RL-TR-96-257
Final Technical Report
March 1997

MIXED FIDELITY SIMULATION TECHNOLOGY DEVELOPMENT

Nichols Research Corporation

B. Gossage, W. Roark, J. Bass, J. Kyser, D. Salazar,
and J. Brown

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-257 has been reviewed and is approved for publication.

APPROVED:

ALEX F. SISTI
Project Engineer

FOR THE COMMANDER:

JOSEPH CAMERA, Technical Director
Intelligence & Reconnaissance Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/IRAE, 32 Hangar Road, Rome, NY 13441-4114. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1997	Final Feb 95 - Feb 96

4. TITLE AND SUBTITLE

MIXED FIDELITY SIMULATION TECHNOLOGY DEVELOPMENT

5. FUNDING NUMBERS

C: F30602-95-C-0035
PE: 62702F
PR: 4594
TA: 15
WU: N0

6. AUTHOR(S)

B. Gossage; W. Roark; J. Bass; J. Kyser; D. Salazar; J. Brown

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Nichols Research Corporation
4040 South Memorial Parkway
Huntsville AL 35815-1502

8. PERFORMING ORGANIZATION REPORT NUMBER

NRC-TR-96-061

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/IRAE
32 Hangar Road
Rome NY 13441-4114

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-257

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Alex F. Sisti/IRAE/(315) 330-4518

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Mixed models with different levels of fidelity within the same simulation present a difficult challenge to simulation developers and users. Very often, changes in fidelity impose changes in the model interfaces and the fidelity requirements of other models.

This report presents the results of our investigation of object oriented approaches that facilitate simulations composed of models of various fidelity levels. The terms "fidelity," "fidelity level," "fidelity boundary," "mixed fidelity," and model "validity" are defined to provide a rigorous framework for discussion. Practical approaches towards defining an interface paradigm that exploits polymorphism and the inheritance characteristics of an object oriented hierarchy are described. These techniques are demonstrated within the context of a sample geolocation problem implemented in C++. In addition, we provide guidelines for model and class reuse in C++ libraries. Other potential technologies that hold additional potential for enhancement of mixed fidelity simulation such as distributed objects and parametrized types are also explored.

14. SUBJECT TERMS		15. NUMBER OF PAGES
Mixed Fidelity Simulation Distributed Simulation	Object Oriented Hierarchical Simulation	110 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Same as Report

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF LISTINGS

TABLE OF TABLES

LIST OF ACRONYMS

FDOA	Frequency Time of Arrival
GPS	Global Positioning Satellite
GUI	Graphical User Interface
MFC®	Microsoft Foundation Classes
MICOM	Missile Command
MIRSAT	MICOM Infrared Seeker Analysis Tool
RTTI	Run Time Typing Interface
TDOA	Time Difference of Arrival

1. INTRODUCTION/OVERVIEW

The complexity of computer models continues to grow with the increase in sophisticated software technologies and distributed computing resources. The tradeoffs between model fidelity, simulation run time, and the reuse of legacy models generate the requirement for models with different levels of fidelity within the same simulation. Creating and maintaining the interfaces between models with different fidelity levels is a challenging task for the simulation developer. Assembling mixed fidelity simulations in flexible and meaningful ways is also a challenge for the simulation user. Object oriented technologies can provide solutions to these problems in both dynamic and static simulation environments. This paper presents some results of our exploration into object oriented solutions to mixed fidelity simulation. The design of all classes and object relationships are presented in Booch diagrams [Bch93][Wht94]. All code examples are presented in C++.

1.1 Previous Work

Over the last five years, Nichols Research Corporation (NRC) has been developing an object oriented simulation environment that allows users to assemble simulations of arbitrary complexity from component objects within a visual environment. Currently called "FlexSim," we have used it as the framework for testing our solutions to mixed fidelity simulation. In FlexSim, objects are characterized by a set of *parameters* and a set of *references*. This approach is based on the ideas developed by Zeigler [Zei90] and similar to the framework described by Lewandowski and Calhoun [Aan94]. An object's references define direct connections to other objects. Only objects of the reference class or subclass type can be assigned to an object's reference. A standard set of services defined by an abstract base class, **CFlexObject**, must be provided by any object class that participates as a FlexSim object. These member functions support the visual inspection of object parameters, the resolution of its references, and object archiving. A visual environment is provided that allows the user to manipulate objects on screen by inspecting/setting their parameters and connecting their references to other simulation objects. Providing the user with the option of creating and connecting objects with different levels of fidelity within the same simulation was one goal of this effort.

2. THEORETICAL DISCUSSION

2.1 Definition of Terms

The use of the term *fidelity,* with respect to a model, is often accompanied by a wide variety of assumptions about its exact meaning. Very often, the term implies *validity*. Validity is the degree to which a model accurately predicts the behavior of the system. We define *fidelity* as the degree to which a model faithfully and accurately represents the details of the system. The details of the system are captured by a set of assumptions that are used as the basis of a mathematical or logical model. In the context of computer simulation models, high fidelity does not imply a high degree of validity. A requirement for high validity in a model does not imply a requirement for high fidelity. We can illustrate these concepts with the following simple example. Suppose a system $F_T(x)$ obeys a cubic polynomial of the form:

$$F_T(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

We are given the choice between a "low" fidelity model:

$$F_L(x) = l_2x^2 + l_1x + l_0$$

and a "high" fidelity model:

$$F_H(x) = h_3x^3 + h_2x^2 + h_1x + h_0$$

To build each model we fit these polynomials to N=10 samples of $F_T(x)$ using linear regression (with MathCad(R)). If the true coefficients of $F_T(x)$ are **a** = [0.01, 0.01, 4.0, 1.0] and the measurement error standard deviation is $\sigma = 2.0$, then the resulting mean square error between the estimated polynomials and the sample points for the two models are $SS_L = 6.465$ and $SS_H = 6.238$, but the mean square errors between the estimated polynomials and the true polynomial are $SS_{LT} = 0.245$ and $SS_{HT} = 0.473$. We may conclude in the absence of knowledge about the true system that the high fidelity model is "better" since it does a better job of explaining the observed data and is more faithful to the details of the underlying true system. We would be unaware that the high fidelity model is overfit to the data [Dra81] resulting in $SS_{LT} < SS_{HT}$. By this measure of validity we should have chosen the more robust "low" fidelity model. How can this be avoided without knowledge of the true system? Note that $SS_L \approx SS_H$ so the high fidelity model is only slightly better and if both models meet the acceptance criteria, we apply "Occam's razor" and choose the simpler, low-fidelity model. Scale analysis [Hol92] can also be used to avoid unnecessary levels of detail in a model. This example shows that greater model fidelity does not imply greater model validity. Hence, model fidelity and model validity are separate, sometimes independent, concepts.

As in our simple example, greater model fidelity often results in greater model complexity. The high-fidelity model required an extra parameter for the coefficient of x^3. Since the interfaces of the two models are identical, the value of x, the selection of either model in a simulation would have no impact on the interface requirements of other models. If we change F_H to require another input y:

$$F_H(x,y) = h_3y + h_2x^2 + h_1x + h_0$$

F_H has not only greater internal complexity than F_L, but also greater complexity in its interface. Exchanging F_L for F_H in a simulation imposes a change in the interfaces for all models that supply inputs to F. Changing model interfaces is one of the fundamental problems in mixing the fidelity of models within the same simulation system. This is a direct result of the increase in model complexity that accompanies an increase in model fidelity.

To avoid any ambiguity between terms like "fidelity" and "validity" we present the following definitions that form the foundation of discussion for this document.

Model Fidelity — the level of detail in a model resulting from a set of modeling assumptions about the system it represents. Low fidelity models are characterized by simplifying assumptions which reduce their level of detail compared to high fidelity models.

Model Fidelity Level — a partition into different levels of detail imposed on a set of models by one or more modeling assumptions. For example, the assumption of a spherical Earth partitions a set of models into a subset of models all of which assume a spherical Earth.

Fidelity Boundary — An interface between two models in different fidelity levels.

Mixed Fidelity Simulation — a simulation composed of a set of models from different fidelity levels.

Validity — an objective, often statistical assessment of the ability of a model to accurately predict the behavior of the system it represents.

Verification — a simulation model is verified when it is shown to be a faithful implementation of the model.

2.2 Fidelity and Model Reuse

While changing model interfaces presents a constant challenge for mixing model fidelities in a dynamic modeling environment, static reuse of existing models is also a significant component of the problem. Static reuse occurs through changes to the model and/or the source code for simulation executive. While usually tractable, the design of the model class relationships and interfaces can minimize the extent of the changes required. Encapsulation and polymorphism provided by OOP help, but careful design of the class hierarchies to prevent dependencies on non-portable class libraries or the creation of direct dependencies on the model implementations is essential. We will present approaches to model reuse as a companion problem to mixed fidelity simulation.

3. TECHNICAL SOLUTIONS

We present several proposed solutions to mixed fidelity simulation. Some have only been explored conceptually while others have been fully implemented in C++ within the FlexSim environment. Some solutions can be implemented within a dynamic simulation environment that allows a user to choose the models and their inter-connections at run time. These solutions take advantage of the inheritance and polymorphism features of object oriented languages. Other solutions are static, compile-time changes to the implementation of models that take advantage of the multiple inheritance and parameterized type (template) features found in C++.

3.1 Dynamic Solutions

3.1.1 Common Base Classes

When a high fidelity model produces output to an interface embodied by an information object, models of the same fidelity or lower may use that information provided that the information object base class forms the low fidelity interface. The low fidelity model interface is a pointer to an object of base class type while the high fidelity model output is an object of a type derived from that same base class. This construct assumes that information content increases as we traverse the information object hierarchy from base to derived as shown in Figure 3-1. Any object of the derived information object type will complete the base type interface required by the low fidelity model through public inheritance. This solution works well within the FlexSim environment since a reference to a low fidelity input can be connected to either a high or low fidelity object.

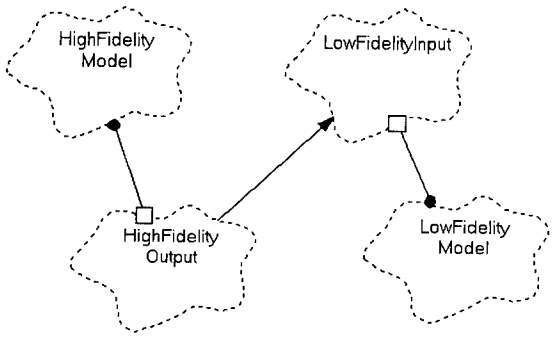


Figure 3-1 Information content increases as the interface object class hierarchy is traversed from base to derived.

3.1.2 Interface Expansion

When two models exchange information, a convenient implementation of that interface is a shared "information" object as illustrated in Figure 3-2. This interface may be a single object or a dynamic collection of many objects (containers). Such information objects can appear at fidelity boundaries and can form a key information transport mechanism across fidelity boundaries. When the information flow is from a low fidelity to a high fidelity model, the interface to the high fidelity model must be expanded. This can be accomplished through a member function that returns an object which satisfies the high fidelity interface. In this way, an information object is viewed as a "smart" container which knows how to represent its contents in various forms. Another solution is for the high fidelity information object to provide a conversion constructor with a low fidelity object as its argument. The advantage of the conversion constructor is that a common base class can serve as the low fidelity interface above. The expansion member function approach would require independent high and low fidelity object interfaces since derived types are not within the scope of their base class. In either case it must be noted that the interface expansion is isentropic - no information gain can take place.

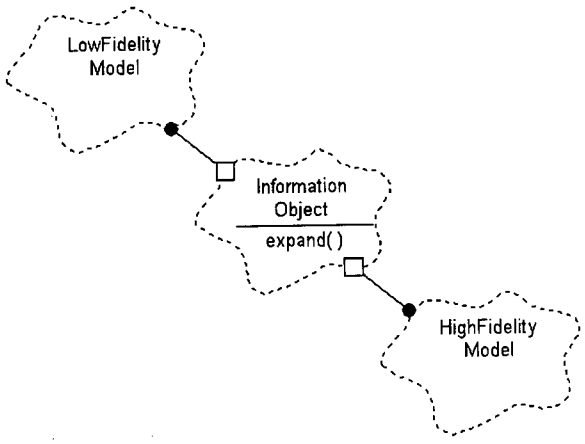


Figure 3-2 Model interfaces through information objects.

3.1.3 "Fat" Base Classes

Another approach that employs inheritance to enable mixed fidelity interfaces is the "fat" base class. The interface to the highest fidelity model is captured by an abstract base class with all the foreseeable member functions for that model. To allow maximum flexibility, the number of member functions can be quite high, hence the term "fat." This approach allows models of any fidelity to be substituted in a simulation so long as they meet the interface requirements of the base class. For example, low fidelity models may implement some of the required member functions by returning constants or simple table lookups while a high fidelity model may accomplish the same result using a set of complex equations. In C++, all the member functions of the base class are declared as "pure virtual" functions and all other simulation models requiring access to models of this type will use variables of type: *pointer-to-base*. The use of virtual functions will cause the actual model to be accessed correctly at run-time. Their declaration as "pure" forces derived classes to complete the specified interface or cause a run or compile-time error. An example of a "fat" base class can be found in the MICOM Infrared Seeker Analysis Tool (MIRSAT) simulation [Can95]. MIRSAT uses a "6-DOF" model to capture missile and seeker flight dynamics. Since 6-DOF implementations vary from one missile system to another, it was necessary to be able to substitute different models without changing the interfaces in the rest of the seeker model. This was accomplished by defining a "fat" **SixDofModel** base class. The Booch diagram of class **SixDofModel** and its related subclasses appears in Figure 3-3. Listing 3-1 contains the definition of the **SixDofModel** class with its numerous member functions. The **TrajSixDof** model reads a trajectory file and interpolates the data while the **TrapSixDof** and **TacSixDof** models run legacy FORTRAN 6-DOF codes with high-fidelity flight dynamics.

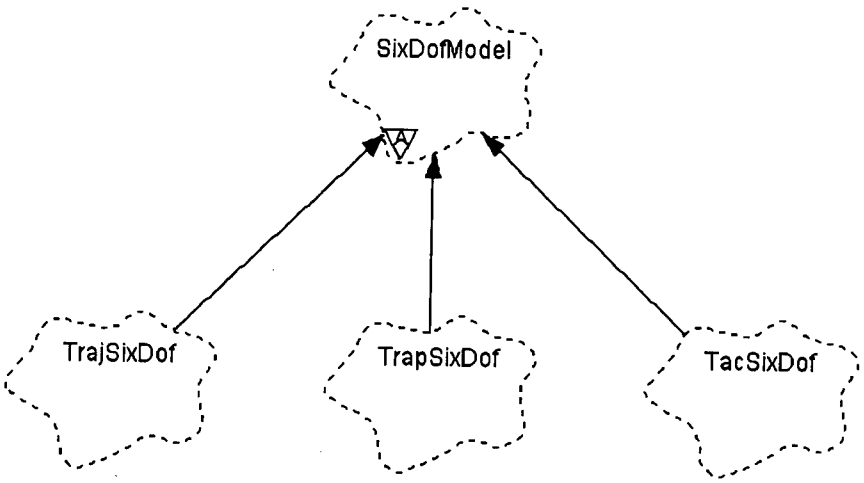


Figure 3-3 The abstract base class SixDofModel and its known subclasses.

```
class SixDofModel : public SimObject
{
  public:

  // Pure virtual functions...
  // (These MUST be redefined by any derived class.)

  // Target related functions...
   virtual Vector targetPos() const=0;
   virtual Vector targetVel() const=0;
   virtual float targetX() const=0;  // target x pos (m)
   virtual float targetY() const=0;
   virtual float targetZ() const=0;
   virtual float targetYaw() const=0;
   virtual void setTgtZ( float z ) =0;

  // Missile related functions...
   virtual Vector missilePos() const=0;
   virtual Vector missileVel() const=0;
   virtual float missileX() const=0;
   virtual float missileY() const=0;
   virtual float missileZ() const=0;
   virtual float range() const=0;
   virtual float inertTgtAz() const=0;
   virtual float inertTgtEl() const=0;

  // Launch aircraft related functions...
   virtual float initACaz() const=0;
   virtual float initACel() const=0;

  // Six DOF control functions...
   virtual float timeToGo() const=0;
   virtual int terminated() const=0;
   virtual String termString() const=0;
   virtual void flyoutLoop() =0;
   virtual void trackControl( float x, float y ) =0;

};// end class SixDofModel
```

Listing 3-1 Definition of "fat" base class SixDofModel in C++.

3.1.4 Property Decomposition/Aggregation

One of the goals of an object oriented simulation design is to create a transparent mapping from the classes and objects in the simulation to the objects in the system under study. This provides clarity for the simulation user and makes the simulation source code easier to maintain. For example, we have created a class **Planet** to capture the properties common to all planets. These properties include the planet's gravity and shape as a function of latitude and longitude. Usually we tie these functions to planet models with equivalent fidelity – a planet model with a high fidelity shape model would also have a high fidelity gravity model. If the fidelity of the gravity and shape functions can be considered independent from one another, a simulation could consist of models which make simple assumptions about a planet's shape while other models assume a complex gravity field. The decoupled planet properties can be captured in separate **Gravity** and **Shape** base classes. The class **Planet** is modified to use instances of **Shape** and **Gravity** models through containment by reference as shown in Figure 3-4. Any Gravity or Shape model can be assigned to the **Planet** model references provided the base classes have "fat" interfaces as describe previously.

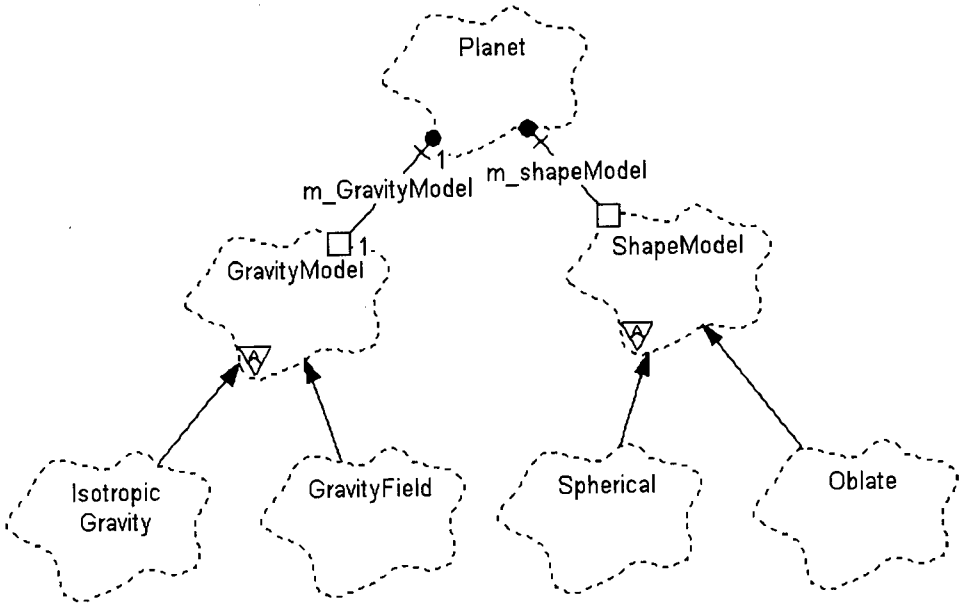


Figure 3-4 The Planet properties are decomposed into separate Shape and Gravity models.

3.1.5 Implicit Model Inconsistencies

When the user is provided a dynamic simulation environment which allows mixed fidelity simulation, no compile time or run time checks are carried out to assure the validity of the results. While the simulation environment can provide interface checking, parameter verification, and results analysis, it cannot prevent the naive use of complex models in those cases where interfaces are not violated. Many simulation models have complex interfaces and operational requirements that overwhelm all but the most expert of users. The use of these models is often limited to their original developers. A possible solution to this problem is to embed an expert system within the simulation environment.

By way of the same information services provided to the user interface by the models (parameters and references) information can be input to the expert system. This information will in turn activate a set of predefined rules captured from model experts and developers. The effect of rule "firings" would be to notify the user of potential problems or errors in the simulation and provide corrective guidance. We have not explored this concept further since it is beyond the scope of the current effort.

3.2 Static Solutions

3.2.1 Model Reuse via Multiple Inheritance

Reusing existing model classes without "contaminating" them with proprietary or non-portable code is part of efficient mixed fidelity modeling in a static environment. We have defined a static simulation environment as one in which source code changes must be made to make model substitutions. Careful use of multiple inheritance can minimize this problem by avoiding direct inheritance of system dependent classes by external model classes. If a model class is to be kept independent, then it must not become a descendant of any class within the simulation system hierarchy. To illustrate this, suppose we are using GUI class library BRANDX (similarity to the names of other class libraries is strictly unintentional) to implement the interface to our simulation. We use a BRANDX class **Displayable** to display model information on the screen. We also have a legacy model class **ModelIt** that we wish to reuse. At first, we may be tempted to modify **ModelIt** by deriving it from **Displayable** and adding the appropriate members. If we use this approach, we have modified **ModelIt** in a way that prevents it from being ported to any environment that does not have access to BRANDX. This can be costly or impossible if BRANDX cannot be purchased for the target platform. Multiple inheritance provides a better solution by allowing a new class **DisplayableModel** to be created by inheriting from both **Displayable** and **ModelIt**. **ModelIt** remains completely independent of the BRANDX class library allowing it to remain portable. This approach also supports the configuration management of **ModelIt** to be maintained at a single location. In this project we have made frequent use of multiple inheritance to keep external models in separate, independent and portable libraries. The general form for this approach appears in Figure 3-5.

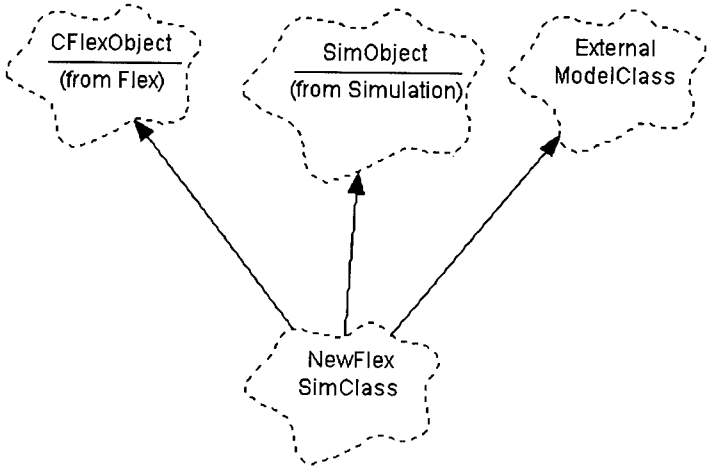


Figure 3-5 Illustration of the use of multiple inheritance from an external class library.

The inheritance from the external library class may be public or private. Public inheritance would result in the inheritance of interface of the external class which could create problems if the interface is used by classes within the simulation and then the external class needs to be replaced by a class with a different interface. Any classes using the external model interface would require changes. This could be prevented by inheriting **private** from the external model class and creating a new generic interface in the derived class. The implementation provided by the external model would be reused, but not its interface. For this reason, private inheritance is called *implementation inheritance* [Str91]. The external model must declare its member variables **protected** if the simulation class variables are to be manipulated in the derived class. An example of the use of multiple inheritance for model reuse within the sample problem will be presented later.

