However, these operations are not visible at the executive level.

The math models can only distinguish among themselves and the various free bodies by means of names assigned in the initialization file. Thus, model implementations are forced to be generic with respect to input sources and output destinations. The model interfaces are maintained via data pointers that are part of the internal model state. The general data flow during simulation execution does not require dedicated action on the part of the executive program. It is the connections between models specified in the initialization file that determines what processing will take place. The role of the simulation executive is simply to execute the different model classes (environment, sensors, and controllers) in proper relation to free body state propagation and to process the transmission and receipt of output and input buffers.

The initialization file is far more than a simple table of values for specifying model parameters. It directs construction of the entire simulation configuration by linking together a library of math models. The function calls which accomplish this might also be used during execution to dynamically reconfigure the simulation.

3. FREE BODY HIERARCHY

In a multibody simulation, one of the central problems is to determine the proper sequence for updating the state of each body so that uniformity in time of the relative interbody dynamics is achieved. Since the update sequence is scenario-dependent, it is important to provide a general method for describing the interbody relationships to the simulation.

The example in figure 2 shows a mission scenario involving a Space Shuttle, a manned maneuvering unit, the Space Station, and a satellite in low Earth orbit. The hierarchical relationships are represented by links between bodies in the graph. One method for representing such relationships in a computer data structure is the binary tree. Figure 3 illustrates how the mission scenario is represented as a binary tree.

The binary tree is easily constructed. As each body in the scenario is defined and its parent body specified, it can be linked at the proper level in the tree. Bodies which have no parent are simply linked to the root level of the tree. The resulting data structure contains all the information pertaining to parent-child relationships. Because the structure is open-ended, there are no constraints on the number of bodies in the test scenario or on how deeply nested their interrelationships may be.

The procedure for updating the free body states now becomes a matter of traversing the binary tree. This is handled by recursively processing down from the root level until a body with no children is reached. After that body is updated, the same procedure is applied iteratively to each of its siblings until none remain on the current level. The recursion then backs up to process the previous level, proceeding in this manner until the root level is completed and all bodies have been updated.

Note that the procedure described is independent of any particular test scenario. The only input data required is the root link of the free body hierarchy. The test scenario is implicit in the structure of the binary tree. Also, there is no constraint that the hierarchy be static. Bodies can be introduced or deleted from the simulation at any time by simply linking them to or unlinking them from the binary tree.

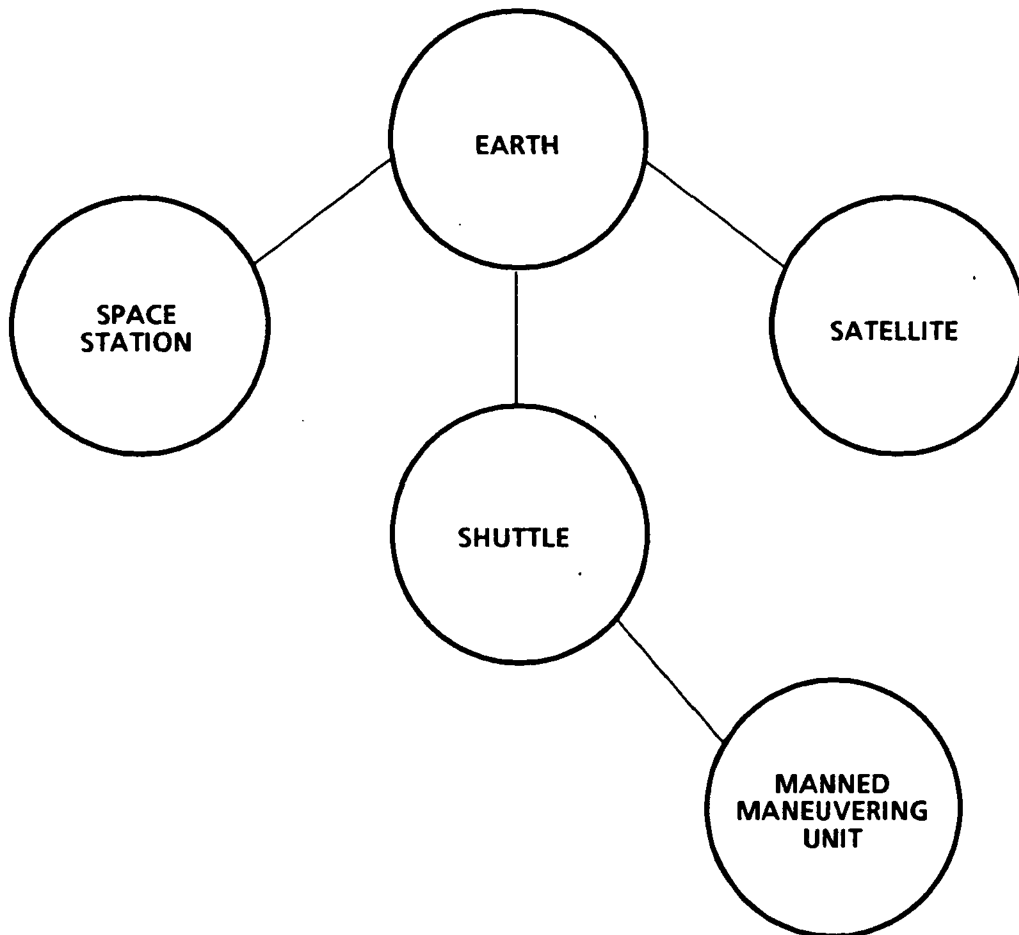

Figure 2.- Test scenario.

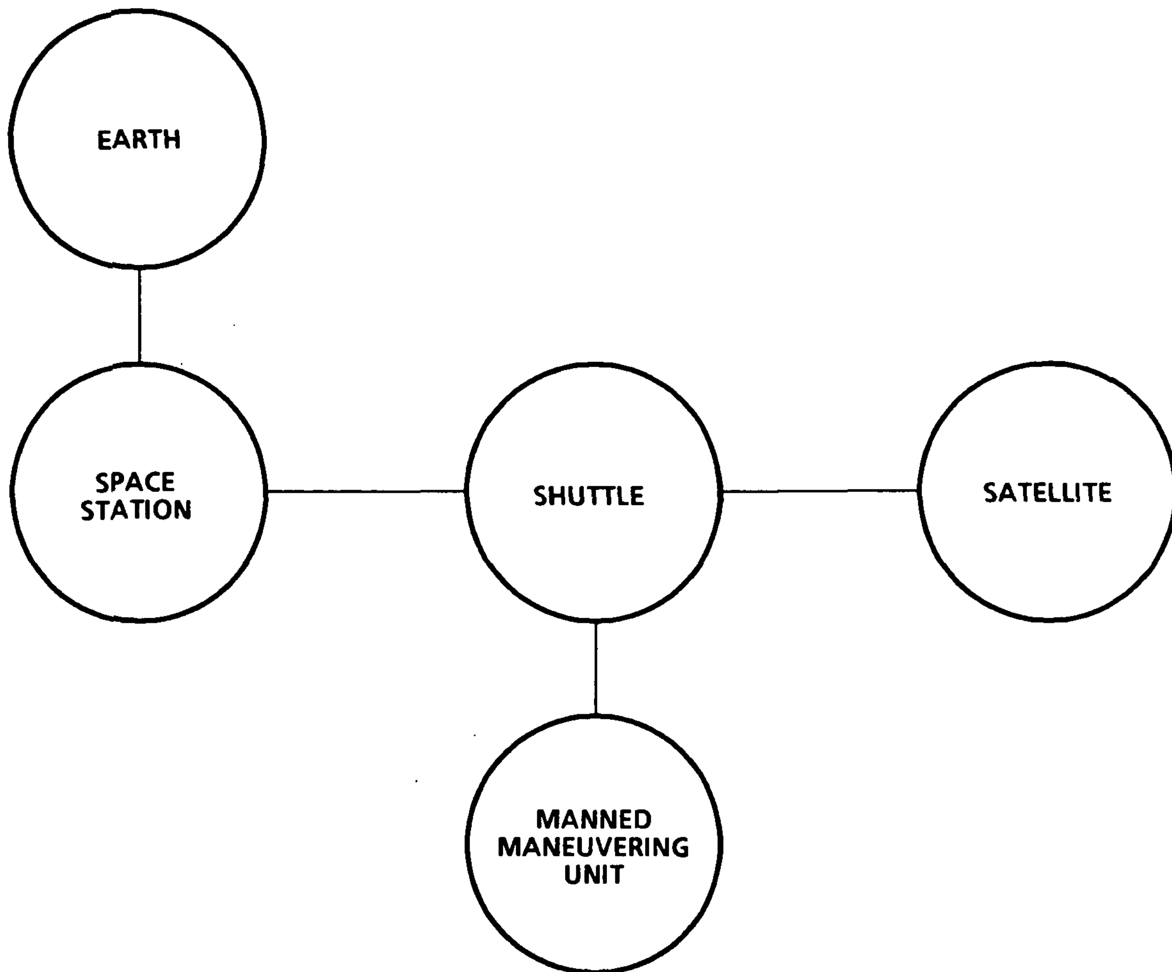

Figure 3.- Binary tree representation.