3.2.2 Parameterized Types

Parameterized types (templates in C++) allow the creation of new models with different levels of fidelity that depend on the template class argument. For example, different orbital dynamics' models can be created by instantiating an integrator template class **RungeKutta4** with the appropriate model class that describes the dynamics of the orbit (gravity, solar pressure, etc.) The relationship between the template and its argument is shown in Figure 3-6. The code fragment in Listing 3-2 illustrates this idea with a C++ template class. Parameterized types offer some distinct advantages over polymorphic classes including type safety, efficiency and ease of use [Car95].

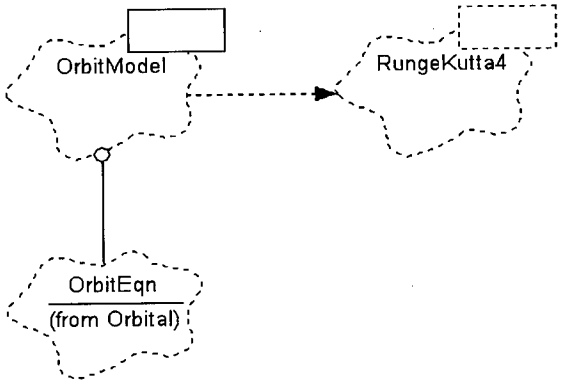


Figure 3-6 Instantiation of template class RungeKutta4 creates new OrbitModel class.

```
template<class DiffEqn>
class RungeKutta4
{
  public:
    RungeKutta4( DiffEqn& anEqn, double stepSize );
    void integrate( double dt );
    void setRelErr( double x ) {relErr = x;}
    void setAbsErr( double x ) {absErr = x;}

};// end template class RungeKutta4

class OrbitEqn
{
  public:
   OrbitEqn( DoubleArray initState, double aTime );
   void derivs( double t, const DoubleArray& x,
  DoubleArray& dx );

}; // end class OrbitEqn

// Create a new class by instantiating the template...
typedef RungeKutta4<OrbitEqn> OrbitModel;
```

Listing 3-2 Example of model creation through a parameterized type in C++.

4. SAMPLE PROBLEM

To establish the feasibility of the various approaches to mixed fidelity simulation, we implemented an object oriented simulation of a geolocation problem within the FlexSim framework. The geolocation problem involves the determination of transmitter locations on Earth's surface by Time Difference of Arrival (TDOA) and Frequency Difference of Arrival (FDOA) measurements [Ho93][Ont89]. The measurements are generated from signals intercepted by satellites with highly accurate clocks similar to Global Positioning Satellites (GPS) [Her96].

4.1 FlexSim Overview

FlexSim is composed of two component frameworks: "Flex" and "Sim." The "Flex" framework supports the manipulation of objects via screen icons in a GUI environment. This manipulation includes the setting of object properties, connecting objects via "references," and object archival. The "Sim" framework supports event-based simulation in an object oriented programming environment. Objects create and schedule events on a global event calendar in step with a global simulation clock. The two frameworks are combined in FlexSim to create a visual, object oriented simulation environment.

4.1.1 The *Flex* Class Framework

The *Flex* class framework provides support for the key capabilities of the visual object manipulation system. These capabilities include:

- The creation of object instances of a class given the name of the class.
- The setting of object references.
- The setting of object attributes.
- The storage of objects in a persistent archive.

- The grouping of objects into subsystems.

The base class **CFlexObject** defines the interface requirements for any class that participates in the *Flex* system. The virtual functions "Copy," "GetAttributes," "GetReferences," and "Serialize" must be defined by any class derived from **CFlexObject**. The "Copy" member is generated automatically by a system of C++ macros that generate all the required code for the Run-Time Typing Interface (RTTI) and meta-class behaviors. Currently, the **CFlexObject** class is derived public from the Microsoft Foundation Classes (MFC®) class **CObject**. This direct dependence on a proprietary class library may be removed in future versions of the *Flex* framework.

Since C++ does not support the concept of "meta-classes," class names are mapped to a predefined instance of that class. The predefined object is copied by way of the virtual "Copy" member function defined by all *Flex* classes.

Support for the setting of object attributes is provided by the abstract base class **Attribute** and its derived classes as shown in Figure 4-1. The derived classes allow objects to export attributes to the *Flex* system for manipulation by the user. The "GetAttributes" function returns a list of attributes for a given object. Note that the **IntegerAttr** and **RealAttr** classes are templates so that the various real and integer types (e.g. unsigned, float, double) can be easily supported without the need to provide **Attribute** subclasses for all integral types.

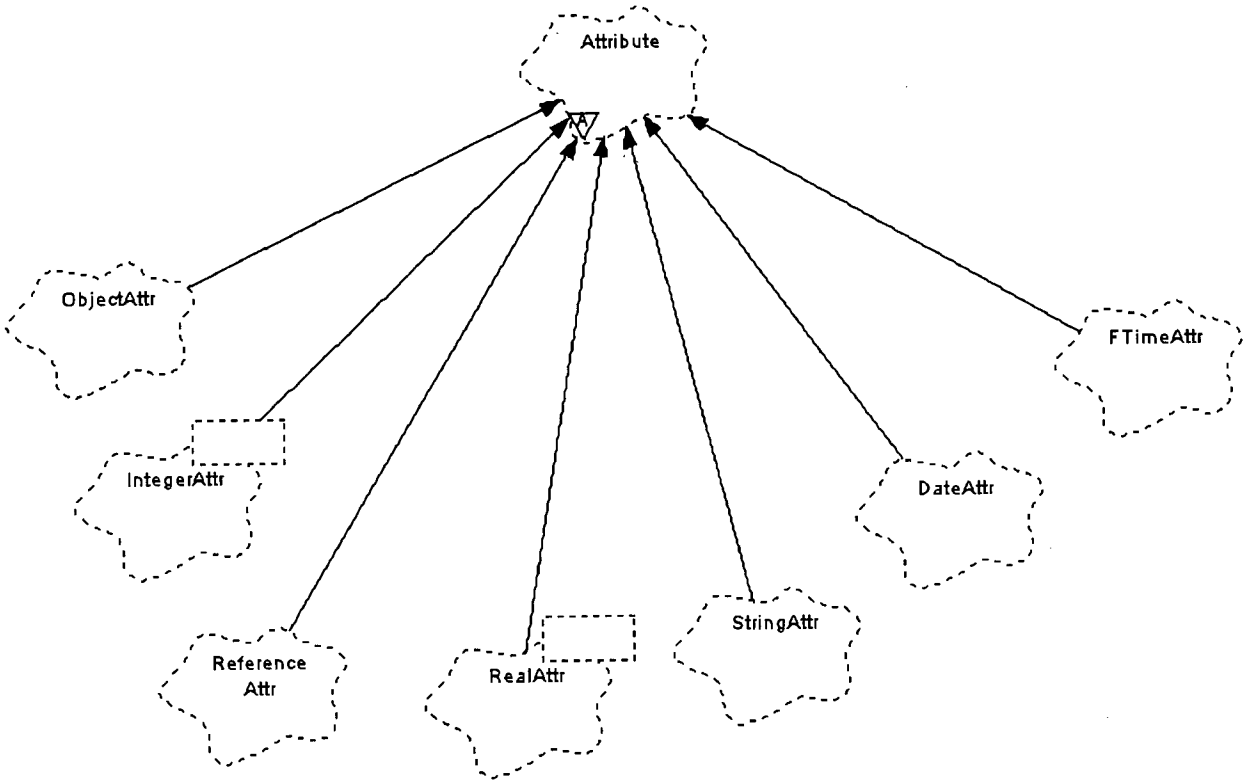


Figure 4-1 The Attribute classes.

The resolution of object references (ports) is accomplished via the class **Reference**. A Reference contains a pointer to an object reference (a C++ object pointer), the class type of the object reference, a

label string, and a flag indicating whether setting the reference is optional. Setting the object reference is accomplished through the reference pointer.

The reference resolver uses the class type to limit the values of the reference to objects that are instances of that class or its subclasses. The label string represents the relationship of the object to the referenced object. For instance, a sensor's platform reference captures the relationship between the sensor and its platform as in "the sensor *is on* the platform."

4.1.1.1 Role of MFC®

The *Flex* framework is based on MFC®. MFC®, GUI, RTTI, archival, and collection classes are used extensively and many of the *Flex* windowing classes are directly (and publicly) derived from MFC® base classes. Figure 4-2 shows the relationships between MFC® classes and the *Flex* framework. All of the classes that participate as FlexSim objects are derived from **Cobject** through **CFlexObject**. This allows the storage and retrieval of FlexSim objects using MFC® collections e.g., **CObList**.

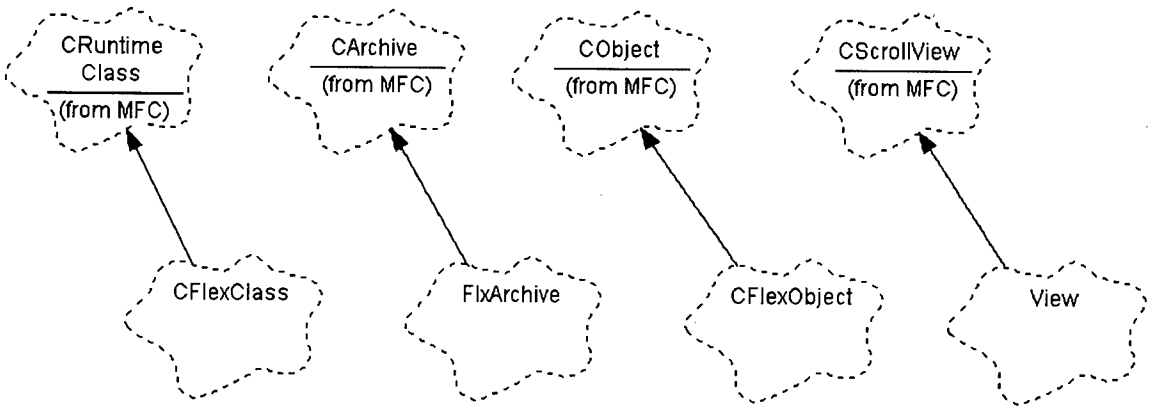


Figure 4-2 Relationship between MFC® and Flex frameworks.

4.1.2 Sim Class Framework

The simulation class framework (see Figure 4-3) provides support for the key capabilities of the event-based simulation system. The capabilities include:

- The initialization of simulation objects.
- The scheduling and execution of events in time order.
- The maintenance of current simulation time by a simulation clock.
- The start and termination of a simulation run.

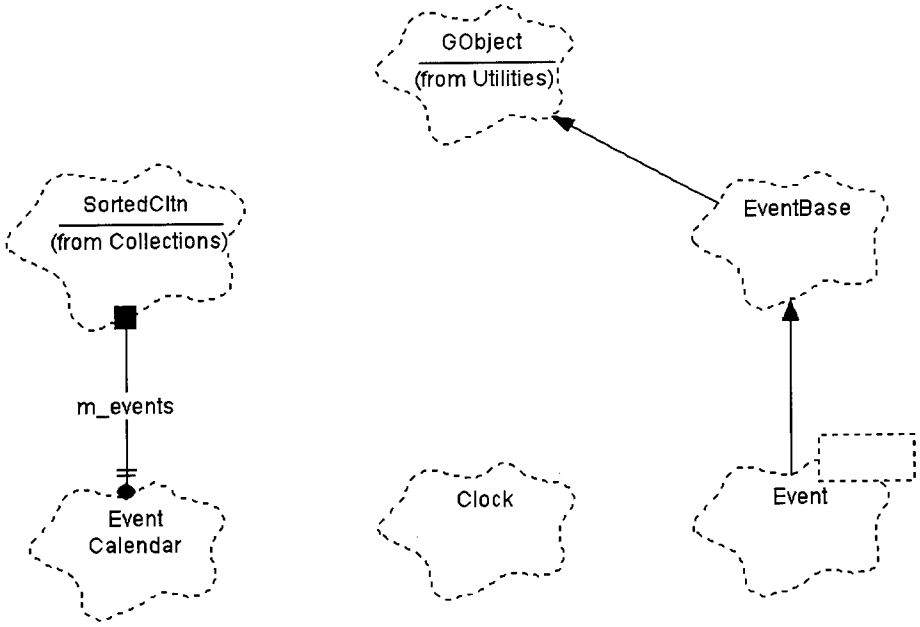


Figure 4-3 Simulation framework classes.

4.2 Geolocation Class Design

The geolocation class design was partitioned into three class categories – database classes, the geolocator class, and matrix classes. The design and construction of each of these class categories will be discussed in the following three sections.

4.2.1 Measurement Database Classes

The overall structure of the measurement database design, as implemented to support the sample problem, is illustrated in Figure 4-7. Each database class will be discussed individually in the following sections.

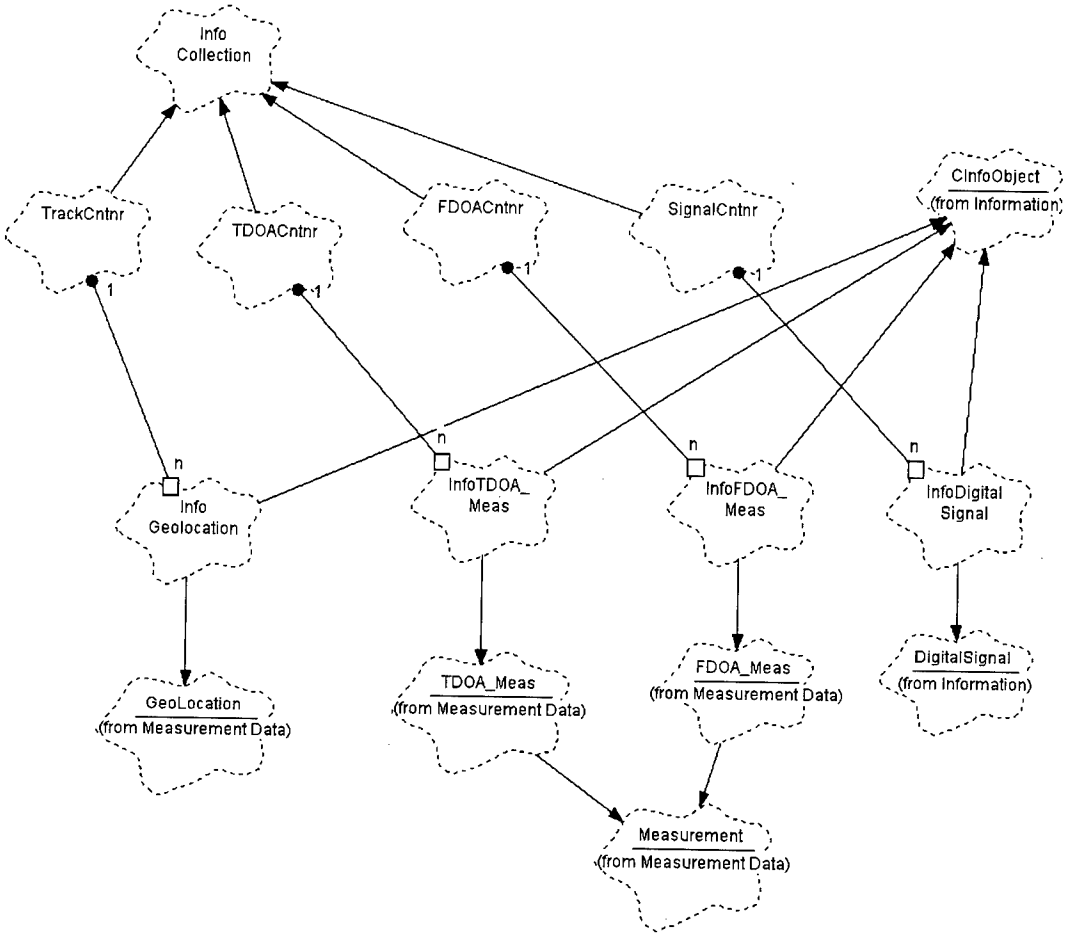


Figure 4-7 Overview of the database classes are shown from the class diagram perspective.

4.2.1.1 Class: Measurement

The class **Measurement** is the base class for the TDOA and the FDOA measurement classes. It contains information such as reference sensor position and velocity and other basic information fields and functions required by both its derived classes. The header file (Measurement.h) for **Measurement** is shown in Listing 4-1.

```
{ class Measurement
public:
  Measurement();
FTime time( ) const { return m_measTime; }
// Returns the variance of the measurement.
  double variance( ) const { return m_var; }
  // Return the measurement error bias.
  double bias( ) const { return m_bias; }
// Return the position vector of the reference sensor.
  Vec3 refSensorPos( ) const;
// Return the velocity vector of the reference sensor.
  Vec3 refSensorVel( ) const;
// Return the position vector of the first (non-reference)
  // sensor.
  Vec3 firstSensorPos( ) const;
// Return the velocity vector of the first (non-reference)
  // sensor.
  Vec3 firstSensorVel( ) const;
// Sets the measurement time
  void setMeasTime( const FTime& aTime) { m_measTime = aTime;}
// Sets the ECR position and velocity of first sensor
  void setSensor1Pos( const Vec3& aVec) { m_sensor1Pos = aVec;}
//  Sets the velocity of the first sensor
  void setSensor1Vel( const Vec3& aVec ) { m_sensor1Vel = aVec;}
// Sets the ECR position of the reference satellite
  void setRefSensorPos( const Vec3& aVec) { m_refSensorPos = aVec;}
//  Sets the velocity of the reference satellite
  void setRefSensorVel( const Vec3& aVec ) { m_refSensorVel = aVec; }
// Sets the reference sensor number
  void setRefSensorNum( int aNum) { m_refSensorNum = aNum;}
// Sets the first sensor number
  void setfirstSensorNum( int aNum) { m_firstSensorNum = aNum;}
// Sets the variance of this FDOA measurement
  void setVar( double& aVar) { m_var = aVar;}
// Sets the bias variance
  void setBias( double& aBias) { m_bias = aBias;}
protected:
  // Measurement time. Arbitrarily the time of arrival at
  // reference satellite.
  FTime m_measTime;
// Caculated difference in time of arrival.
  double m_deltaF;
// ECR position and velocity of first sensor
  Vec3 m_sensor1Pos;
 Vec3 m_sensor1Vel;
// ECR position and velocity of reference satellite.
  Vec3 m_refSensorPos;
  Vec3 m_refSensorVel;
// Reference sensor number. (arbitrary)
  int m_refSensorNum;
// First sensor number
  int m_firstSensorNum;
// Variance of this FDOA measurement computed as:
  // varSPT + 2 * varAtmR + 2 * varBias
  double m_var;
// The bias variance computed as:
  // varClock + varAtmB
  double m_bias;
}; // end class Measurement
```

Listing 4-1 Header file for class Measurement.

4.2.1.2 Class: FDOA_Meas

The class **FDOA_Meas** is a derived class of **Measurement**. **FDOA_Meas** contains the data required to describe a FDOA measurement. The header file (FDOA_Meas.h) for class **FDOA_Meas** is shown in Listing 4-2.

```
class FDOA_Meas : public Measurement
{
 public:
   FDOA_Meas();
   double cntnrFreq() const { return m_cntnrFreq; }
   void setCntnrFreq( double aFreq ) { m_cntnrFreq = aFreq; }
   double deltaFreq() const { return m_deltaFreq; }
   void setDeltaFreq( double aFreq ) { m_deltaFreq = aFreq; }
   double sigmaSPF() const { return m_sigmaSPF; }
   void setSigmaSPF( double aSigma ) { m_sigmaSPF = aSigma; }

 protected:
   // Caculated difference in time of arrival.
   double m_cntnrFreq;
   double m_deltaFreq;
   double m_sigmaSPF;

}; // end class FDOA_Meas
```

Listing 4-2 Header file for class FDOA_Meas.

4.2.1.3 Class: TDOA_Meas

The class **TDOA_Meas** is a derived class of **Measurement**. **TDOA_Meas** contains the data required to describe a TDOA measurement. The header file (TDOA_Meas.h) for class **TDOA_Meas** is shown in Listing 4-3.

```
class TDOA_Meas : public Measurement
{

 public:
   TDOA_Meas();
   double delTime() const { return m_deltaT; }
   void setTime( double aTime ) { m_deltaT = aTime; }
   // Returns the covarance between this TDOA measurement and
   // another.
   double covWith( TDOA_Meas& aTDOA ) const;

 protected:
   // Caculated difference in time of arrival.
   double m_deltaT;

}; // end class TDOA_Meas
```

Listing 4-3 Header file for class TDOA_Meas.

4.2.1.4 Class: GeoLocation

The class **GeoLocation** contains the state information for the position estimates as generated by the class **GeoLocator**. The header file (GeoLocation.h) for class **GeoLocation** is shown in Listing 4-4.

```
class GeoLocation
 public:
  GeoLocation();

  //  Initialize this GeoLocation with a set of TDOA
  //measurements.
  void initializeWith( TDOA_Set& tdoaSet );

  // The time of the estimate.
  FTime time() const { return m_time; }
  Vec3 position() const { return m_ecrPos; }
  Vec3 velocity() const { return m_ecrVel; }
  void setPosition( const Vec3& pos );
  void setVelocity( const Vec3& vel ) { m_ecrVel = vel; }

 protected:

  // Data Members
  FTime m_time;
  //latitude in radians. (pi/2 <= lat <= -pi/2)
  double m_lat;
  //location longitude in radians (pi <= long <= -pi)
  double m_long;
  //latitude variance
  double m_varLat;
  //longitude variance
  double m_varLong;
  //Covariance of lat and long
  double m_covLatLong;
  //The track id number.
  unsigned m_trkID;
  //Rate of change in latitude. (rad/s)
  double m_latDot;
  //Rate of change in longitude (rad/s)
  double m_longDot;
   //Altitude in meters.
  double m_alt;
  //Rate of change in altitude (m/s).
  double m_altDot;
  //Variance of the altitude estimate.
  double m_varAlt;
  //  ECR position vector...
  Vec3 m_ecrPos;
  //  ECR velocity vector...
  Vec3 m_ecrVel;

  // class variables...
  static int c_baseID;

}; // end class GeoLocation
```

Listing 4-4 Header file for class GeoLocation.