Thus, this approach is well-suited for dynamic test scenarios, where new bodies may be created by vehicles docking or undocking from each other.

4. STATE PROPAGATION

Each free body has an associated state vector that consists of a translational state and a rotational state. The translational state is initialized from a known position and velocity. The rotational state is initialized from a set of Euler angles and angular rates. Any of the six canonical Euler sequences may be used. All functions that accept Euler angles as arguments are required to test for the particular sequence and process accordingly.

The translational state is defined in a user-specified, rectangular inertial reference frame. According to Newton's law, the total external force divided by the body mass yields the nongravitational acceleration. The gravitational acceleration is calculated from the current position by means of a gravitational potential model. The total acceleration is then integrated to obtain an updated velocity and position. The integration method is selectable at initialization time. Three methods have been implemented: rectangular, trapezoidal, and Cowell's methods.

The rotational state is defined in a rectangular coordinate frame located at the body's center of gravity. The Euler angles are defined with respect to a user-specified, nonrotating reference frame. The total external moment and the inertia dyad are used to calculate the angular acceleration. This acceleration is integrated to obtain the angular rate. The angular rate is used to integrate the rotational quaternion, from which the updated Euler angles are extracted. The integration method for the rotational state is selectable independently from the translational method, and the same methods are available for it.

The equations of motion default to particle dynamics for the case of a body with zero-mass properties. The linear acceleration is set to zero, and only gravity acts to determine the translational state propagation. The angular velocity is fixed at a constant value for rotational state propagation. It is also possible to totally bypass either translational or rotational state propagation by specifying a null reference frame for the associated state. In fact, this is the default condition if no value is given in the initialization file.

Multistep integration methods require that previous values of the state vector be retained. To accommodate this, the state vector data structure includes several copies of the translational and rotational data structures. A circular queue of pointers to these multiple states is maintained by the integration algorithm. As each integration step is completed, the queue is rotated forward by one position so that the most recently computed state becomes the current state. The previous states shift back, with the oldest state being overwritten during the next update cycle. This technique avoids the large number of load/store operations usually associated with retaining of old state data. It is also independent of the implementation details of the translational and rotational states.

A complete set of previous states is maintained regardless of which integration method is currently selected. This makes it possible to switch integration methods while a run is in progress. For example, a rendezvous test scenario might use a low order integration method during early approach phases and then switch to a more accurate high order method for close-in docking maneuvers.

Initialization is more difficult for multistep integration methods because of the need to generate previous state values. The approach taken to overcome this problem is to use a first order method to integrate backwards with a negative time step. This provides satisfactory results if no large transient disturbances are present in the initial state.

5. FREE BODY COMPONENTS

In addition to a state vector, each free body has three other components: mass properties, forces, and moments. These are data structures that define the interface whereby environment models can influence the propagation of the state vector. The environment models associated with a free body will depend on the simulation configuration specified in the initialization file. Some mechanism must be provided to accommodate these configuration dependencies while still maintaining a generic approach to state propagation.

A means for achieving this goal is illustrated by the diagram in figure 4. The free body components are maintained as linked chains. The root component of each chain is reserved as a summation element for the total mass properties, total force, and total moment that are required for state propagation. The summation procedure, which is quite simple, must be carried out prior to the integration of the state equations. After the summation element is zeroed, each element in the chain is added to the sum until a null link indicates the end of the chain. In the case where the first link is null, the summation element is treated as a constant. This is useful for bodies which have a single invariant component.

Since state propagation only interfaces with the summation element, it is effectively isolated from the simulation configuration which is represented in the component chains. Each environment model has the responsibility for linking its output data to the appropriate free body component chain during the initialization processing. Thus, the interface between environment models and free bodies is controlled by the functions which build the linked chains. There is no requirement for the chains to remain static; thus, components can be introduced or deleted dynamically by simply modifying the component chains.