4.2.1.5 Class: DigitalSignal

The class **DigitalSignal** contains the information germane to a digital signal. The signal is generated by the class **SignalProcessor**. The header file (DigitalSignal.h) is shown in Listing 4-5.

```
class DigitalSignal
{

 public:
   DigitalSignal( );
   DigitalSignal( double bandWidth, double cntrFreq,
  int receiverID, double refTime, const FTime& aTime,
  double sampleRate, double SNR );
   virtual ~DigitalSignal( );
   double bandwidth( ) const { return m_bandWidth; }
   double sampleRate( ) const { return m_sampleRate; }
   double snr( ) const { return m_SNR; }
   double refTime( ) const { return m_refTime; }
   double cntnrFreq( ) const { return m_cntrFreq; }
   double clockError( ) const { return m_spClockError; }
   void setClockError( double error ) { m_spClockError = error; }
   Vec3 sensorPos( ) const { return m_sensorPos; }
   void setSensorPos( const Vec3& p ) { m_sensorPos = p; }
   Vec3 sensorVel( ) const { return m_sensorVel; }
   void setSensorVel( const Vec3& v ) { m_sensorVel = v; }
   FTime time( ) const { return m_time; }
   void setTime( const FTime& aTime ) { m_time = aTime; }

 protected:
   // Bandwidth of the system that created this signal.
   double m_bandWidth;
   // Center freqency of the signal.
   double m_cntrFreq;
   // The travel time
   double m_refTime;
   // time that the signal was created
   FTime m_time;
   // Unique ID of satellite which received signal
   int m_receiverID;
   // Sample rate (Hz).
   double m_sampleRate;
   // Signal to noise ratio of the system that created this
   // signal.
   double m_SNR;
   double m_spClockError;
   Vec3 m_sensorPos;
   Vec3 m_sensorVel;


};// end class DigitalSignal
```

Listing 4-5 Header file for class DigitalSignal.

4.2.1.6 Class: InfoFDOA_Meas

The class **InfoFDOA_Meas** inherits from **FDOA_Meas** and **CInfoObject**. Inheritance from **CInfoObject** gives **InfoFDOA_Meas** the ability to be inspected by the user (**CInfoObject** inherits from **CFlexObject**) and eligibility to be added to a container class derived from **InfoCollection**. Inheritance from **FDOA_Meas** incorporates the FDOA measurement data fields and functions. Note that multiple inheritance for **InfoFDOA_Meas** has kept the measurement class **FDOA_Meas** separate from the MFC® hierarchy. The header file (InfoFDOA_Meas.h) for this class is shown in Listing 4-6.

```
class InfoFDOA_Meas : public CInfoObject, public FDOA_Meas
{
  DECLARE_FLEX_SERIAL( InfoFDOA_Meas);
public:
   InfoFDOA_Meas( );
   InfoFDOA_Meas( const FDOA_Meas& aFDOA);
   virtual ~InfoFDOA_Meas();
// Override CSimObject virtuals
   virtual CObList* GetAttributes();
   virtual CObList* GetReferences();
   virtual BOOL ReferencesResolved();
   virtual void Serialize( FlxArchive& anArc );
   BOOL isEqualTo( const InfoFDOA_Meas*);
// Override GObject virtual
   BOOL isEqual( const CInfoObject*) const;
   int compare( const CInfoObject* ) const;
   unsigned long hash() const { return 0;}
   void printOn( ostream&) const {;}
};// end class InfoFDOA_Meas
```

Listing 4-6 Header file for class InfoFDOA_Meas.

4.2.1.7 Class: InfoTDOA_Meas

The class **InfoTDOA_Meas** inherits from **TDOA_Meas** and **CInfoObject**. Inheritance from **CInfoObject** gives **InfoTDOA_Meas** the ability to be inspected by the user (**CInfoObject** inherits from **CFlexObject**) and eligibility to be added to a container class derived from **InfoCollection**. Inheritance from **TDOA_Meas** incorporates the TDOA measurement data fields and functions. Note that multiple inheritance for **InfoTDOA_Meas** has kept the measurement class **TDOA_Meas** separate from the MFC® hierarchy. The header file (InfoTDOA_Meas.h) for this class is shown in Listing 4-7.

```
class InfoTDOA_Meas : public CInfoObject, public TDOA_Meas
{
  DECLARE_FLEX_SERIAL( InfoTDOA_Meas);

  public:
   InfoTDOA_Meas( );
   InfoTDOA_Meas( const TDOA_Meas& aTDOA);
   virtual ~InfoTDOA_Meas();

   // Override CSimObject virtuals
   virtual CObList* GetAttributes();
   virtual CObList* GetReferences();
   virtual BOOL ReferencesResolved();
   virtual void Serialize( FlxArchive& anArc );
   BOOL isEqualTo( const InfoTDOA_Meas*);

   // Override GObject virtual
   BOOL isEqual( const CInfoObject*) const;
   int compare( const CInfoObject* ) const;
   unsigned long hash() const { return 0;}
   void printOn( ostream&) const {;}

};// end class InfoTDOA_Meas
```

Listing 4-7 Header file for class InfoTDOA_Meas.

4.2.1.8 Class: InfoGeoLocation

The class **InfoGeoLocation** inherits from **GeoLocation** and **CInfoObject**. Inheritance from **CInfoObject** gives **InfoGeoLocation** the ability to be inspected by the user (**CInfoObject** inherits from **CFlexObject**) and eligibility to be added to a container class derived from **InfoCollection**.

Inheritance from **GeoLocation** incorporates the **GeoLocation** data fields and functions. Note that multiple inheritance for **InfoGeoLocation** has kept the measurement class **GeoLocation** separate from the MFC® hierarchy. The header file (InfoGeoLocation.h) for this class is shown in Listing 4-8.

```
class InfoGeoLocation : public CInfoObject, GeoLocation
{
  DECLARE_FLEX_SERIAL( InfoGeoLocation);
public:
   InfoGeoLocation( );
   InfoGeoLocation( GeoLocation& gl );
   virtual ~InfoGeoLocation();
//  Override CSimObject virtuals
   virtual CObList* GetAttributes();
   virtual CObList* GetReferences();
   virtual BOOL ReferencesResolved();
   virtual void Serialize( FlxArchive& anArc );
   BOOL isEqualTo( const InfoGeoLocation*);
//  Override GObject virtual
   BOOL isEqual( const CInfoObject*) const;
   int compare( const CInfoObject* ) const;
   unsigned long hash() const { return 0;}
   void printOn( ostream&) const {;}
};// end class InfoGeoLocation
```

Listing 4-8 Header file for class InfoGeoLocation.

4.2.1.9 *Class: InfoDigitalSignal*

The class **InfoDigitalSignal** inherits from **DigitalSignal** and **CInfoObject**. Inheritance from **CInfoObject** gives **InfoDigitalSignal** the ability to be inspected by the user (**CInfoObject** inherits from **CFlexObject**) and eligibility to be added to a container class derived from **InfoCollection**. Inheritance from **DigitalSignal** incorporates the **DigitalSignal** data fields and functions. Note that multiple inheritance for **InfoDigitalSignal** has kept the measurement class **DigitalSignal** separate from the MFC® hierarchy. The header file (InfoDigitalSignal.h) for this class is shown in Listing 4-9.

```
class InfoDigitalSignal : public CInfoObject, public DigitalSignal
{
  DECLARE_FLEX_SERIAL( InfoDigitalSignal);
public:
   InfoDigitalSignal( );
   InfoDigitalSignal( double bandWidth, double cntrFreq,
     int receiverID, double refTime, const FTime& aTime,
     double sampleRate, double SNR);
   virtual ~InfoDigitalSignal( );
//  Override CSimObject virtuals
   virtual CObList* GetAttributes( );
   virtual CObList* GetReferences( );
   virtual BOOL ReferencesResolved( );
   virtual void Serialize( FlxArchive& anArc );
   BOOL isEqualTo( const InfoDigitalSignal*);
// Override GObject virtual
   BOOL isEqual( const CInfoObject*) const;
   int compare( const CInfoObject* ) const;
   unsigned long hash( ) const { return 0;}
   void printOn( ostream&) const {;}
};// end class InfoDigitalSignal
```

Listing 4-9 Header file for class InfoDigitalSignal.

4.2.1.10 *Class: InfoCollection*

The class **InfoCollection** is an information object container. Objects that are descendents of the class **CInfoObject** can be contained within this class. The header file (InfoCollection.h) for this class is shown in Listing 4-10.

```
class InfoCollection : public CFlexObject
{
 DECLARE_FLEX_SERIAL( InfoCollection)

 public:
  InfoCollection();
  virtual ~InfoCollection();

  // Generic public manipulator functions
  void add( CInfoObject* value);
  void remove( CInfoObject* value);
  CInfoObject* at(int i);
  unsigned size( ) const { return m_contents.size(); }
  BOOL isEmpty();
  void removeAll();

  // Serializle the objects in this container.
  void Serialize( FlxArchive& anArc );
  OrderedCltn m_contents;

  class InfoNode : public GObject
  {
    public:
    // Override the required virtuals from GObject
    int compare( const GObject&) const;
    BOOL isEqual( const GObject&)const;
    unsigned long hash() const { return 0;}
    void printOn( ostream&) const {;}
    // Object to be contained
    CInfoObject* m_infoObj;

  };// end class InfoCollection::InfoNode

};// end template class InfoCollection
```

Listing 4-10 Header file for class InfoCollection.

4.2.1.11 *Class: FDOACntnr*

The class **FDOACntnr** is a derived class of **InfoCollection**. The **FDOACntnr** class is designed to hold objects of type **InfoFDOA_Meas**. **FDOACntnr** objects are designed to be manipulated by the user. The header file (FDOACntnr.h) for this class is shown in Listing 4-11.

```
class FDOACntnr : public InfoCollection
{
 DECLARE_FLEX_SERIAL( FDOACntnr)
 public:
  FDOACntnr( );
  ~FDOACntnr();
  void removeAllFDOAs();
  // Override CSimObject virtuals
  virtual CObList* GetAttributes();
  virtual void Serialize( FlxArchive& anArc );
}; // end class FDOACntnr
```

Listing 4-11 Header file for class FDOACntnr.

4.2.1.12 *Class: TDOACntnr*

The class **TDOACntnr** is a derived class of **InfoCollection**. The **TDOACntnr** class is designed to hold objects of type **InfoTDOA_Meas**. **TDOACntnr** objects are designed to be manipulated by the user. The header file (TDOACntnr.h) for this class is shown in Listing 4-12.

```
class TDOACntnr : public InfoCollection
{
 DECLARE_FLEX_SERIAL( TDOACntnr)

 public:
  TDOACntnr( );
  ~TDOACntnr( );

  // Override CSimObject virtuals
  virtual CObList* GetAttributes();
  virtual void Serialize( FlxArchive& anArc );

  // Public manipulator functions for TDOA_Meas
  void  addTDOA( InfoTDOA_Meas* value);
  void  removeTDOA( InfoTDOA_Meas* value);
  InfoTDOA_Meas*  nextTDOA();
  BOOL isEmptyTDOACntnr();
  void removeAllTDOAs();
};// end class TDOACntnr
```

Listing 4-12 Header file for class TDOACntnr.

4.2.1.13 *Class: TrackCntnr*

The class **TrackCntnr** is a derived class of **InfoCollection**. The **TrackCntnr** class is designed to hold objects of type **InfoGeoLocation**. **TrackCntnr** objects are designed to be manipulated by the user. The header file (TDOACntnr.h) for this class is shown in Listing 4-13.

```
class TrackCntnr : public InfoCollection
{
 DECLARE_FLEX_SERIAL( TrackCntnr)

 public:
  TrackCntnr( );
  ~TrackCntnr( );

  // Override CSimObject virtuals
  virtual CObList* GetAttributes();
  virtual void Serialize( FlxArchive& anArc );
  // Public manipulator functions for DigitalSignals
  void  addTrack( InfoGeoLocation* value);
  void  removeTrack( InfoGeoLocation* value);
  InfoGeoLocation*  nextTrack( );
  BOOL  isEmptyTrackCntnr( );
  void removeAllTracks();

};// end class TrackCntnr
```

Listing 4-13 Header file for class TrackCntnr.

4.2.1.14 *Class: SignalCntnr*

The class **SignalCntnr** is a derived class of **InfoCollection**. The **SignalCntnr** class is designed to hold objects of type **InfoDigitalSignal**. **SignalCntnr** objects are designed to be manipulated by the user. The header file (SignalCntnr.h) for this class is shown in Listing 4-14.

```
class SignalCntnr : public InfoCollection
{
 DECLARE_FLEX_SERIAL( SignalCntnr)
public:
  SignalCntnr( );
  ~SignalCntnr( );
// Override CSimObject virtuals
  virtual CObList* GetAttributes();
  virtual void Serialize( FlxArchive& anArc );
// Public manipulator functions for DigitalSignals
  void  addSignal( InfoDigitalSignal* value);
  void  removeSignal( InfoDigitalSignal* value);
  InfoDigitalSignal*  nextSignal();
  BOOL  isEmptySignalCntnr();
  void removeAllSignals();
};// end class SignalCntnr
```

Listing 4-14 Header file for class SignalCntnr.

4.2.2 Class: GeoLocator

Class **GeoLocator** inherits from **CFlexObject** and **SimObject** and is responsible for processing the FDOA and TDOA measurements to determine the estimated position(s) of the transmitter(s). **GeoLocator** contains references to three objects that are satisfied by the user. These references are to a **TDOACntnr** (mandatory), a **TrackCntnr** (mandatory), and a **FDOACntnr** (optional). TDOA and FDOA measurements are take from the attached **TDOACntnr** and **FDOACntnr** objects, processed by the geolocation algorithms (see Appendix A) and the estimated target position(s) are placed in the attached **TrackCntnr** for inspection by the user. The class diagram is shown in Figure 4-8, and Listing 4-15 contains the header file (GeoLocator.h) for this class.

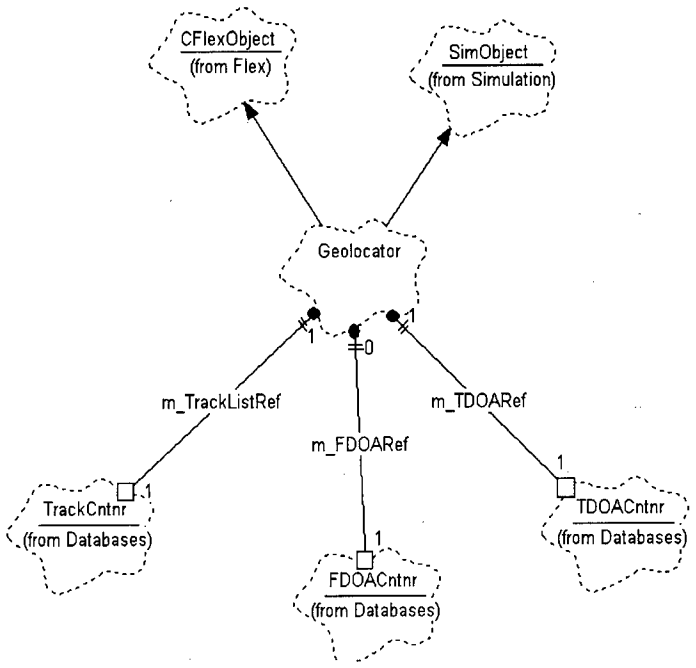


Figure 4-8 Class diagram for the class GeoLocator.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// A Geolocator processes TDOA and FDOA measurements from
// multiple receivers to create estimates of emitter
// locations.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Forward declarations...
class TDOACntnr;
class FDOACntnr;
class TrackCntnr;
class DSymMatrix;
class DoubleMatrix;
class Geolocator : public CFlexObject, public SimObject
{
  DECLARE_FLEX_SERIAL( Geolocator)
public:
   Geolocator( );
   virtual ~Geolocator();
// Override CFlexObject virtuals
   CObList* GetAttributes( );
   CObList* GetReferences( );
   BOOL ReferencesResolved( );
   void Serialize( FlxArchive& anArc );
   CObject* copy( );
   void flexInit( );
// Available TDOA and FDOA measurements are processed.
   void process();
private:
// A reference to a TDOA container. (Set via user interface.)
   TDOACntnr* m_TDOARef;
   //  A reference to an FDOA container. (Set via user interface.)
   FDOACntnr* m_FDOARef;
   // The current geolocation estimate.
   GeoLocation* m_geoLoc;
   // A reference to the set of output geolocation estimates.
   // (Set via user interface.)
   TrackCntnr* m_TrackListRef;
// Private member functions...
// Solve for the geolocation estimate using marquardt's
   // algorithm.
   void doMarquardt( GeoLocation& geoLocEst,
      const TDOA_Set& tdoaSet,
      const FDOA_Set& fdoaSet,
      const DSymMatrix& tdoaInvCov );
// The the contribution of the the TDOA measurents to the
   // convariance matrix and gradient vector.
   void addTdoaContrib( const GeoLocation& geoLocEst,
       const TDOA_Set& tdoaSet,
  const DSymMatrix& tdoaCov,
  DoubleMatrix& grad,
  DSymMatrix& locEstInvCov );
void addFdoaContrib(const GeoLocation& geoLocEst,
       const FDOA_Set& fdoaSet,
        DoubleMatrix& grad,
       DSymMatrix& locEstInvCov );
double computeFdoaChiSqrContrib( const GeoLocation& geoLocPos,
                        const FDOA_Set& fdoaSet );
double computeChi2( const GeoLocation& geoLocEst,
       const TDOA_Set& tdoaSet,
              const FDOA_Set& fdoaSet,
       const DSymMatrix& tdoaCov );
DoubleMatrix computeDeltaTs( const Vec3& geoLocPos,
       const TDOA_Set& tdoaSet );
typedef Event<Geolocator> FlxEvent;

};// end class Geolocator
```

Listing 4-15 Header file for class GeoLocator.

4.2.3 Matrix Classes

Several matrix classes were developed to support the geolocation algorithms. In particular, a matrix inverse is required to compute the Chi-Square metric used by the Marquardt optimization algorithm. Since the covariance matrices are symmetric, the computation of the inverse can be simplified by decomposing the matrix to triangular form using Cholesky decomposition. Then the inverse can be computed quickly via back substitution [Pre94]. The classes **DoubleArray**, **DoubleMatrix**, and **DSymMatrix** implement the basic array and matrix operators including indexing, assignment, and matrix multiplication. The class **CholeskyD** holds the result of the Cholesky decomposition of a **DSymMatrix** and provides the back substitution algorithm.

The **DSymMatrix** class employs an internal symmetric matrix storage scheme to reduce redundant memory use. The design of the matrix classes is shown in Figure 4-9. The interfaces for the **DSymMatrix** and **CholeskyD** classes are shown in Listing 4-16. Listing 4-17 shows an example use of these classes to compute a matrix inverse and the quadratic form of the chi-square.

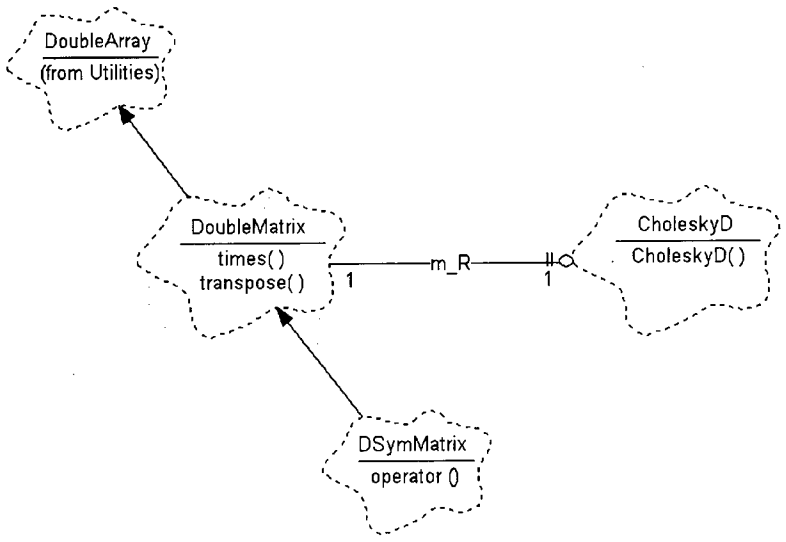


Figure 4-9 Class diagram for the matrix classes.

```
class DSymMatrix : public DoubleMatrix
{
 public:
 DSymMatrix( unsigned n );
 DSymMatrix( const DSymMatrix& m );
 DSymMatrix( const DoubleMatrix& m );
 ~DSymMatrix();
 double& operator () ( unsigned i, unsigned j );
 const double& operator () ( unsigned i, unsigned j ) const;
 DSymMatrix& operator = ( const DSymMatrix& m );
 unsigned getIndex( unsigned i, unsigned j );
 unsigned dim() const { return m_dim; }
 unsigned rows() const { return m_dim; }
 unsigned cols() const { return m_dim; }
 // Matrix operators...
 // Post multiply...
 DoubleMatrix operator * ( const DoubleMatrix& m ) const;
 DoubleMatrix product( const DoubleMatrix& m) const { return (*this)*m; }
 // Pre-multiply...
 friend DoubleMatrix operator * ( const DoubleMatrix& m, const DSymMatrix & s );
 void printOn( ostream& s );
 protected:
 unsigned m_dim;
 UShortArray m_index;
}; // end class DSymMatrix
class CholeskyD
{
 public:
 // Construct a Cholesky factorization of a symmetric matrix
 CholeskyD( const DSymMatrix& x );
 // Destructor...
 ~CholeskyD();
 void printOn( ostream& s );
 const DoubleMatrix& factors() { return m_R; }
 void solve( const DoubleMatrix& b, DoubleMatrix& x );
 friend DSymMatrix inverse( const CholeskyD& x );
 private:
 DoubleMatrix m_R;
};// end class CholeskyD
```

Listing 4-16 Class declaration for DSymMatrix and CholeskyD in C++.

```
#include "DSymMatrix.h"

int main()
{
DSymMatrix M;

// Read the matrix from a file...
M.readFrom( "input.dat" );

// Compute the inverse...
DSymMatrxi Minv = inverse( M );

// Print the matrix inverse...
Minv.printOn( cout );

// Now compute the Cholesky decomposition...
CholeskyD covDC( M );

// Create a vector for the chi-square...
DoubleMatrix rhs(M.rows())
rhs.readFrom( "lhs.dat" );

// Avoid the inverse by computing part of the chi-square by backsubstitution...
covDC.solve( lhs, rhs );

// Complete the the chi-square calculation...
DoubleMatrix chiSquare = transpose(lhs) * rhs;

// Print the chi-square value...
cout << chiSquare(0) << endl;

}// end main ////////////////////////////////////////////////////////////////////////////////
```

Listing 4-17 Example use of the matrix classes.

4.3 Detailed Algorithm Design

The detailed designs for the geolocation algorithms are described in the following sections. Appendix B contains the class diagrams for the relevant geolocation classes and Appendix C contains the corresponding source code.

4.3.1 Geolocation - An Overview

The use of the GPS network to provide precision location information almost anywhere on the surface of Earth is a proven navigational tool. With a GPS receiver, a person can accurately determine his geographical location. To accomplish this precision geolocation, each GPS satellite continuously transmits a signal - time synchronized among the GPS satellites. A GPS receiver intercepts these signals from three or more GPS satellites, and based on the received times of the signal, determines the propagation times from each active satellite. The propagation times are converted to distances between the GPS receiver and the active satellites. When coupled with precise knowledge of the GPS satellite positions at the time of signal reception, the distance between the GPS receiver and each active GPS satellite defines a single point (the location of the GPS receiver) and the surface of Earth.

Relative to determining the position of a ship for navigation purposes, the system is more than adequate. However, relative to the mission of search and rescue, the system has a major flaw. A small ship lost at sea might very well be able to locate itself, but without adequate communications has no way of transmitting its position to rescue services. One way to overcome this problem is to reverse the geolocation process - that is, instead of having a constellation of GPS satellites transmitting a unique

time synchronized signal to be received by a GPS receiver, a GPS emitter could transmit a signal to be received by each visible GPS satellite. By computing the propagation times from transmitter to the visible GPS satellites, the distances from the transmitter to the visible satellites can be determined.

For this procedure to work, the propagation times from the transmitter to each satellite must be determined. However, without elaborate and expensive timing, the exact time of signal transmission cannot be determined; without knowing the time of signal transmission, the propagation time cannot be computed. But is not necessary to determine the propagation time from the transmitter to each satellite. Knowing the difference in propagation time from the transmitter to two satellites also provides information regarding the location of the transmitter. Since correlation techniques can be used to compute the TDOA without exact knowledge of the time of transmission, TDOA techniques overcome the unknown time of transmission problem. Each TDOA measurement restricts the transmitter to lie on a two-dimensional surface in three dimensional space. With two independent TDOA measurements, the transmitter is restricted to lie on a line (the intersection of the two surfaces). If the transmitter is further restricted to lie on Earth's geodetic ellipsoid of revolution (such as would be the case for a ship at sea), the transmitter's location will be the intersection of the TDOA line with the geodetic ellipsoid of revolution. Normally this line will intersect with the ellipsoid of revolution at two points; however, this dual location ambiguity can usually be resolved on the basis of line-of-sight visibility. That is, only one of the two points will be visible to the subset of satellites that collected the transmitted signal. If the transmitter is not restricted to lie on Earth's geodetic ellipsoid of revolution (such as an airplane), a third independent TDOA measurement is required to locate the transmitter.

For stationary or very slowly moving transmitters and rapidly moving satellites, information regarding the transmitter location can be obtained from the frequency of arrival of the transmitted signal at each satellite. Motions between the transmitter and the satellites introduce a Doppler shift into the received frequency relative to the transmitted frequency. The amount of frequency shift is, among other things, a function of the transmitter's location. As with time measurements, the exact transmitter frequency is usually unknown. Thus, to exploit received frequency information to locate the transmitter, the FDOA of the signal as collected at two satellites is used instead of the absolute frequency shift.

For stationary transmitters restricted to lie on Earth's geodetic ellipsoid of revolution, it is often possible to locate the transmitter by using a combination of TDOA and FDOA measurements. The advantage of using both TDOA and FDOA instead of TDOA alone is the reduction in the number of satellites required to "see" the signal. A single pair of satellites is often sufficient for TDOA/FDOA geolocation, whereas for TDOA geolocation alone, three satellites must resolve the transmitted signal. The drawback of TDOA/FDOA geolocation relative to TDOA geolocation alone is that FDOA techniques give erroneous results if the transmitter is moving.

In terms of FlexSim++, Geolocation refers to the location of a surface emitter using a constellation of satellite receivers with GPS orbits.

A track position is determined by calculating an error ellipsoid for a stationary emitter as a function of signal strength, various error sources, and data fusion techniques (including TDOA and FDOA).

4.3.2 TDOA Processing

TDOAs are computed within the **TDOAGenerator** from times of arrival which are computed in the class Environment (see Figure 4-10).

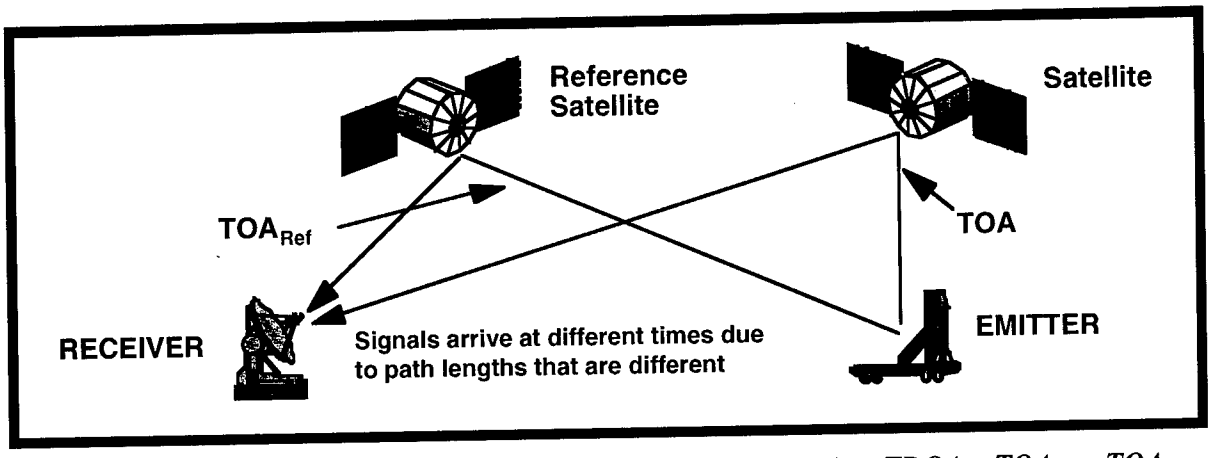


Figure 4-10 Illustration of the time of arrival phenomena. Note that, TDOA = TOA_{ref} - TOA.

The Time Of Arrival (TOA) is stored in an analog signal which is created by the **Environment**. The GPS receiver and signal processor collect the analog signal, process the signal to create a digital signal, and place the digital signal in the **SignalCntnr**.

The **TDOAGenerator** polls the **SignalCntnr** periodically, extracts signals when present, and processes the digital signals to produce a TDOA. TDOAs are computed by the **TDOAGenerator** in the following manner:

1. Poll the **SignalCntnr** every deltaT (update rate for the **TDOAGenerator**).
2. If the **SignalCntnr** is empty repeat 1., else place the N signals in a local data structure and label the first signal as the reference signal.
3. Iterate through the list starting with the second signal and subtract the time of arrival for the reference signal from the time of arrival at the current signal to produce a TDOA. Apply measurement errors to the TDOA and store the result in the **TDOACntnr**. Repeat this step N-1 times.

4.3.3 FDOA Processing

FDOAs are computed within the **FDOAGenerator** from shifted frequencies which are computed in the class Environment (see Figure 4-11). The shifted frequency is computed as follows:

Let, f_e be the emitted frequency

P_e be the position vector of the emitter

V_e be the velocity vector of the emitter

P_r be the position vector of the receiver

V_r be the velocity vector of the receiver

C be the speed of light

r be the range rate

f_s be the shifted frequency

<vec1, vec2>be the dot product of vec1 with vec2

Then, $d = | P_e - P_r |$

$r = < (P_e - P_r), (V_e - V_r) > / (C*d)$

$f_s = f_e*(1 - r)$

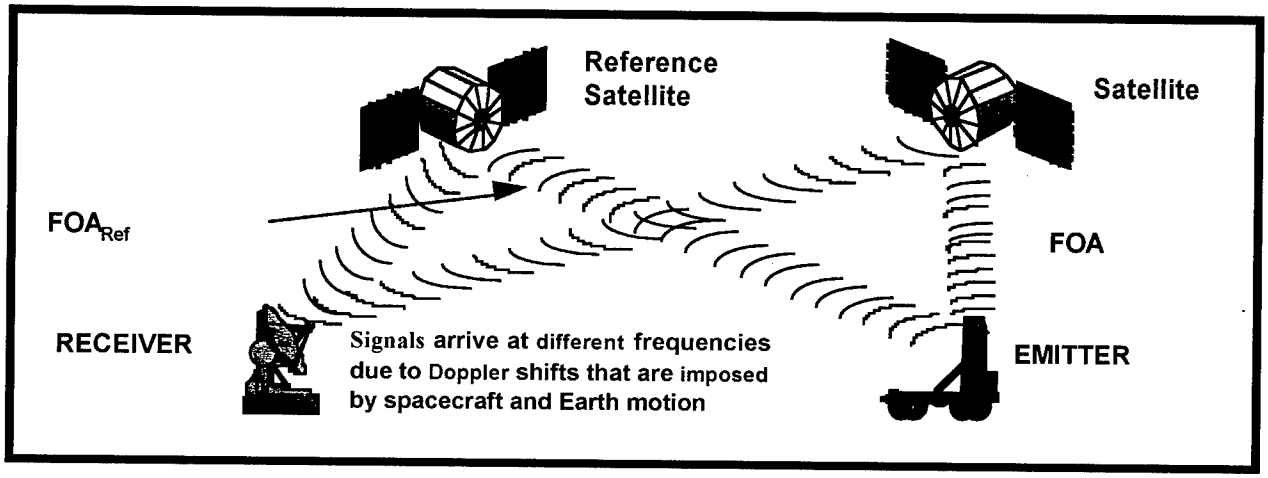


Figure 4-11 Illustration of the frequency of arrival phenomena. Note that, FDOA = FOA_{ref} - FOA.

The shifted frequency is stored in an analog signal which is created by the **Environment**. The GPS receiver and signal processor collect the analog signal, process the signal to create a digital signal, and place the digital signal in the **SignalCntnr**. The **FDOAGenerator** polls the **SignalCntnr** periodically, extracts signals when present, and processes the digital signals to produce an FDOA. FDOAs are computed by the **FDOAGenerator** in the following manner:

1. Poll the **SignalCntnr** every deltaT (update rate for the **FDOAGenerator**).
2. If the **SignalCntnr** is empty repeat 1., else place the N signals in a local data structure and label the first signal as the reference signal.
3. Iterate through the list starting with the second signal and subtract the shifted frequency of the reference signal from the shifted frequency of the current signal to produce an FDOA. Apply measurement errors to the FDOA and store the result in the **FDOACntnr**. Repeat this step N-1 times.

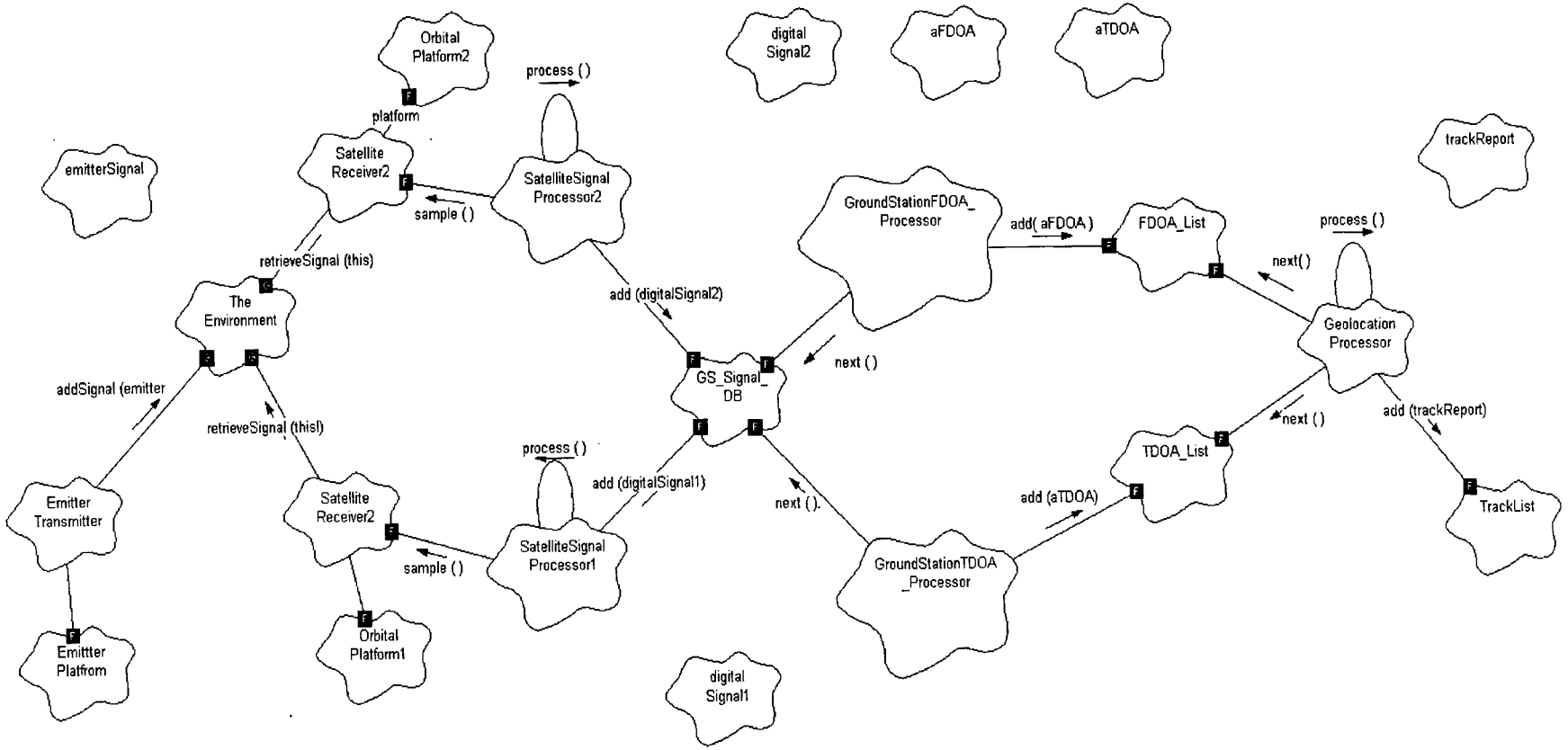


Figure 4-12. Object Scenario Diagram for the Sample Problem

4.4 Geolocation Test Environment

In order to investigate the mixed fidelity techniques discussed in earlier sections and to test the geolocation algorithms, a test environment was created. The driver models created to support the geolocation tests and the mixed fidelity investigations are discussed in the following sections. The object scenario diagram for the geolocation problem is shown in Figure 4-12.

4.4.1 Driver Models

To drive the geolocation algorithms, we created several models that combine to generate a digital signal for processing by the TDOA and FDOA generators. These models capture transmitter signal properties, environmental effects, satellite orbit dynamics, receiver characteristics and satellite signal processing capabilities.

A *FlexSim* class captures each model's parameters, interfaces, and behavior. Instances of these classes are created to model the transmitting platform and receiver satellites in a geolocation problem scenario.

4.4.1.1 Transmitter Model

The transmitter model is characterized by the bandwidth, center frequency, and power of the signal it radiates into the environment. These properties are captured by the class **Transmitter** provided by the physics library. The class **FlxTransmitter** adapts the class **Transmitter** to the *FlexSim* environment and provides a reference to the transmitter's host platform. The parameters of the **Transmitter** class are exported to the *FlexSim* environment by the **FlxTransmitter**::GetAttributes member function. At initialization time, the transmitter model adds a signal to the environment with the appropriate characteristics for possible interception by other models (e.g., a receiver model). The relationship among these classes is shown in the Booch diagram of Figure 4-13 while the interfaces are shown in Listings 4-18 and 4-19.

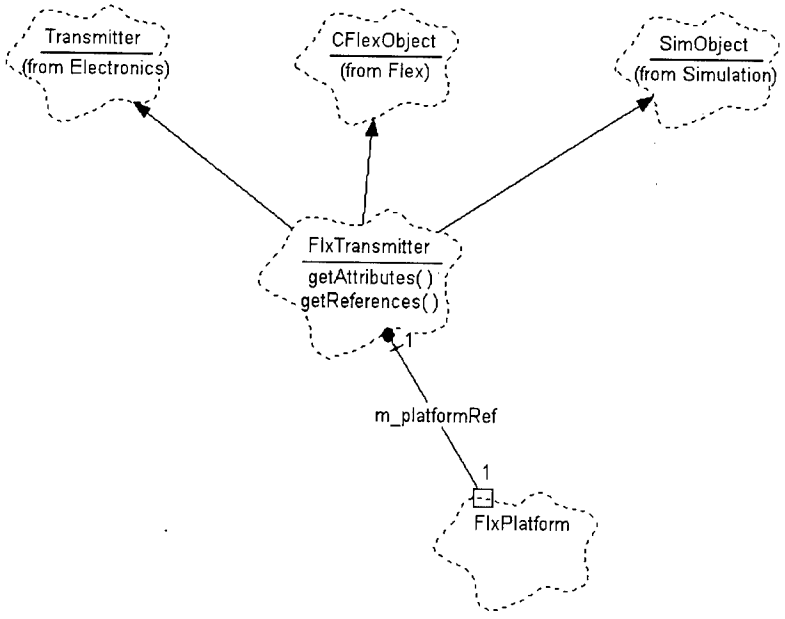


Figure 4-13 Class diagram for the transmitter model.

```
class Transmitter : public PhysObject
{
  public:
   Transmitter();
   Transmitter( const Transmitter& );
   double bandWidth() const { return m_bandWidth; }
   double centerFreq() const { return m_centerFreq; }
   double power() const { return m_power; }
   int id() const { return m_id; }
protected:
  // Bandwidth (Hz) of a signal transmitted by this receiver.
     double m_bandWidth;
// Center frequency (Hz) of this transmiiter.
   double m_centerFreq;
// Power (W) of transmitter
   double m_power;
// Unique ID for this instance of transmitter
   int m_id;
// Unique transmitter ID; as it is declared a static member, it
  // is common to all instances of class Transmitter so each
  // time an instance is constructed, it can be incremented to
  // represent a unique ID
   static int c_uniqueID;
}; // end class Transmitter
```

Listing 4-18 The Transmitter class declaration in C++.

```
class FlxTransmitter : public CFlexObject, public SimObject, public Transmitter
{
 DECLARE_FLEX_SERIAL( FlxTransmitter )
public:
   FlxTransmitter();
   FlxTransmitter( const FlxTransmitter& trans );
   virtual ~FlxTransmitter();
// Override CFlexObject virtuals
   CObList* GetAttributes();
   CObList* GetReferences();
   BOOL ReferencesResolved();
   void flexInit();
   void simulate();
   CObject* copy();
void Serialize( FlxArchive& anArc );
private:
// The platform mounting this transmitter. (Set via user
  //interface.)
   FlxPlatform *m_platformRef;
}; // end class FlxTransmitter
```

Listing 4-19 The FlxTransmitter class declaration in C++.

4.4.1.2 Receiver Model

The receiver model is characterized by the bandwidth of possible signal it can receive from the environment. No modeling of the receiver antenna beamwidth and direction is accomplished at this time. These properties are captured by the class **Receiver** provided by the physics library. The class **FlxReceiver** adapts the class **Receiver** to the *FlexSim* environment and provides a reference to the receiver's host platform. The parameters of the **Receiver** class are exported to the *FlexSim* environment by the **FlxReceiver**::GetAttributes member function. When the **Receiver**::sample member function is invoked, the receiver model retrieves signals from the environment that have been placed there by a transmitter. The relationship among these classes is shown in the Booch diagram of Figure 4-14 while the interfaces are shown in Listings 4-20 and 4-21.

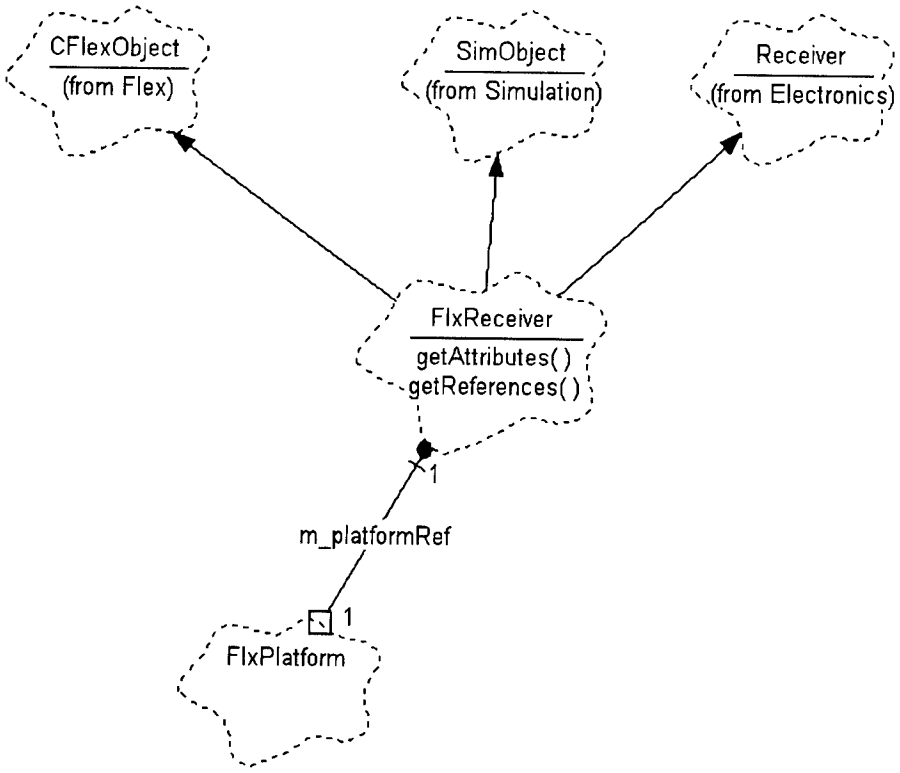


Figure 4-14 Class diagram for the receiver model.

```
class Receiver : public PhysObject
{
 public:
  Receiver();
  Receiver( const Receiver& r );

  // Return an analog signal that is the result of sampling
  // the environment for signal within the bandwidth and
  // beamwidth of this Receiver.
  AnalogSignal sample( Environment& );

  unsigned id() const { return m_receiverID; }

 protected:

  // Receiver bandwidth.
  double m_bandWidth;

  // Unique ID for this instance of receiver
  int m_receiverID;

  // Unique receiver ID; as it is declared a static member, it
  //  is common to all instances of class Receiver so each
  //  time an instance is constructed, it can be incremented to
  //  represent a unique ID
  static int c_uniqueID;

}; // end class Receiver
```

Listing 4-20 The Receiver class declaration in C++.