Each of the free body components is implemented as a simulation resource. Functions are provided for allocating, linking, unlinking, and deallocating components. These routines are designed to carefully validate the component data types so that extraneous data cannot be introduced into the component chains. The resource management functions are generic with respect to the different component types. Only the operations for zeroing and addition of two components

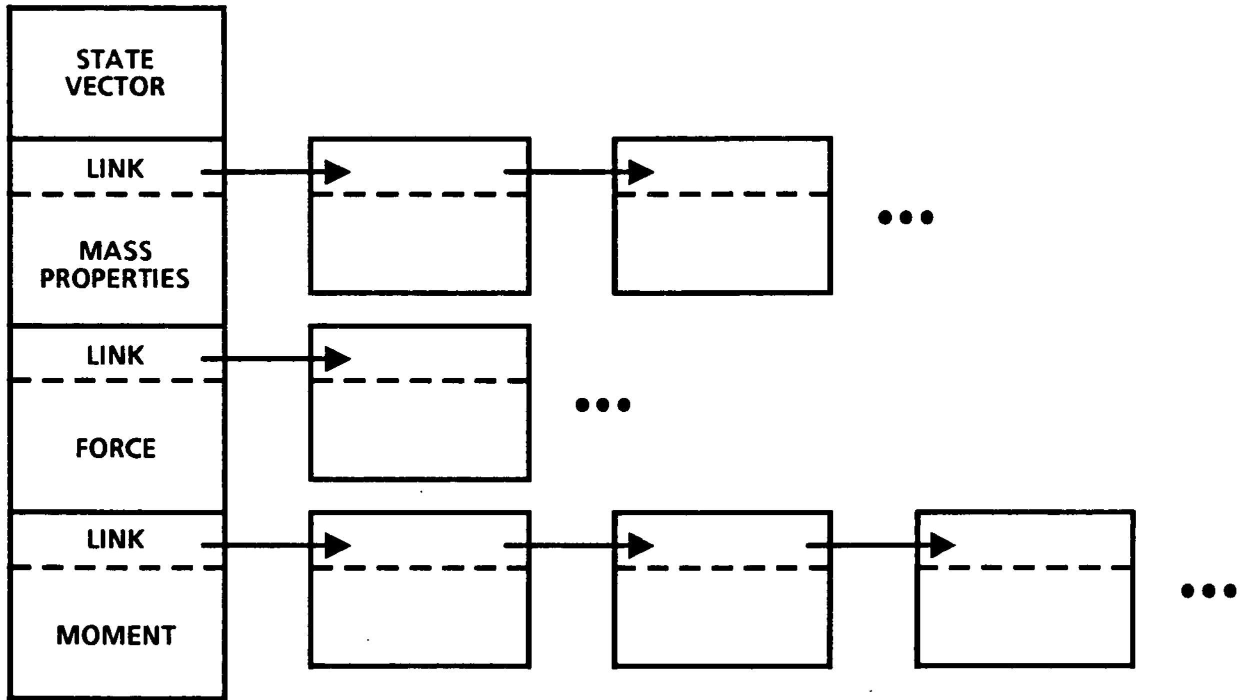

Figure 4.- Free body structure.

are specific to the implementation details of the component data structure. This provides a well-defined template for adding any new free body components that might be required.

6. COORDINATE TRANSFORMATIONS

Each free body component has an associated coordinate frame. During the summation operation, a check is made for compatibility with the frame specified in the summation element. If the frames differ, then the component data must be transformed before the summation operation can take place. By placing the conversion process in the summation routine, the environment models can operate independently from the body to which their output is attached. It is always the end user's responsibility to make the necessary coordinate transformations. This protects math models from the effects of configuration changes that might occur when different test scenarios are selected.

To implement this approach, a transformation processor must be programmed with the necessary conversion procedures. This requires that all coordinate frames be declared during the initialization process. The relationships between frames are then defined in terms of translation vectors and rotation matrices. Five classes of transformations must be considered.

The first of these classes is constant transformations. These are defined during initialization and are invariant during the course of a simulation run.

The second class of transformations consists of those derived from the free body rotational state. They are implicitly defined by the rotational quaternion. During state vector propagation, a rotation matrix is extracted from the quaternion and sent to the transformation processor. This transformation can be viewed as an extension of the free body state.

The third class is related to the free body mass properties. The mass properties data structure includes the location of the center of gravity referenced to some user-defined structural frame. The summation operation thus yields the translation vector from the body's structural frame to the total body center of gravity, which by definition is the origin of the rotational reference frame. This vector is automatically forwarded to the transformation processor during the rotational state update.