```
class FlxReceiver : public CFlexObject, public SimObject, public Receiver
{
DECLARE_FLEX_SERIAL( FlxReceiver )

public:
FlxReceiver();
FlxReceiver( const FlxReceiver& r);
virtual ~FlxReceiver();


// Override CFlexObject virtuals
virtual CObList* GetAttributes();
virtual CObList* GetReferences();
BOOL ReferencesResolved();
void flexInit();
void simulate();
CObject* copy();

void Serialize( FlxArchive &anArc );
FlxPlatform* getPlatformRef() { return m_platformRef;}

private:

// The platform mounting this receiver. (Set via user interface.)
FlxPlatform* m_platformRef;

}; // end class FlxReceiver
```

Listing 4-21 The FlxReceiver class declaration in C++.

4.4.1.3 Environment Model

For the geolocation problem, the primary purpose of the environment model is the manipulation of signals as they are transmitted into and received out of Earth environment. Signals that are added to the environment are stored in their original form along with their point of origin. This allows any transformations due to travel distance, path attenuation, and Doppler shift to be applied at the time that a receiver samples the environment. Currently, the signal phase and Doppler shifts are given respectively by:

$$\phi = d / C$$

$$\Delta f = (\vec{P}_t - \vec{P}_r) * (\vec{V}_t - \vec{V}_r) / (C * d)$$

Where d is the distance between the satellite and the transmitter, C is the speed of light and P and V are the transmitter and satellite velocities. Transmitters do not have to have any knowledge of the receivers that may intercept their signals. When a transmitter ceases to transmit its signal, it removes the signal from the environment. Receivers are returned a modified copy of the original signal that reflects the applied environmental effects or no signal at all if the Earth-satellite geometry prevents interception. The environment model is implemented by the class **Environment** provided by the physics library. The class **FlxEnvironment** adapts the class **Environment** to *FlexSim*. Transmitted signals are represented by the class **EnvSignal**. The **Environment** class maintains a list of all environment signals as a class (static) variable. A single, global **Environment** object is created for access by all the models within *FlexSim*. The design of the environment model is illustrated in Figure 4-15 and the interface is shown in Listing 4-22.

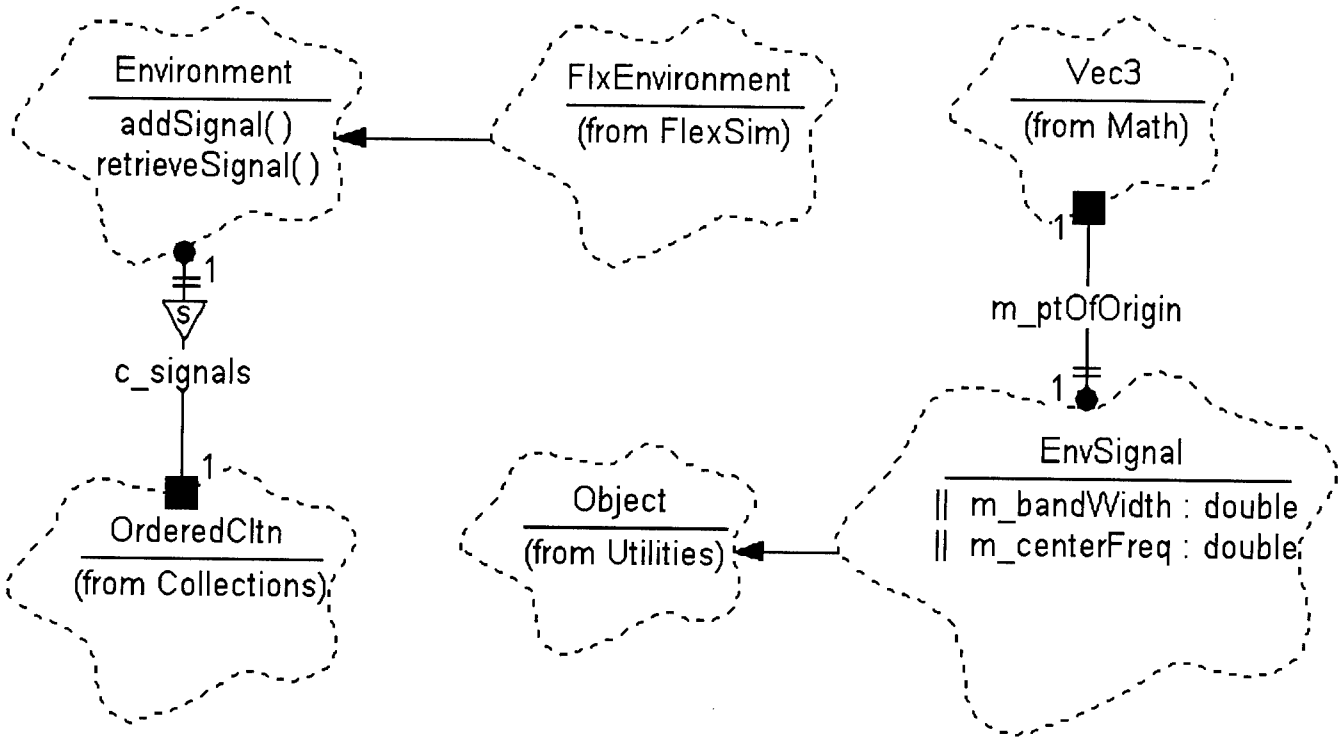


Figure 4-15 The Environment Model Classes.

```
class EnvSignal : public Object
{
 public:
 EnvSignal( const Transmitter& );
 ~EnvSignal();
 double centerFreq() const { return m_centerFreq;}
 void setCenterFreq( double f ) { m_centerFreq = f; }
 double phase() const { return m_phase; }
 void setPhase( double p ) { m_phase = p; }
 double bandWidth() const { return m_bandWidth;}
 Vec3 originPos() const { return m_ptOfOrigin;}
 Vec3 originVel() const { return m_velOfOrigin;}
 int transmitterID() const { return m_transmitterID; }
// Override Object virtuals
 int isEqual( const GObject&) const;
 int compare( const GObject& ) const;
 unsigned long hash() const { return 0;}
 void printOn( ostream&) const {;}

 private:
// The signal bandWidth (Hz)
 double m_bandWidth;
// The center frequency of the signal. (Hz)
 double m_centerFreq;
// The phase of the signal (secs)
 double m_phase;
// The signal power (dB)
 double m_power;
// A unique identifier for the source transmitter. Allows
// signal to be deleted from the environment when
// transmitter is silent.
 int m_transmitterID;
// The ECR position of the source transmitter.
 Vec3 m_ptOfOrigin;
// The ECR velocity vector of the source transmitter.
 Vec3 m_velOfOrigin;
};// end class EnvSignal

typedef OrderedCltn EnvSignalList;
class Environment
{
 public:
 ~Environment();
 const FTime& epochTime() { return m_epochTime; }
 const void epochTime(FTime& t) { m_epochTime = t; }
// Adding signal to the environment means that a
// transmitter is current transmitting the signal. If
// transmitters cease transmitting, the signal should be
// removed.
 void addSignal( EnvSignal* aSignal );
// All signals within the beamwidth, and bandwidth of the
// receiver are modified according to the relative geometry
// of the emitter and the receiver including atmospheric
// attenuation effects and added to the returned list of
// signal detectable by that receiver.
 EnvSignalList retrieveSignal( const Receiver& receiver );
// The signal with the given transmitter ID is removed.
 void removeSignal( const Transmitter& transmitter );
 protected:
 FTime m_epochTime;
// A list of all signals currently transmitted into the
// environment by transmitters.
 static EnvSignalList c_signals;

};// end class Environment
```

Listing 4-22 Environment and EnvSignal class declarations in C++.

4.4.1.4 *Signal Processor Model*

The signal processor model is parameterized by the sample rate (integration time) and effective SNR (Signal to Noise Ratio) of the satellite receiver electronics. The model is implemented directly in FlexSim by the **SignalProcessor** class. A reference to receiver and to an output signal container is also provided. The signal processor calls the receiver **sample** function to create an analog signal. This analog signal is then converted to a digital signal and added to the output signal container. Figure 4-16 shows the Booch class diagram for the **SignalProcessor** class and the interface is shown in Listing 4-23.

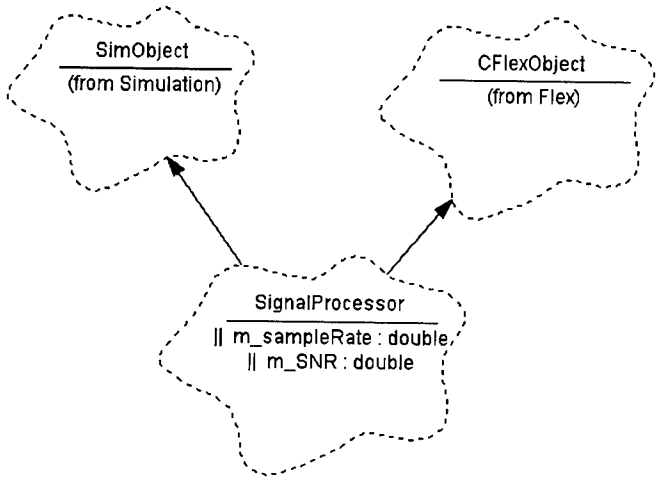


Figure 4-16 Class Diagram for the SignalProcessor Class.

```
class SignalProcessor : public CFlexObject, public SimObject
{
DECLARE_FLEX_SERIAL( SignalProcessor )
public:
 SignalProcessor();
 ~SignalProcessor();
 SignalProcessor( const SignalProcessor& sp);
 // Override CSimObject virtuals
 CObList* GetAttributes();
 CObList* GetReferences();
 BOOL ReferencesResolved();
 void flexInit();
 void simulate();
 CObject* copy();
 void Serialize( FlxArchive& anArc );
protected:
 void Process();
private:
 // Sample Rate ( Hz). ( Set via the user interface)
 double m_sampleRate;
 // Signal-to-noise ratio. ( Set via the user inteface)
 double m_SNR;
 // A reference the collection containing the output digital
 // signals. (Set via user interface.)
 SignalCntnr* m_outputRef;
 // Reference to a receiver. (Set via the user interface)
 FlxReceiver* m_receiverRef;
}; // end class SignalProcessor
```

Listing 4-23 SignalProcessor class declaration in C++.

4.4.1.5 *Orbital Platform Model*

The orbital platform model simulates the dynamics of a satellite in Earth orbit. The class **FlxOrbPlatform** implements the model and is derived from the **FlxPlatform** class to allow any object's **FlxPlatform** reference to be assigned to a **FlxOrbPlatform** object. An orbital platform periodically updates its position by propagating its orbital state to the current simulation clock time. Asynchronous updates of the platform state are allowed through the **propagateTo** member function. The orbital dynamics are modeled as planar, elliptical trajectories using orbital elements. [Bat71].

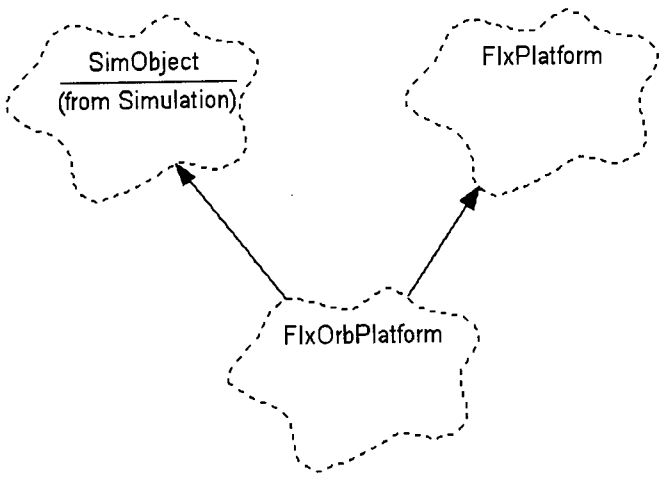


Figure 4-17 Diagram of the FlxOrbPlatform Class.

```
class FlxOrbPlatform : public FlxPlatform, public SimObject
{
DECLARE_FLEX_SERIAL(FlxOrbPlatform)
public:

 FlxOrbPlatform();
 FlxOrbPlatform( const FlxOrbPlatform& );
 FlxOrbPlatform( const FTime&  aTime,
 const FlxOrbState& aState );
 ~FlxOrbPlatform();
 void Serialize( FlxArchive& );
 CObList* GetAttributes();
 void flexInit();
 virtual Vec3 getPos();
 virtual CObject* copy();
 void propagateTo(const FTime& aTime);
 void simulate();
 void log();
 virtual Vec3 getVel();
 virtual double angularVel();

 private:
 double logRate;
 FlxOrbState state;
 typedef Event<FlxOrbPlatform> FlxEvent;

};// end class FlxOrbPlatform
```

Listing 4-24 FlxOrbPlatform class declaration in C++.

4.4.1.6 *TDOA_Generator Model*

The **TDOA_Generator** model generates TDOA measurements from digital signals. **TDOA_Generator** contains two user-setable references (see Figure 4-18) to a **SignalCntnr** and a **TDOACntnr**.

Signals are extracted from the **SignalCntnr**, processed to produce "noisy" TDOA measurements, and the TDOA measurements are then placed in the **TDOA_Cntnr** to await processing by the **GeoLocator**. **TDOA_Generator** has an associated "flex" class, **FlxTDOA_Generator**, that handles the simulation and user interface duties keeping the model code separate from the simulation and user interface hierarchy. The separation of model code and administrative code (simulation and GUI) greatly facilitates portability and code reuse. The header file (TDOA_Generator.h) is shown in Listing 4-25.

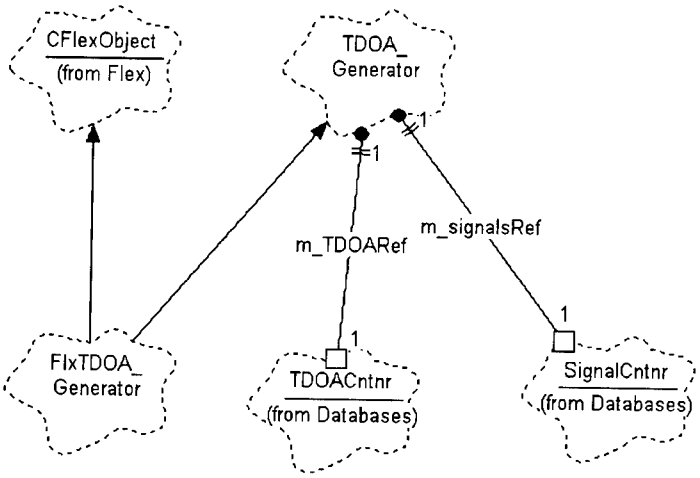


Figure 4-18 Class Diagram for the TDOA_Generator and Class FlxTDOA_Generator.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// A TDOA_Generator creates Time Difference of Arrival
// measurements from two digital signals created by the
// reception of a signal by two or more satellites.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class TDOA_Generator : public SimObject
{
public:
  TDOA_Generator();
  virtual ~TDOA_Generator();
protected:
  // Periodically iteratives throught the input signal list
  // and creates new TDOA meaurements.
  void Process();
// a random number seed for generating measurement errors
  long m_Seed;
// A reference to a collection digital signals. (Set via
  // user interface. )
  SignalCntnr *m_signalsRef;
// A reference to the the output TDOA set. (Set via user
  // interface.)
  TDOACntnr *m_TDOARef;
}; // end class TDOA_Generator
```

Listing 4-25 Header file for class TDOA_Generator.

4.4.1.7 *FDOA_Generator Model*

The **FDOA_Generator** model generates FDOA measurements from digital signals. **FDOA_Generator** contains two user-setable references (see Figure 4-19) to a **SignalCntnr** and a **FDOACntnr**. Signals are extracted from the **SignalCntnr**, processed to produce "noisy" FDOA measurements, and the FDOA measurements are then placed in the **FDOA_Cntnr** to await processing by the **GeoLocator**. **FDOA_Generator** has an associated "flex" class, **FlxFDOA_Generator**, that handles the simulation and user interface duties keeping the model code separate from the simulation and user interface hierarchy. The separation of model code and administrative code (simulation and GUI) greatly facilitates portability and code reuse. The header file (FDOA_Generator.h) is shown in Listing 4-26.

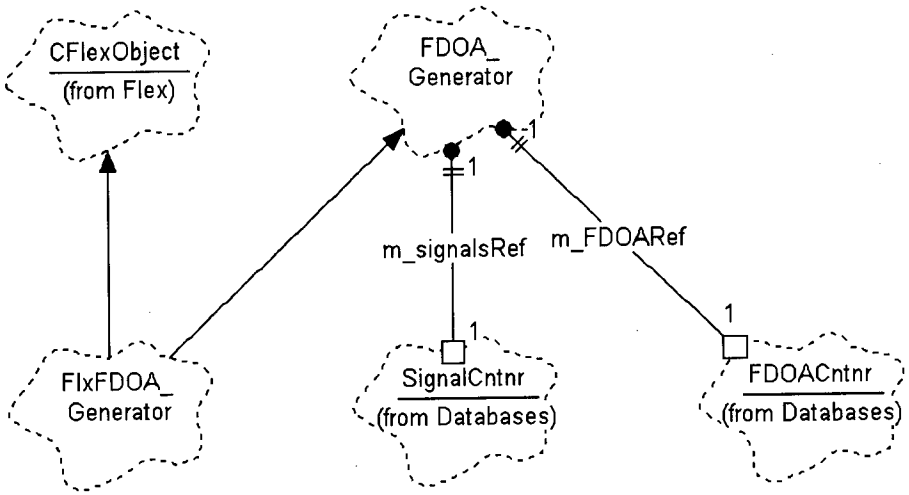


Figure 4-19 Class diagram for the FDOA_Generator and class Flx FDOA_Generator.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// A FDOA_Generator creates Time Difference of Arrival
// measurements from two digital signals created by the
// reception of a signal by two or more satellites.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
class FDOA_Generator : public SimObject
{
public:
  FDOA_Generator();
  virtual ~FDOA_Generator();

 protected:
// Periodically iteratives throught the input signal list
  // and creates new FDOA meaurements.
  void Process();
// a random number seed
  long m_Seed;
// A reference to a collection digital signals. (Set via
  // user interface. )
  SignalCntnr *m_signalsRef;
// A reference to the the output FDOA set. (Set via user
  // interface.)
  FDOACntnr *m_FDOARef;
typedef Event<FDOA_Generator> FlxEvent;

}; // end class FDOA_Generator
```

Listing 4-26 Header file for class FDOA_Generator.

4.4.2 Mixed Fidelity Modeling Options/Classes

4.4.2.1 Reuse Through Multiple Inheritance Example

One example from the sample problem of model reuse through multiple inheritance is shown in Figure 4-20. We create new class **FlxTransmitter** by multiply inheriting from the **CFlexObject** class and the external model class **Transmitter**.

Using this approach, the **Transmitter** class can be maintained in a separate library while only those services necessary for interaction with the simulation environment need be added to the derived class. Listing 4-27 shows the actual implementation in C++.

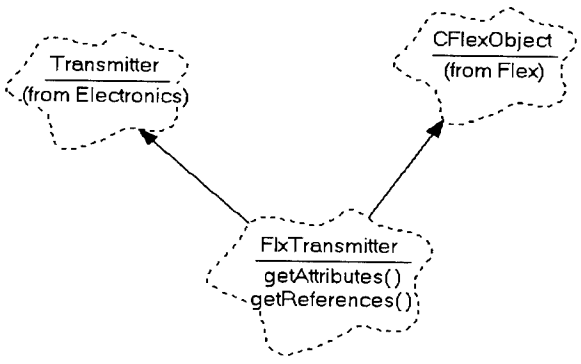


Figure 4-20 Example of multiple inheritance in the creation of the Geolocation Transmitter Model.

```
class FlxTransmitter : public CFlexObject, public SimObject, public Transmitter
{
 public:
  FlxTransmitter();
  FlxTransmitter( const FlxTransmitter& trans );
  virtual ~FlxTransmitter();
// Override CFlexObject virtuals...
  CObList* GetAttributes();
  CObList* GetReferences();
  BOOL ReferencesResolved();
  void flexInit();
  void simulate();
  CObject* copy();
void Serialize( FlxArchive& anArc );
private:
 // The platform mounting this transmitter. (Set via user
 // interface.)
  FlxPlatform *m_platformRef;
}; // end class FlxTransmitter
```

Listing 4-27 Definition of class FlxTransmitter using multiple inheritance in C++.

5. GEOLOCATION PROBLEM — ANALYSES AND RESULTS

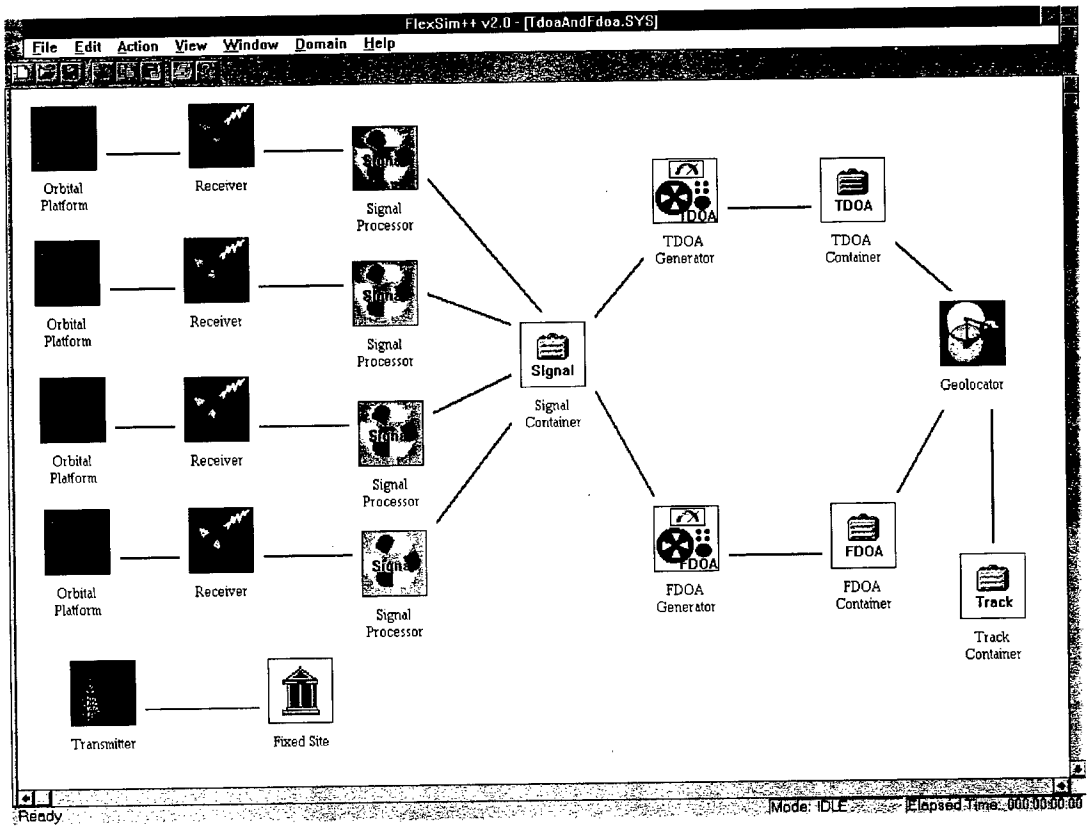


Figure 5-1 System configuration used for Monte Carlo analysis of TDOA/FDOA geolocation.

In order to test the ability of the geolocation algorithms to estimate the position of a transmitter, a modest Monte Carlo analysis was performed. The system configuration that was the basis for the Monte Carlo analysis is shown in Figure 5-1. The principle components of the system include the following:

- constellation of four GPS satellites in representative orbits (ephemeris' data shown in Table 5-1)
- single transmitter (transmitter parameters shown in Figure 5-2 and transmitter positions shown in Figure 5-3)
- position estimates based on a single TDOA and FDOA measurement per satellite

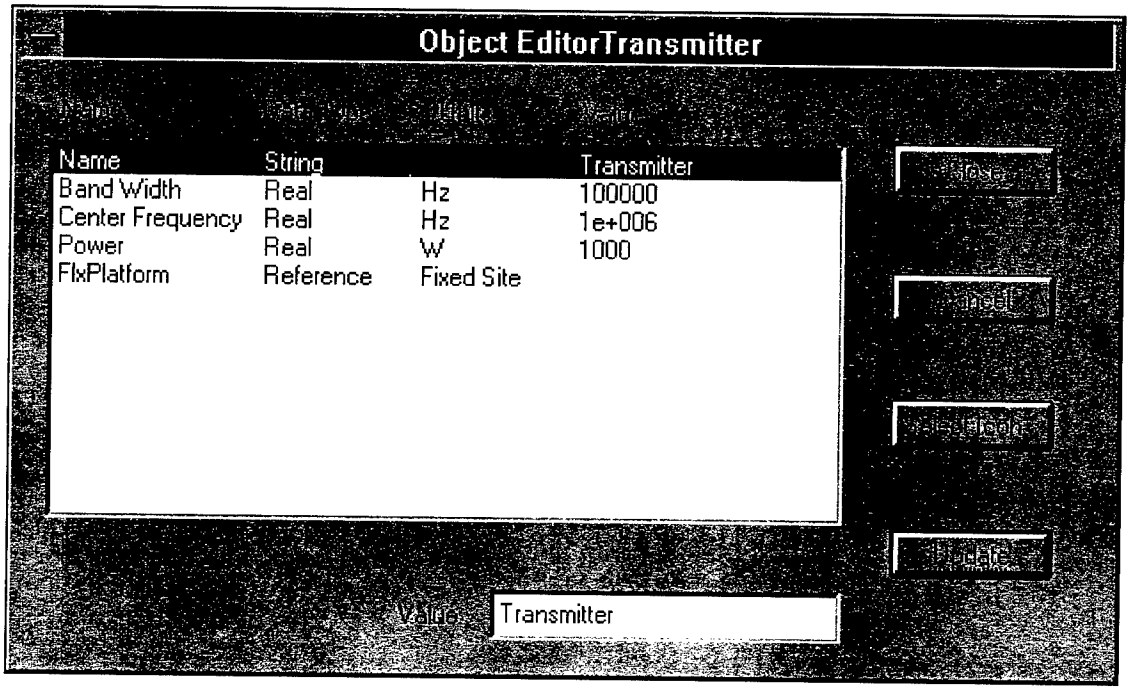
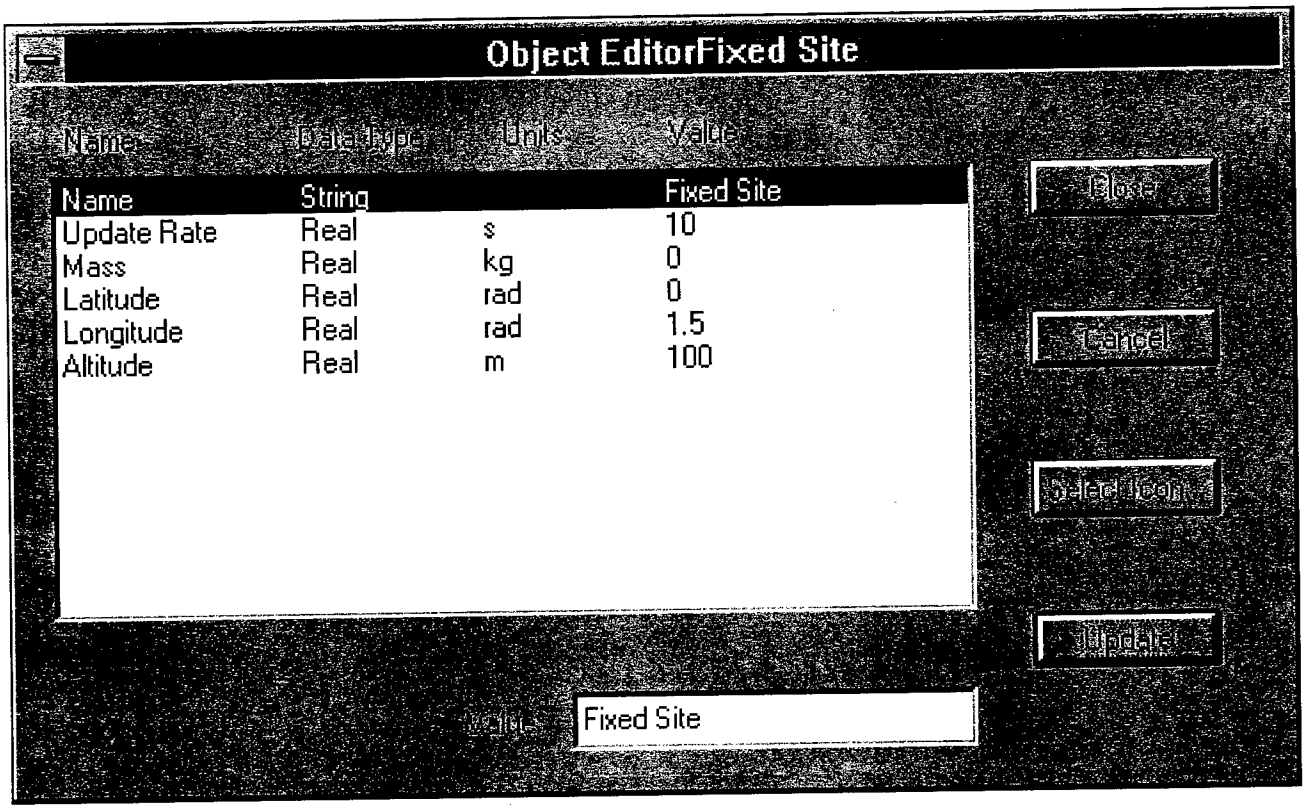


Figure 5-2 Transmitter Characteristics.



5-3 Transmitter location.

Table 5-1 Satellite Ephemeris Data

Satellite Ephemeris Data (Earth Centered Rotational (ECR) Coordinates)						
	x (m)	y (m)	z (m)	x dot (m/s)	y dot (m/s)	z dot (m/s)
Satellite 1	12836945.0	22234240.0	6868244.0	-1000.0	1012.0	-1406.0
Satellite 2	9400573.0	16282271.0	18770904.0	500.0	-1580.0	1120.0
Satellite 3	24799072.0	6644891.0	6868244.0	-500.0	-131.0	1932.0
Satellite 4	16269153.0	9393000.0	18786000.0	1000.0	1039.0	-1385.0

The simulation was run 15 times, where the random number seeds for both the TDOAGenerator and the FDOAGenerator were the only parameters changed from run to run. The results of the Monte Carlo analysis are shown in Table 5-2. The average absolute error (magnitude of the vector difference) between the estimated positions and the actual transmitter position is approximately 90 meters. The standard deviation for the estimates was nearly zero. It should be noted that the transmitter position was in a "sweet spot" with respect to the satellite positions and that other transmitter positions show greater errors. The determining factor for this geolocation scheme is the relative geometrics between the transmitters and the satellites. As such, a more comprehensive study would have investigated a larger matrix of transmitter positions with geolocation occurring at various times of the day and year with a full complement of GPS satellites. A study of this magnitude was beyond the scope of this project but could be accomplished using the simulation in its current state of development.

Table 5-2 Position estimates for each of the 15 Monte Carlo trials

Results of Monte Carlo Trials		
Latitude (radians)	**Longitude (radians)**	**Altitude (meters)**
3.26E-06	1.50000	116.3300
-2.28E-07	1.50000	99.7084
-8.14E-06	1.50001	50.4425
1.44E-06	1.50000	99.2963
4.36E-06	1.49990	126.9280
5.90E-07	1.50000	103.4750
2.05E-07	1.50000	105.8900
-6.27E-07	1.50000	92.6837
-2.88E-06	1.50000	80.8904
-1.10E-06	1.50000	93.0702
6.74E-07	1.50000	104.5140
2.76E-06	1.50000	117.6170
1.55E-06	1.50000	108.1250
-1.89E-06	1.50000	82.1709
-7.43E-06	1.50001	57.3046

6. CONCLUSIONS, LESSONS LEARNED

We have presented several promising solutions to mixed fidelity simulation. We have explored both dynamic solutions that support flexible assembly of mixed-fidelity simulations and static solutions that support implementation of mixed-fidelity models via software reuse. The use of multiple inheritance for external model reuse has been established as a viable approach and the separation of object model properties in the design of model components has been established as a design alternative. A stable set of geolocation algorithms has been created that provides a foundation for further analysis. Some of our design approaches have been implemented in software and their use demonstrated in the geolocation sample problem.

The choice of solution for a particular model or project depends not only on technical merit, but also on software reuse considerations. For example, the creation of a "fat" 6-DOF class was driven by a need to reuse large, monolithic, legacy programs. The selection of a static solution such as parameterized types may be preferred if efficiency is given priority over flexibility for the user. Other potential solutions such as interface expansion and embedded expert systems remain candidates for future work. A simulation developer needs to have a number of design approaches available and we have attempted to expand that list.

Since one of the main motives for mixed-fidelity simulation is model reuse, we have also explored design approaches that enhance the development of reusable class libraries and portable model classes. We have defined a significant set of lessons learned in the creation of reusable class libraries for modeling and simulation.

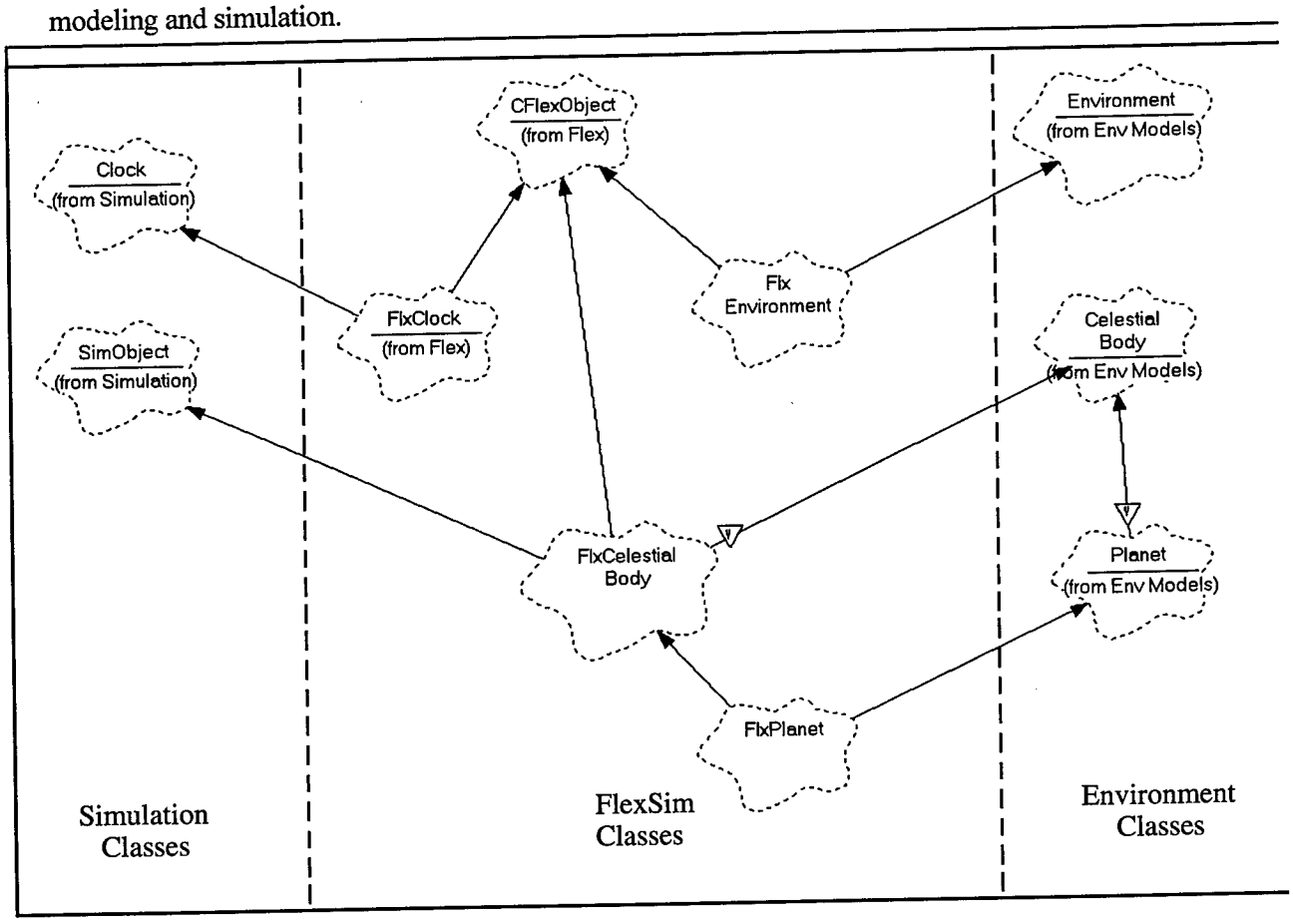


Figure 6-1 Overview of design that promotes class library reuse.

Figure 6.1 illustrates several design constructs that we employed to promote software portability and model reuse. We used multiple inheritance from both external model and simulation classes to keep their source libraries independent of the *FlexSim* software. Note that the **Planet** class had to be derived "virtual" from the base class **CelestialBody** to prevent the multiple inclusion of a **CelestialBody**