The fourth class of transformations includes all those updated during execution of the math models. Their definition is model-specific and not directly related to the propagation of the free body state. The transformation processor only needs to know which two frames are related by the transformation.

The fifth class consists of all transformations defined as combinations of two previously defined transformations. A derived transformation can be defined by specifying its two component transformations and the two frames which it relates. By examining the coordinate frames of the component transformations, the transformation processor can determine the sequence of matrix and vector products that yields the desired compound transformation. This construction method is then stored as part of the transformation definition.

When a coordinate transformation is required during simulation execution, the "from" and "to" frames are sent to the transformation processor, which locates the required rotation matrix and/or translation vector. The lookup procedure must be fairly efficient to avoid degrading simulation performance. For each "from" frame, the transformation processor maintains a linked list of "to" frames. In most cases, there are only a few entries in the list, so search time will not be severe.

Each transformation is tagged with the time of its last update. If the current value is obsolete, then it must be recalculated from the component transformations. This procedure is applied recursively to the component transformations. Eventually, all the components are reduced to one of the first four transformation classes. All of these should be current if the free body hierarchy is properly constructed and math model execution is properly sequenced. In this way, the transformation processor is able to ensure that calculations are not skewed by usage of obsolete data.

The gravitational potential model provides a good illustration of how the transformation processor is used. This model is designed to calculate the gravitational acceleration vector in the geographic reference frame. The input is a translational state vector which is calculated in a user-defined inertial reference frame. The gravity model simply calls the transformation processor to perform the coordinate conversion. After completing its calculation, it calls again for the inverse transformation on the model output. Since such inverse transformations occur

frequently, the transformation processor is designed to generate them automatically when needed.

Centralization of coordinate transformations is an important ingredient for maintaining a generic simulation. It allows math model implementations to be parameterized in terms of coordinate frames. However, there may be some loss in computational efficiency because generic processing cannot take advantage of simplifications such as a diagonal rotation matrix. On the other hand, implicit coding of shortcuts into math models is often a source of error when the math models are later adapted to other purposes.

7. SIMULATION INITIALIZATION

In most simulations, initialization is accomplished by taking a fixed set of input parameters and calculating a fixed set of internal variables. When new models are added to the simulation, the input processor has to be revised to meet the new requirements. In a simulation where many different initial configurations are possible, this approach leads to a proliferation of special control flags and sections of infrequently used code.

A generic simulation by its very nature requires a more flexible approach to initialization. Instead of being coded for all possible initial configurations, the initialization processor has been designed as a flexible grammar that maps user inputs into the simulation data structures. The syntax allows the expression of algebraic relationships between input parameters directly in the initialization file. The initialization processor is fully described in the Rendezvous Expert System Summary Report[1]. A general outline of the initialization processor's major features will be given here.

The initialization processor is implemented with four levels of command processing.

1. Command line preprocessor
2. Executive level command processor
3. Application level command processor
4. General purpose syntax scanner

The first level closely resembles the C language preprocessor. It provides a set of control structures for conditional processing and symbolic substitution. Its purpose is to allow parameterization of the initialization file. Command processing at this level is independent of the simulation structure.

The second level is concerned with initialization of the primary simulation data structures. It is organized as a hierarchical grammar where keywords are used to invoke processing routines (production rules). Keywords are provided for defining

the free body hierarchy and initializing the free body components. Coordinate frames and transformations are also defined at this level.

Any keyword that is not recognized by the executive level is passed down to the application level. A simple table lookup is used to determine the corresponding math model initialization entry point. As new models are added to the simulation, it is a simple matter to append new entries to the table.

Processing at the application level depends on the initialization requirements of individual math models. The syntax scanner provides a set of common facilities which are used to decode tokens from the command line. The basic tokens are keywords, strings, numeric expressions, and Boolean expressions. By combining appropriate calls to the syntax scanner, the production rules specific to a particular math model can be implemented.

An initialization grammar is well-suited for processing key model parameters. However, it tends to be cumbersome for models that have a large number of internal variables. Supplying default initialization values directly in the model is one way to reduce the amount of data in the initialization file. A mechanism for overriding the defaults is occasionally needed. This is accomplished by providing access to symbolic parameters defined by the command preprocessor. The model simply asks for a parameter by name, and if it has been defined, that value overrides the default.