object in instances of the *FlexSim* **FlxPlanet** class. The use of virtual inheritance must be implemented apriori in the external class library to promote reuse by application developers that may not have access to that libraries source code. Also note that no path exists up the inheritance hierarchy of the external library classes to any base class in the *FlexSim* application class hierarchy. This prevents any dependence of the external library classes on the **CFlexObject** class and by inheritance on the class **CObject** from MFC®. Use of private (implementation) inheritance can also limit the dependence of application classes on proprietary class libraries like MFC®. This form of inheritance reuses the *implementation* of the windowing system without exporting that interface to the rest of the application. When this approach is used, the application depends on the interface exported by the reuse class - insulating the application from any changes to the proprietary library (changes in new versions of MFC® have already adversely affected this project) or re-implementation via another class library. The following are lessons learned for class library reuse:

- Do not declare global variables in the global name space from within a class library.
- Declare member variables "protected" to allow access by derived classes. (Remember that in doing so you are promising the users of this class that you will not change or delete this variable!)
- Use "virtual" inheritance within the class library when both the base and derived classes are to be reused. (This will prevent casting pointers down to virtually derived classes thus limiting the use of certain container class implementations.)
- Declare member functions, including destructors, as "virtual" to allow overriding wherever sensible to do so.
- Use "const" wherever that promise is kept. If a member function does not change its object, then declare it "const."

7. RECOMMENDATIONS FOR FURTHER WORK

Further work is required to explore the use of expert systems to be able to detect model inconsistencies when no interface type rules are violated. The relationships that exist between this work and simulation formalisms such as DEVS [Zei90] also need to be explored. The combination of flexible model assembly, a broad simulation formalism, a GUI environment, automated expert user support, data analysis tools, and an object oriented software development environment can achieve far better use of simulation in the support of system development and scientific research.

8. REFERENCES

[Bch93] Grady Booch, *Object oriented Analysis and Design with Applications,* Benjamin/Cummings, Redwood City, CA, 1993.

[Car95] Martin D. Carrol and Margaret A. Ellis, *Designing and Coding Reusable C++,* Addison Wesley, 1995.

[Str91] Bjarne Stroustrup, *The C++ Programming Language, 2nd Ed.* Addison Wesley, 1991.

[Wht94] Iseult White, *Using the Booch Method: A Rational Approach*, Benjamin/Cummings, Redwood City, CA, 1994.

[Can95] Cannon et al., *Development of the MICOM Infrared Seeker Analysis Tool,* Proceedings 1995 KRC Conference.

[Ho93] K. C. Ho and Y. T. Chan, *Solution and Performance Analysis of GeoLocation by TDOA,* IEEE Transactions on Aerospace and Electronic Systems, Vol 29, No. 4, October 1993.

[Otn89] Robert K Otnes, *Frequency Difference of Arrival Accuracy*, IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No.2, February 1989.

[Her96] Thomas Herring, *The Global Positioning System*, Scientific American, February 1996.

[Zei90] Bernard Zeigler, *Object oriented Simulation with Hierarchical, Modular Models*, Academic Press, 1990.

[Aan94] Aandrzej Lewandowski and Donna Calhoun, *Object oriented Framework for Dynamical Systems Modeling*, Object oriented Simulation Conference Proceedings, 1994.

[Dra81] Draper and Smith, *Applied Regression Analysis*, Wiley Inc., 1981.

[Hol92] James Holton, *An Introduction to Dynamic Meteorology*, Acadamic Press, 1992.

[Bat71] Roger R. Bate, Donald D. Mueller, Jerry E. White, *The Fundamentals of Astrodynamics,* Dover Publications, Inc., 1971.

[Pre94] Press, Veterling, Teukolsky, & Flannery, *Numerical Recipes in C*, 2nd Edition, Cambridge University Press, 1994.

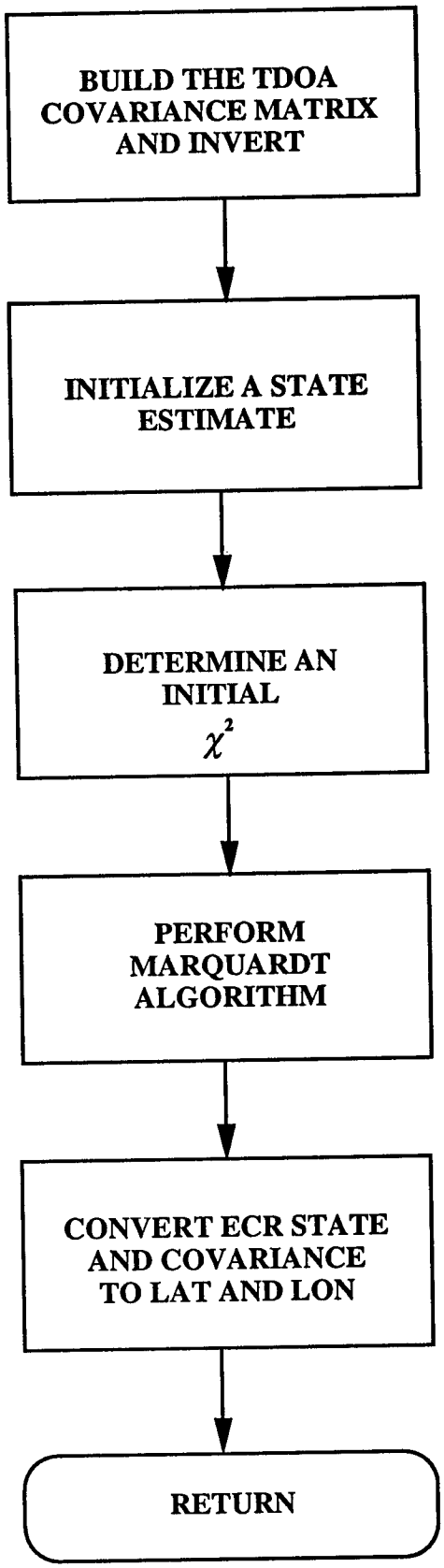
APPENDIX A

This appendix contains the detailed algorithm design flowcharts for the geolocation algorithm. These designs, in conjunction with the class designs shown in Appendix B, were used to produce the geolocation source code.

INPUTS:

$N_{\Delta T}$	Number of TDOA Measurements, = 0 IF FDOA Only
NS1( )	First Sensor Number for each TDOA/FDOA Measurement
NS2( )	Reference Sensor Number for each TDOA/FDOA Measurement
S1(6,)	ECR Position and Velocity of First Sensor for each TDOA/FDOA Measurement
S2(6,)	ECR Position and Velocity of Reference Sensor for each TDOA/FDOA Measurement
σ_{SPT}	1 SIGMA Uncertainty in TDOA Measurements Due to Signal Processing: $\frac{\sqrt{3}}{\pi \ B \ \sqrt{B \bullet T \bullet SNR}}$ Where B is the Signal Bandwidth, T is the Integration Time, and SNR is the Signal to Noise Ratio
σ_{CLOCK}	1 SIGMA Clock Error (Sec)
σ_{ATM_R}	1 SIGMA Measurement to Measurement Uncertainty in Travel Time Due to the Atmosphere
σ_{ATM_B}	1 SIGMA Residual Bias Error in Travel Time Due to the Atmosphere After Calibration
σ_{SPF}	1 SIGMA Uncertainty in FDOA Measurements Due to Signal Processing: $\frac{\sqrt{3}}{\pi \ T \ \sqrt{B \bullet T \bullet SNR}}$
$N_{\Delta F}$	Number of FDOA Measurements, = 0 IF TDOA Only
TMEAS( )	Time of each TDOA/FDOA Measurement (Arbitrarily the Time of Arrival at Reference Satellite)
R_e	Radius of the Earth (Nominal)

NSTATE	Number of State Elements (3 or 6)
C	Speed of Light (m/s)
$\Delta T( )$	TDOA Measurements
$\Delta F( )$	FDOA Measurements
$F( )$	Mean Frequency for each FDOA Measurement
MAPOPT	= 1 IF Topographical Map Used, = 0 Otherwise
σ_{MAP}	1 SIGMA Uncertainty in Topographical Map Altitude
σ_{POS}	1 SIGMA Uncertainty in each Sensor Position Component