A disadvantage of this approach is that parameter names for a given model must be unique. In contrast, identical keywords can be used for several different models because their meaning is context-dependent. The design of a math model initialization function involves careful consideration of which parameters are common to other models and which are model-specific. Normally, parameters that define the interfaces to other math models will be initialized with keywords. Parameters that define the internal model configuration use preprocessor variables.

8. EXAMPLE

The following example shows how the simulation is configured for a simple rendezvous test scenario with a Space Shuttle approaching the Space Station. The rendezvous radar sensor is providing guidance inputs. Guidance issues commands to the digital auto pilot, which fires the reaction control jets to maintain the desired trajectory.

Lines beginning with "#" are preprocessor commands. Text following semicolons are comments. Character strings are enclosed in quotes. Numeric arguments may be simple values or algebraic expressions.

```
#define Title "Rendezvous Example"

#define PI  3.14159265358979
#define DEG_to_RAD  PI/180.

#define TimeIC  16939373.400    ;  seconds since Jan. 0
#define deltaT  0.040           ;  integration time step (sec)

;  DEFINE REFERENCE FRAMES

; Note : "inertial" and "geographic" are predefined frames
Frame "mean 1950"              ; Aries mean of 1950
Frame "orbiter body"           ; vehicle rotational frame
Frame "orbiter vehicle"        ; vehicle structural frame
Frame "rendezvous radar"       ; radar sensor frame

;  DEFINE COORDINATE TRANSFORMATIONS

; constant transformation
#define Tepoch  16934400.000    ; midnight GMT 7/15/85
Transform "[I:M50]"  To "inertial"  From "mean 1950"
Construct "constant" Matrix  \
  0.683481810691 , -.729963786754 , -.00233360053191 ,  \
  0.729959452662 , 0.683485794199 , -.00251551851639 ,  \
  0.003431220390 , 0.000015877300 , 0.999994113200

; radar location in vehicle structural frame
#define psiR  67.0*DEG_to_RAD
Transform "[R:OV]" To "rendezvous radar" From "orbiter vehicle"
Construct "constant" \
  Matrix -cos(psiR) , sin(psiR) ,  0.0 ,  \
          sin(psiR) , cos(psiR) ,  0.0 ,  \
                0.0 ,       0.0 , -1.0 ,  \
  Vector 565.841/12 , -134.365/12 , 443.875/12
```

```
; primary transformations - updated from rotational quaternion
Transform "[G:I]"  To "geographic"    From "inertial"
Transform "[B:I]"  To "orbiter body"  From "inertial"

; structural to body transformation
Transform "[B:OV]" To "orbiter body" From "orbiter vehicle"
Construct "static" \
  Matrix -1.0 ,  0.0 ,  0.0 ,  \
          0.0 ,  1.0 ,  0.0 ,  \
          0.0 ,  0.0 , -1.0 ,  \
  Vector  0.0 ,  0.0 ,  0.0     ; updated from vehicle CG

; composite transformation
Transform "[R:B]" To "rendezvous radar" From "orbiter body"
Construct Using "[R:OV]" "[B:OV]"

;  INITIALIZE EARTH  -  rotational dynamics only

#define omegaE  7.29211514647e-5  ;  earth rate (radians/sec)
Body  "Earth"  ;  default parent is Root
State "Earth's Rotation"
; rotational state frame and integration method
Rstate "geographic"  Integ "rectangular"
AngVel 0.0, 0.0, omegaE
Euler  "Yaw-Pitch-Roll"  omegaE*(TimeIC - Tepoch), 0.0, 0.0
; quaternion reference frame and integration method
Quat   "inertial"    Integ "rectangular"

;  INITIALIZE SHUTTLE  -  translational and rotational dynamics

Body  "Shuttle"  Parent "Earth"
State "Vehicle"  Frame "mean 1950"  ;  initialization frame
Pos    11915001.5369 , 8086863.11685 ,-16878413.7981  ;  ft
Vel   -12956.2421875 , 21558.2304687 , 1171.98974609  ;  ft/sec
; translational state frame and integration method
Tstate "inertial"  Integ "trapezoidal"
; rotational state frame and integration method
Rstate "orbiter body"  Integ "Cowell"
AngVel 0.0, 0.0, 0.0
Euler  "Pitch-Yaw-Roll" 1.43620392, 0.69847930, 0.27676572 ; rad
; quaternion reference frame and integration method
Quat   "inertial"  Integ "Cowell"

; constant mass properties
Mprop  "Shuttle Body"  Frame "orbiter body"
Mass     7000. , CG  92.48 , 0. , 31.6 , "orbiter vehicle"
Inertia  1005670. , 7419840. , 7760100. , \
           -3545. ,    1030. , -260180.
; summation elements for forces and moments
Fvect  "total body force"  Frame "orbiter body"
Mvect  "total body moment" Frame "orbiter body"
```

```
;  INITIALIZE SPACE STATION  -  translational dynamics only

Body  "Space Station"  Parent "Earth"
State "Target"  Frame "mean 1950"  ;  initialization frame
Pos     11912409.98     , 8091174.56     ,-16878178.96     ;  ft
Vel    -12959.3049      , 21556.1509     , 1176.3413       ;  ft/sec
; translational state frame and integration method
Tstate "inertial"  Integ "trapezoidal"

;  INITIALIZE ENVIRONMENT SENSOR AND CONTROLLER MODELS

OrbRCS "Shuttle RCS"  Vehicle "Shuttle"

Radar "Shuttle Radar"  Vehicle "Shuttle"  Target "Space Station"
Radar Frame "rendezvous radar"

#define LOW  0  ;  table selection index
#define DAP_gain  LOW  ;  low gain
#define DAP_limit LOW  ;  low rate limit
#define DAP_dband LOW  ;  low deadband
OrbDAP "Shuttle DAP"  Vehicle "Shuttle"  OrbRCS "Shuttle RCS"

OrbGuid "Shuttle Guidance"  Vehicle "Shuttle"  Target "Space Station"
OrbGuid OrbDAP "Shuttle DAP"  Radar "Shuttle Radar"
```

9. CONCLUSIONS AND FUTURE DEVELOPMENT

The generic simulation has run successfully on several different computer systems. These include the VAX 11/780, SEL 32-87, IBM PC-AT, and LMI Lambda 2x2 + . This portability has been essential in applying the work to different projects. It has also provided useful benchmarks in the evaluation of possible new computer acquisitions (VAX 8600 and Micro VAX). The execution time for state propagation using trapezoidal integration is 17 milliseconds per body on the VAX 11/780.

The simulation program consists of 18,000 lines of C source code. Use of the C preprocessor has been a key factor in maintaining multiple configurations in a single set of source files. System dependencies, such as file pathname syntax, are handled by conditional inclusion of code segments based on preprocessor control flags. Only the header file defining the control flags has to be maintained separately on the different systems.

The ability to isolate simulation elements by simple modifications to the initialization file has proved helpful in troubleshooting math model problems. This capability also eliminates the need to maintain special driver programs for unit testing. The initialization syntax scanner has been adequate for relatively simple state initializations. An extension to handle matrix and vector operators in addition to the scalar arithmetic operators would greatly improve its utility.

Use of the transformation processor has resulted in much cleaner math model implementations. With messy coordinate conversions removed, the underlying algorithms are more clearly defined and easier to maintain. However, the definition of the required transformations in the initialization file is the price that has to be paid for this simplification. The situation could be improved by adding a construction algorithm to the transformation processor. By examining the lists of primitive transformations, it would be possible to automatically define any compound transformation that might be needed.

The coordinate conversions do not include scaling transformations. Such a capability would be possible if a measurement units attribute were added to the simulation data structures. This would enable more comprehensive compatibility checking.

However, the requirement for additional initialization data and the reduction in processing speed outweighed the potential benefits.

The object-oriented approach to math model design enables more rapid software development. Since all models must provide the same set of interface functions, it is often possible to take an existing model and modify it slightly to meet different requirements. Most of the differences between models tend to be concentrated in the state update function. A programming language which directly supports inherited characteristics would be superior to C language in this respect.

The simulation is implemented as a single computation task and a separate input/output interface task. The interface task can be customized for particular project or system requirements without disturbing the main simulation. For applications requiring real-time execution speeds, it might be desirable to divide the computation task for parallel processing. However, the transformation processor makes this difficult since it has to be easily accessible from all the math models.

ACKNOWLEDGMENTS

This work was carried out at the Systems Engineering Simulation Laboratory at the National Aeronautics and Space Administration (NASA) Lyndon B. Johnson Space Center (JSC) under contract NAS 9-15800, job order 24-123.

REFERENCES

1 DUNN, C.
Rendezvous Expert System Summary Report. LEMSCO-23427 (November 1986).