σ_{VEL}	1 SIGMA Uncertainty in each Sensor Velocity Component


BUILD THE TDOA COVARIANCE MATRIX AND INVERT
INITIALIZE A STATE ESTIMATE
DETERMINE AN INITIAL χ^2
PERFORM MARQUARDT ALGORITHM
CONVERT ECR STATE AND COVARIANCE TO LAT AND LON
RETURN

BUILD TDOA COVARIANCE MATRIX

$$\sigma^2_{BIAS} = \left(\sigma^2_{CLOCK} + \sigma^2_{ATM_B}\right)$$

$$\sigma^2_{\Delta T_i} = \sigma^2_{SPT} + 2\sigma^2_{ATM_R} + 2\sigma^2_{BIAS}$$

$$\sigma^2_{\Delta Ti \Delta Tj} = \sigma^2_{BIAS} \quad \text{IF NS1(i) = NS1(j)}$$

$$= \sigma^2_{BIAS} \quad \text{IF NS2(i) = NS2(j)}$$

$$= -\sigma^2_{BIAS} \quad \text{IF NS1(i) = NS2(j)}$$

$$= -\sigma^2_{BIAS} \quad \text{IF NS2(i) = NS1(j)}$$

$$= 2\sigma^2_{BIAS} \quad \text{IF NS1(i) = NS1(j) and NS2(i) = NS2(j)}$$

$$= -2\sigma^2_{BIAS} \quad \text{IF NS1(i) = NS2(j) and NS2(i) = NS1(j)}$$

$$\omega = \begin{bmatrix} \sigma^2_{\Delta T_1} & \sigma_{\Delta T, \Delta T_2} & \cdots \\ & \ddots & \\ & & \sigma^2_{\Delta T_{N\Delta T}} \end{bmatrix}$$

INITIALIZE A STATE ESTIMATE

GET MEASUREMENT NUMBERS OF ALL MEASUREMENTS TAKEN AT LAST MEASUREMENT TIME T_L : $I_1, I_2 \ldots I_N$

$$T_X = \sum_{i=1}^{N} S1(1, I_i) + S2(1, I_1)$$

$$T_Y = \sum_{i=1}^{N} S1(2, I_i) + S2(2, I_1)$$

$$T_Z = \sum_{i=1}^{N} S1(3, I_i) + S2(3, I_1)$$

STATE ESTIMATE:

$$T_X = R_e \frac{T_X}{\left(T_X^2 + T_Y^2 + T_Z^2\right)^{1/2}}$$

$$T_Y = R_e \frac{T_Y}{\left(T_X^2 + T_Y^2 + T_Z^2\right)^{1/2}}$$

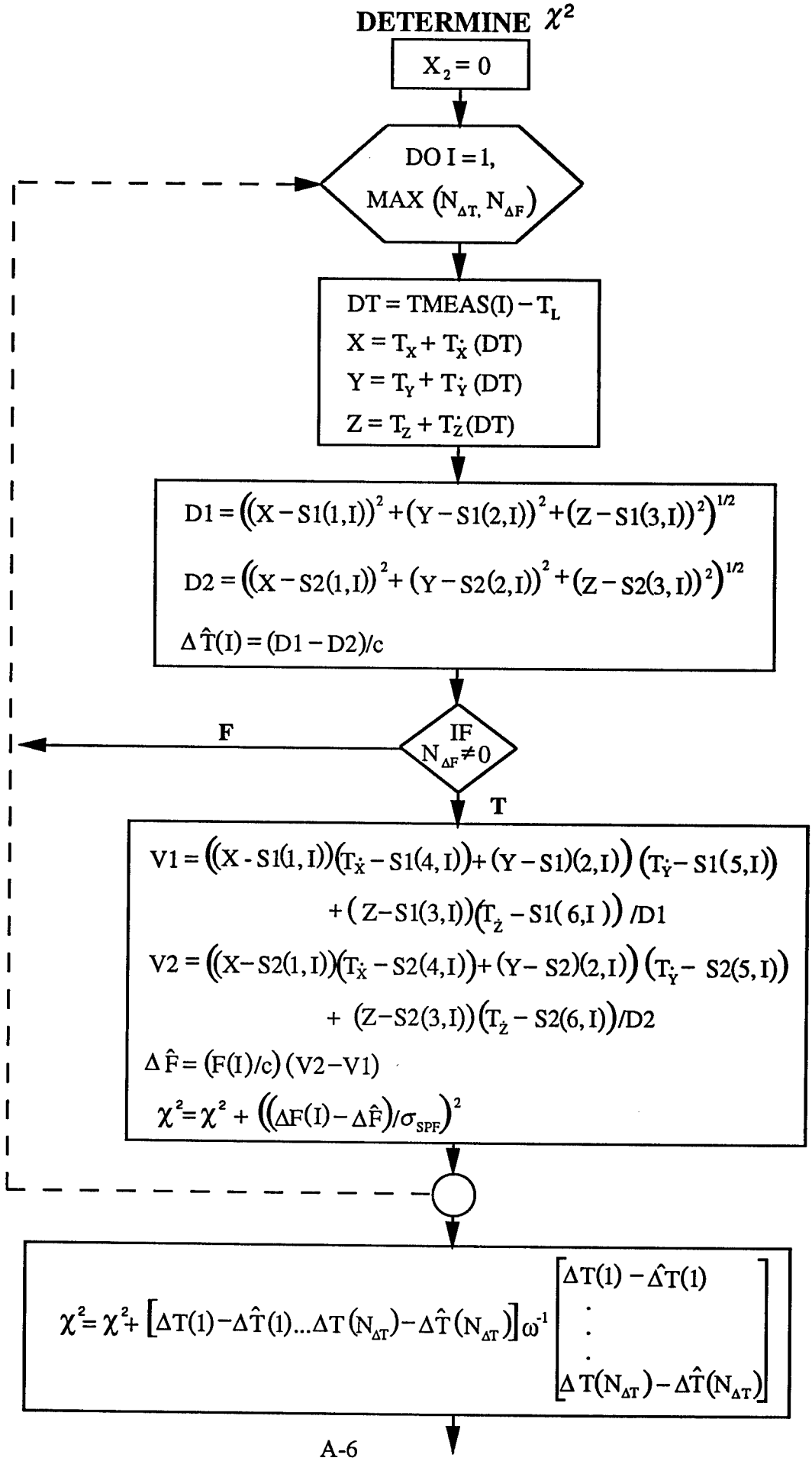
$$T_Z = R_e \frac{T_Z}{\left(T_X^2 + T_Y^2 + T_Z^2\right)^{1/2}}$$

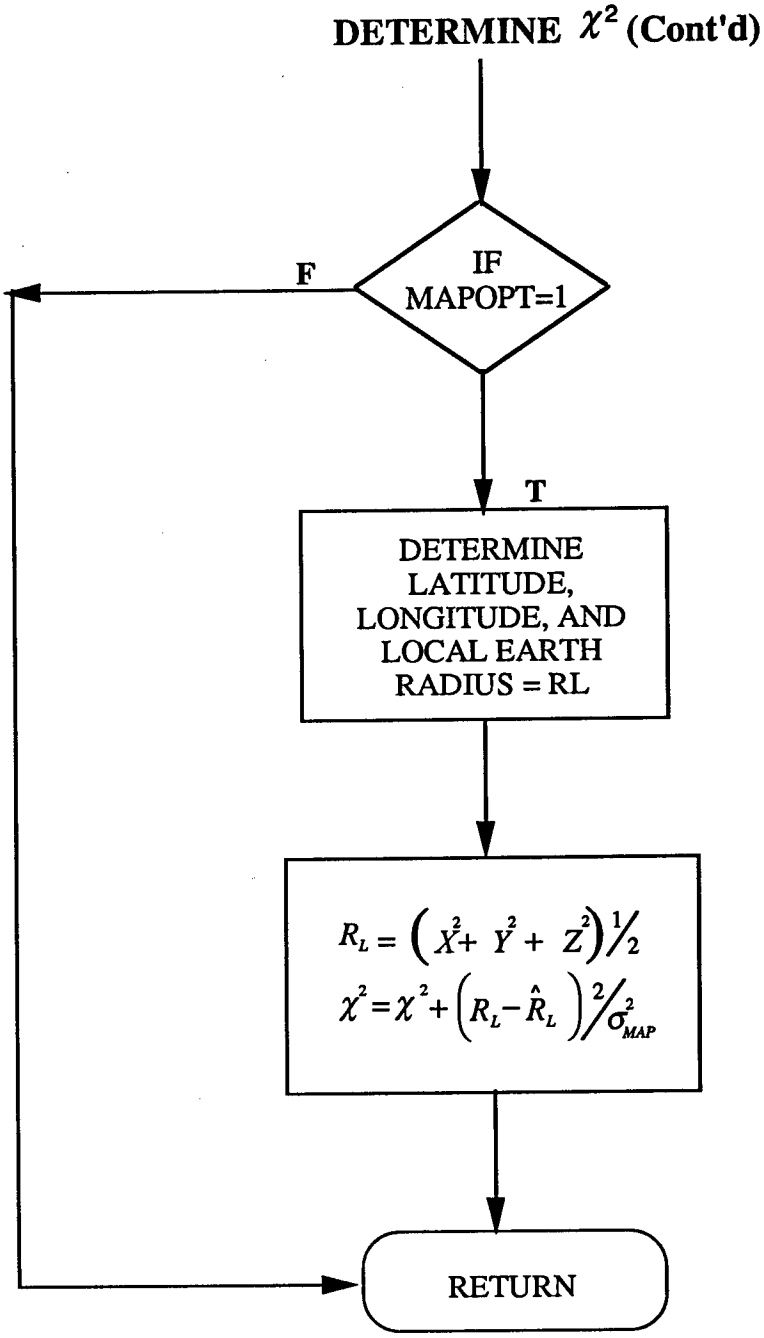
NUMBER OF SYMMETRIC COVARIANCE ELEMENTS: NCOV = 6

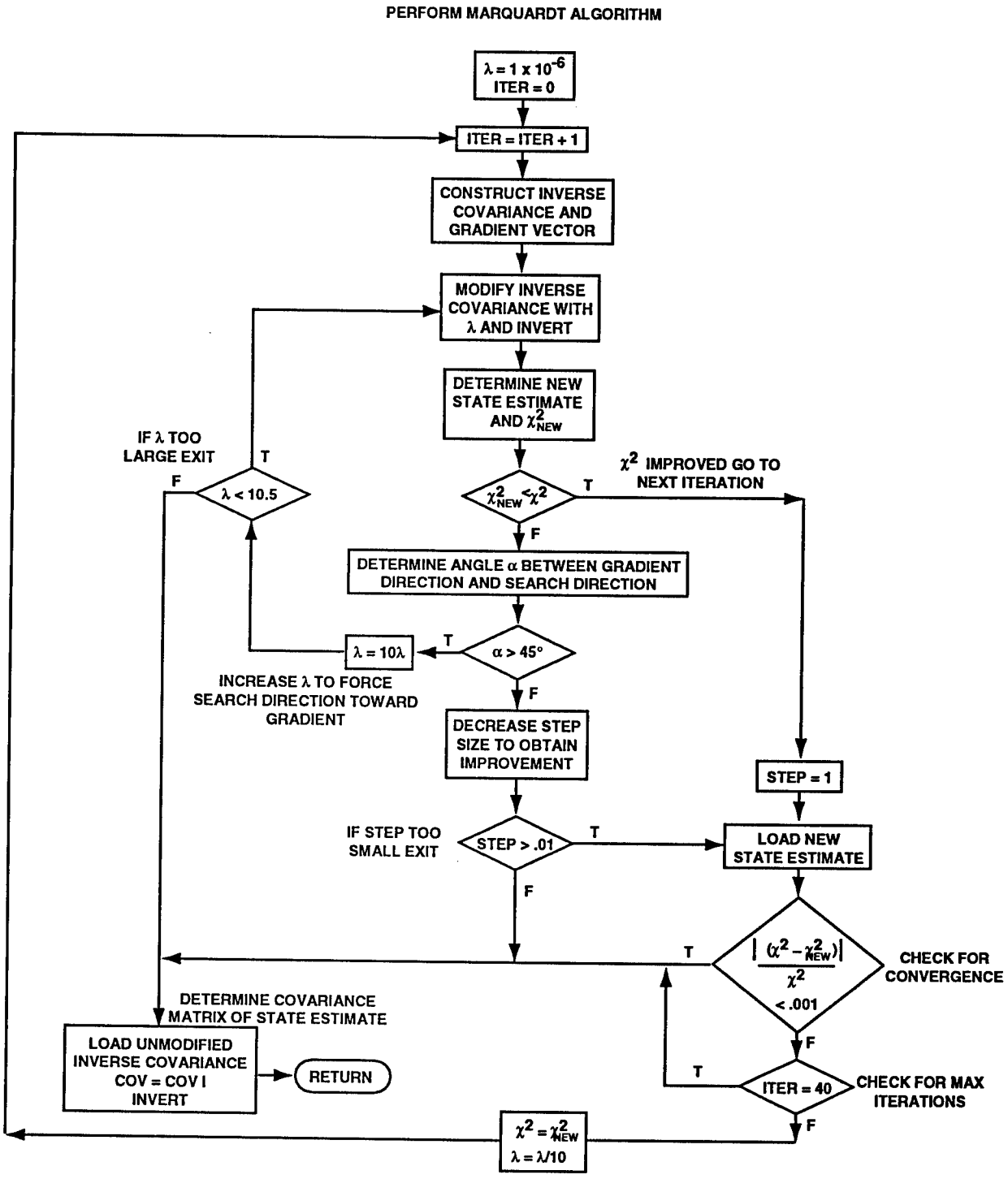
IF NSTATE = 6 THEN

$$T_{\dot{X}} = T_{\dot{Y}} = T_{\dot{Z}} = 0$$

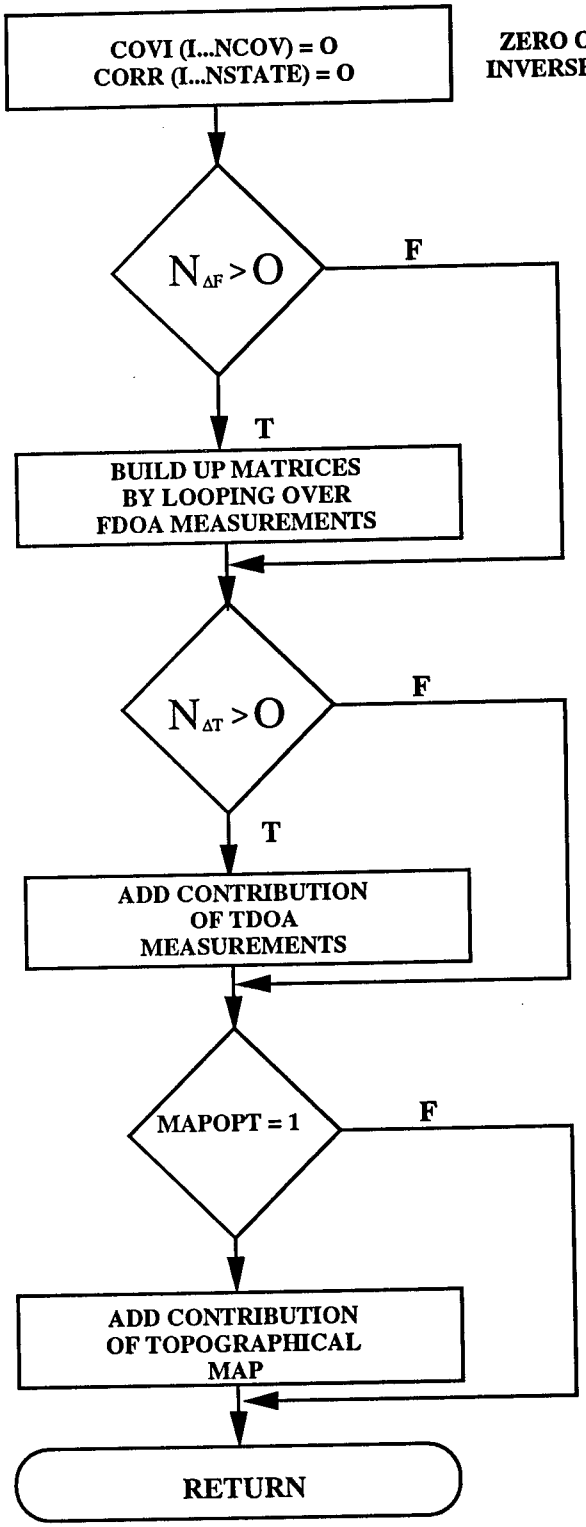
NCOV = 21


DETERMINE χ^2
$X_2 = 0$
DO I = 1,
MAX $(N_{\Delta T}, N_{\Delta F})$
DT = TMEAS(I) − T_L
X = T_X + $T_{\dot{X}}$ (DT)
Y = T_Y + $T_{\dot{Y}}$ (DT)
Z = T_Z + $T_{\dot{Z}}$ (DT)
$D1 = \left((X - S1(1,I))^2 + (Y - S1(2,I))^2 + (Z - S1(3,I))^2\right)^{1/2}$
$D2 = \left((X - S2(1,I))^2 + (Y - S2(2,I))^2 + (Z - S2(3,I))^2\right)^{1/2}$
$\Delta\hat{T}(I) = (D1 - D2)/c$
IF $N_{\Delta F} \neq 0$
F
T
$V1 = \left((X - S1(1,I))(T_{\dot{X}} - S1(4,I)) + (Y - S1)(2,I)\right)(T_{\dot{Y}} - S1(5,I)) + (Z - S1(3,I))(T_{\dot{Z}} - S1(6,I))/D1$
$V2 = \left((X - S2(1,I))(T_{\dot{X}} - S2(4,I)) + (Y - S2)(2,I)\right)(T_{\dot{Y}} - S2(5,I)) + (Z - S2(3,I))(T_{\dot{Z}} - S2(6,I))/D2$
$\Delta\hat{F} = (F(I)/c)(V2 - V1)$
$\chi^2 = \chi^2 + \left((\Delta F(I) - \Delta\hat{F})/\sigma_{SPF}\right)^2$
$\chi^2 = \chi^2 + [\Delta T(1) - \Delta\hat{T}(1) \ldots \Delta T(N_{\Delta T}) - \Delta\hat{T}(N_{\Delta T})]\omega^{-1} \begin{bmatrix} \Delta T(1) - \Delta\hat{T}(1) \\ \vdots \\ \Delta T(N_{\Delta T}) - \Delta\hat{T}(N_{\Delta T}) \end{bmatrix}$

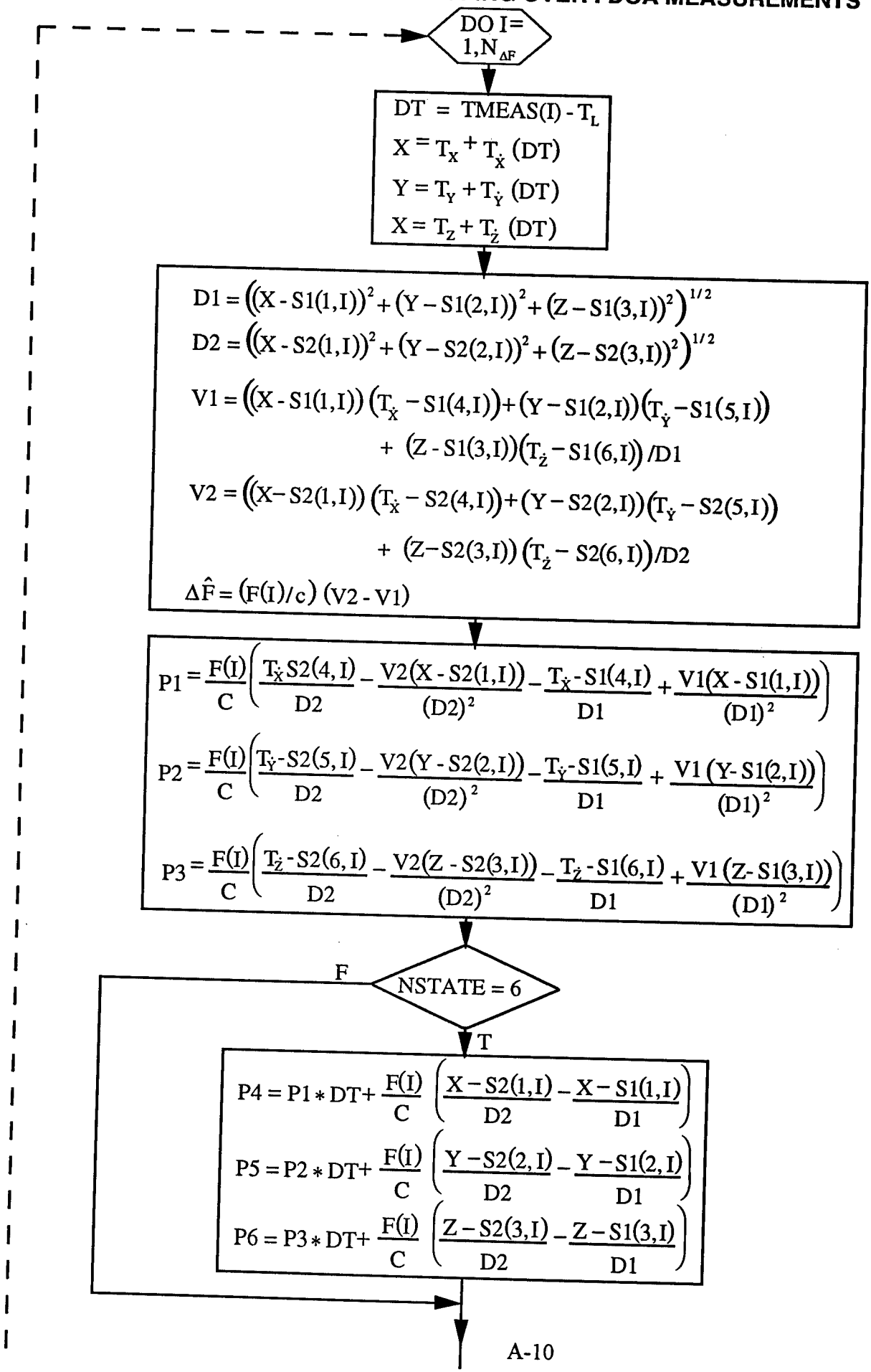

DETERMINE χ^2 (Cont'd)
IF MAPOPT=1
F
T
DETERMINE LATITUDE, LONGITUDE, AND LOCAL EARTH RADIUS = RL
$R_L = \left(X^2 + Y^2 + Z^2\right)^{1/2}$
$\chi^2 = \chi^2 + \left(R_L - \hat{R}_L\right)^2 / \sigma^2_{MAP}$
RETURN

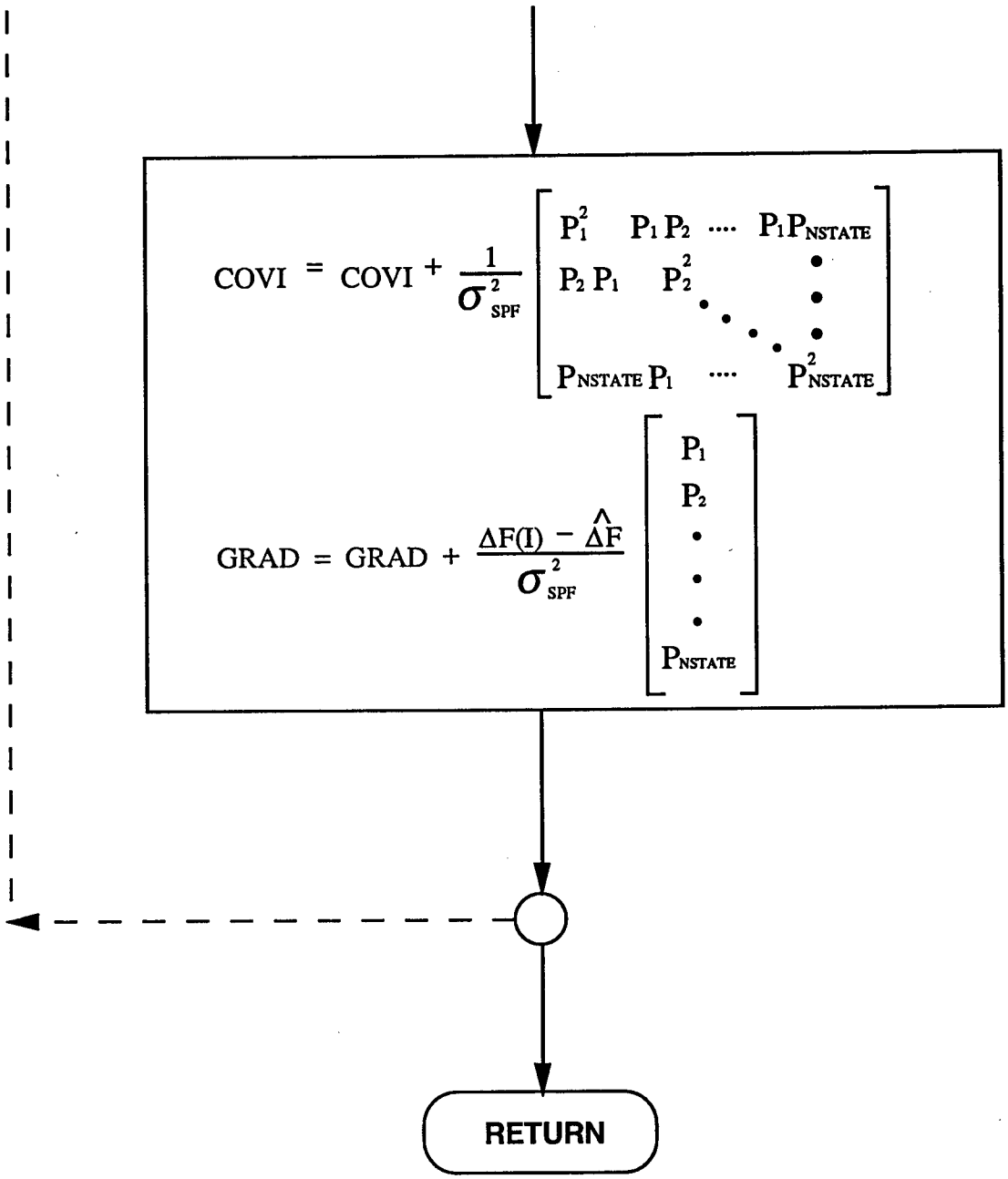

PERFORM MARQUARDT ALGORITHM
λ = 1 x 10^-6
ITER = 0
ITER = ITER + 1
CONSTRUCT INVERSE COVARIANCE AND GRADIENT VECTOR
MODIFY INVERSE COVARIANCE WITH λ AND INVERT
DETERMINE NEW STATE ESTIMATE AND χ²NEW
IF λ TOO LARGE EXIT
T
F
λ < 10.5
χ²NEW < χ²
T
χ² IMPROVED GO TO NEXT ITERATION
F
DETERMINE ANGLE α BETWEEN GRADIENT DIRECTION AND SEARCH DIRECTION
α > 45°
T
λ = 10λ
INCREASE λ TO FORCE SEARCH DIRECTION TOWARD GRADIENT
F
DECREASE STEP SIZE TO OBTAIN IMPROVEMENT
STEP = 1
IF STEP TOO SMALL EXIT
STEP > .01
T
LOAD NEW STATE ESTIMATE
F
T
|(χ² − χ²NEW)| / χ² < .001
CHECK FOR CONVERGENCE
F
T
ITER = 40
CHECK FOR MAX ITERATIONS
F
χ² = χ²NEW
λ = λ/10
DETERMINE COVARIANCE MATRIX OF STATE ESTIMATE
LOAD UNMODIFIED INVERSE COVARIANCE COV = COV I INVERT
RETURN

CONSTRUCT INVERSE CONVARIANCE AND GRADIENT VECTOR

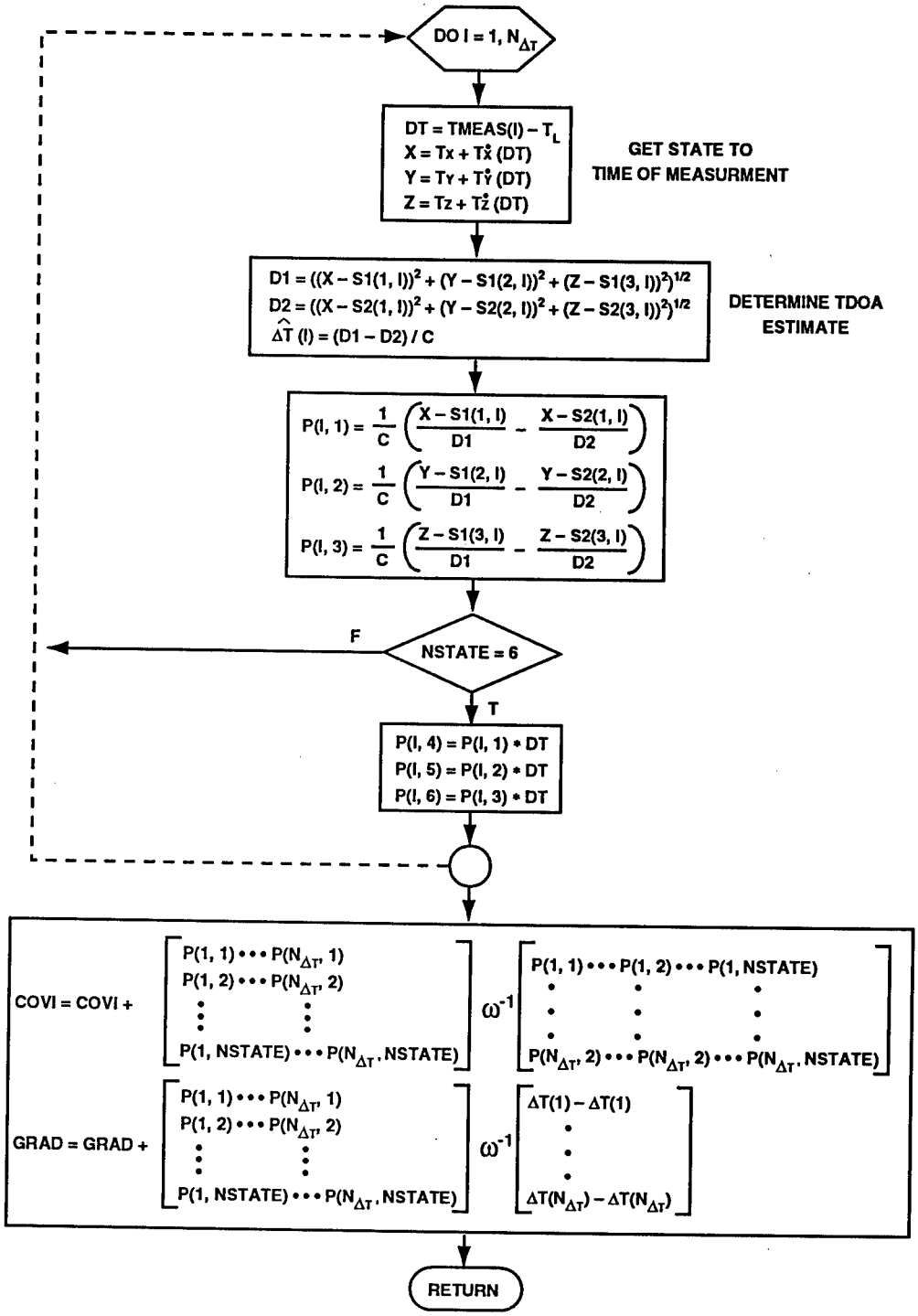


BUILD UP MATRICES BY LOOPING OVER FDOA MEASUREMENTS

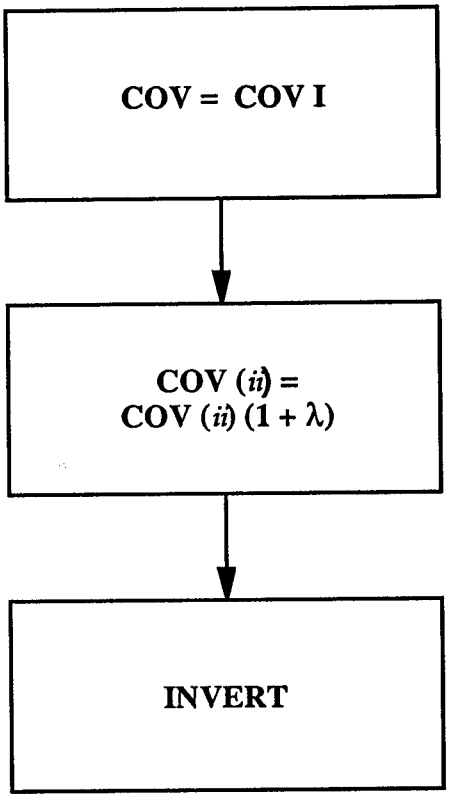



$$COVI = COVI + \frac{1}{\sigma^2_{SPF}} \begin{bmatrix} P_1^2 & P_1P_2 & \cdots & P_1P_{NSTATE} \\ P_2P_1 & P_2^2 & & \\ & & \ddots & \\ P_{NSTATE}P_1 & \cdots & & P_{NSTATE}^2 \end{bmatrix}$$
$$GRAD = GRAD + \frac{\Delta F(I) - \hat{\Delta F}}{\sigma^2_{SPF}} \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_{NSTATE} \end{bmatrix}$$
RETURN

ADD CONTRIBUTION OF TDOA MEASUREMENTS

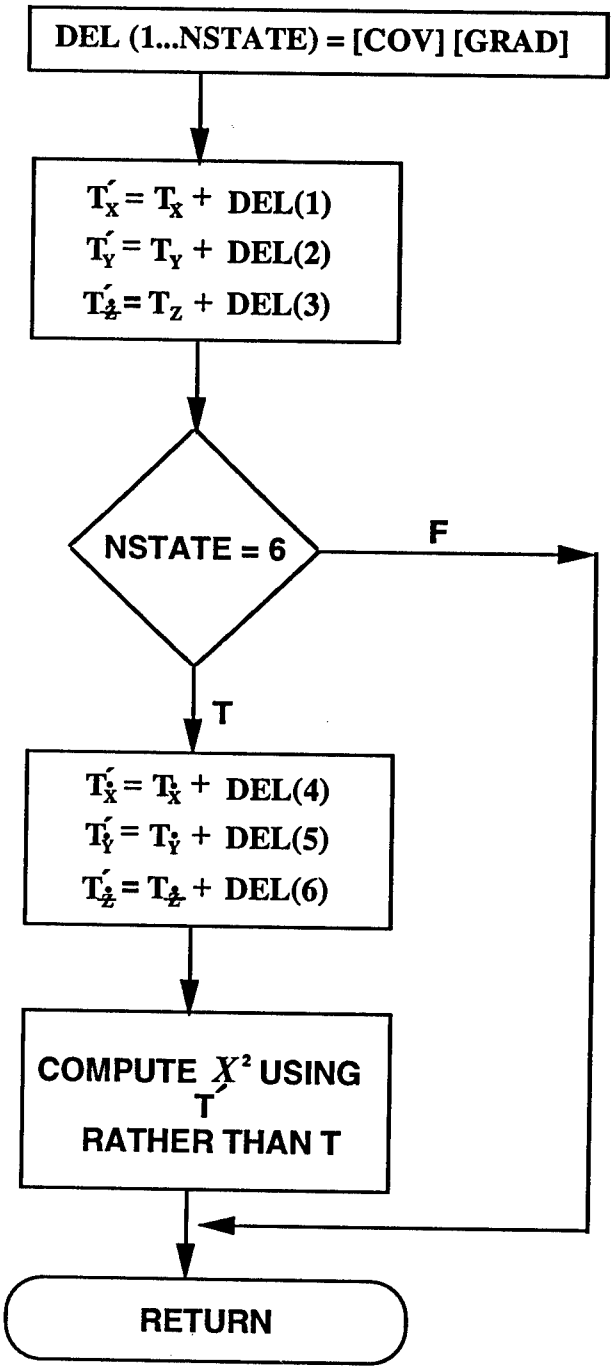


MODIFY INVERSE COVARIANCE WITH λ AND INVERT



LOAD INVERSE COVARIANCE

DETERMINE NEW STATE ESTIMATE AND X^2_{NEW}



DETERMINE ANGLE α BETWEEN GRADIENT DIRECTION AND SEARCH DIRECTION

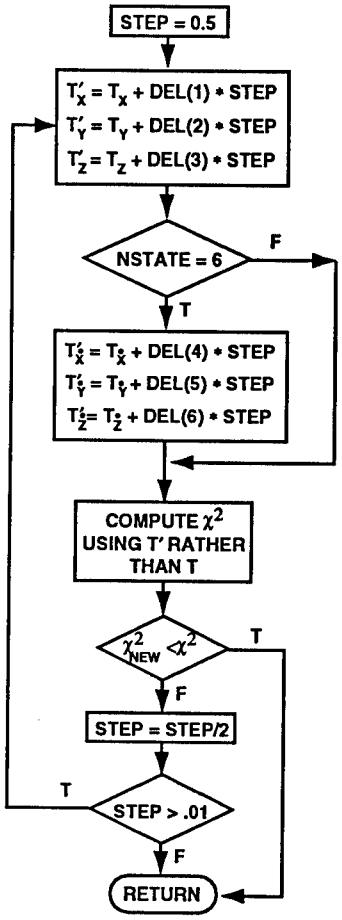
$$G = \left(\sum_{i=1}^{NSTATE} GRAD(I)^2 \right)^{1/2} \qquad \text{MAGNITUDE OF GRADIENT VECTOR}$$

$$D = \left(\sum_{i=1}^{NSTATE} DEL(I)^2 \right)^{1/2} \qquad \text{MAGNITUDE OF SEARCH DIRECTION VECTOR}$$

$$P = \sum_{i=1}^{NSTATE} GRAD(I) * DEL(I) \qquad \text{DOT PRODUCT}$$

$$\alpha = \cos^{-1}\left(\frac{P}{G * D}\right)$$

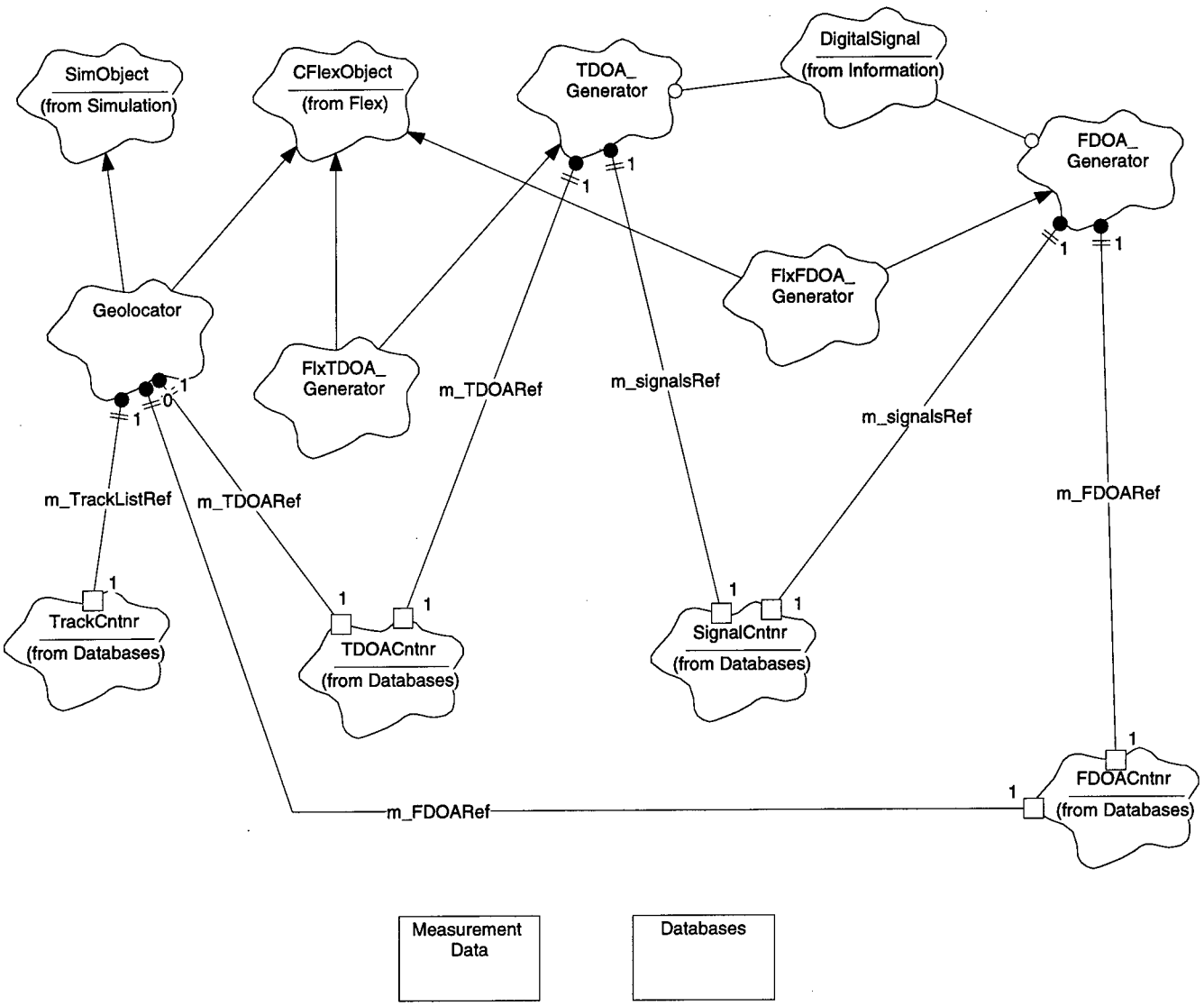
DECREASE STEP SIZE TO OBTAIN IMPROVEMENT

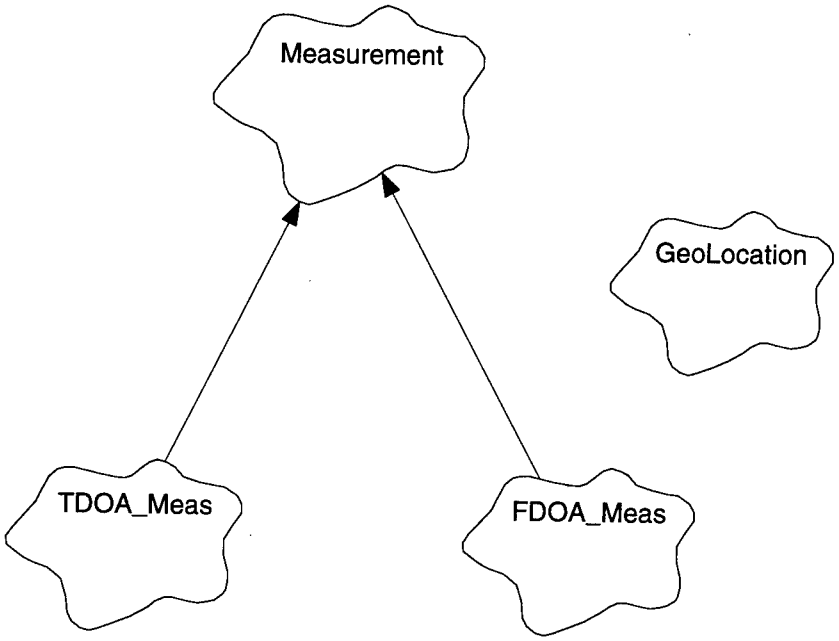


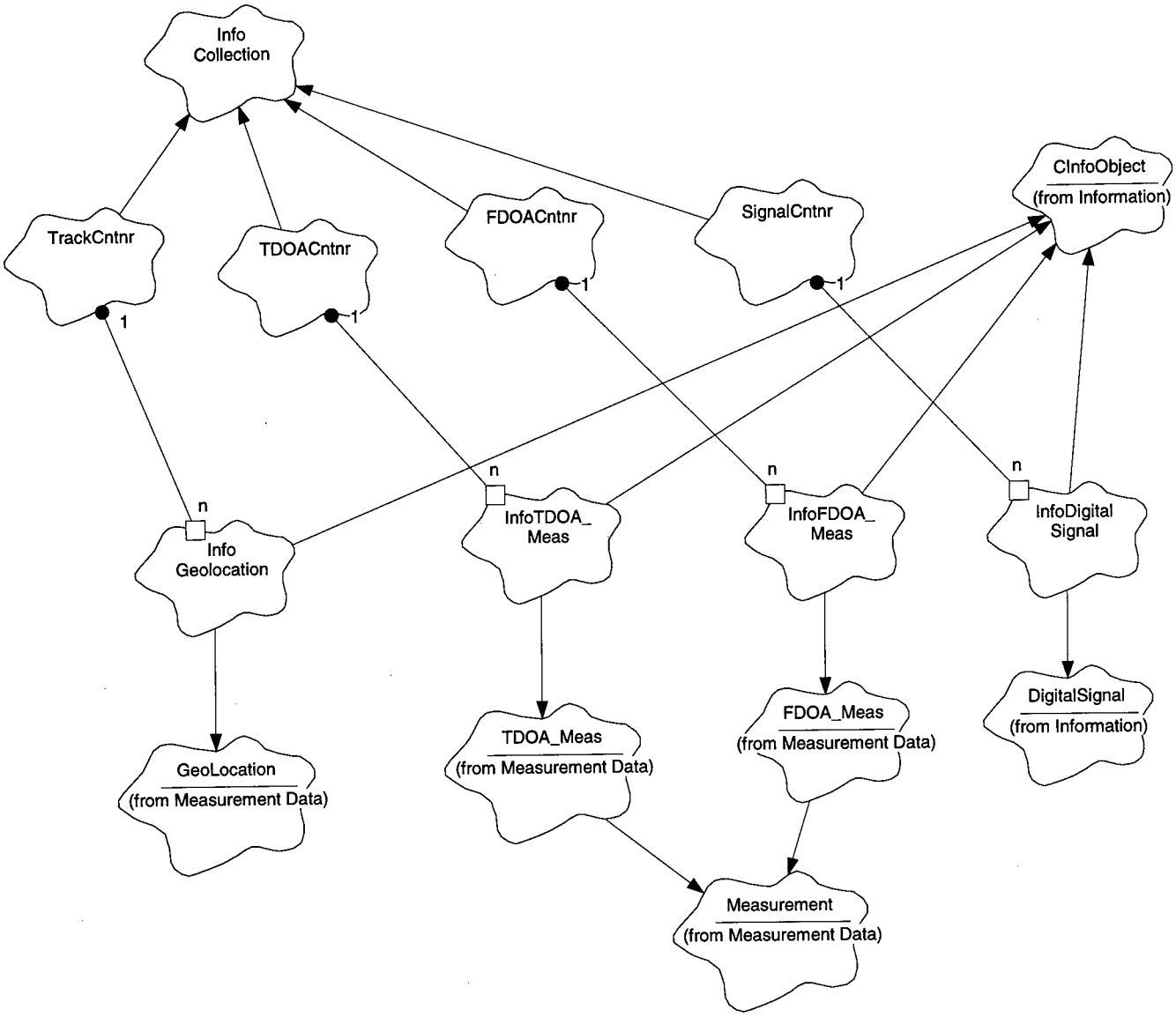
APPENDIX B

This appendix contains the class diagrams for the core classes used for performing geolocation. The class diagrams are presented in the Booch method, as generated in Rational Rose/C++.

Appendix B


SimObject
(from Simulation)
CFlexObject
(from Flex)
TDOA_
Generator
DigitalSignal
(from Information)
FDOA_
Generator
Geolocator
FlxTDOA_
Generator
FlxFDOA_
Generator
m_TDOARef
m_signalsRef
m_signalsRef
m_TrackListRef
m_TDOARef
m_FDOARef
TrackCntnr
(from Databases)
TDOACntnr
(from Databases)
SignalCntnr
(from Databases)
FDOACntnr
(from Databases)
m_FDOARef
Measurement
Data
Databases


Measurement
GeoLocation
TDOA_Meas
FDOA_Meas


Info
Collection
TrackCntnr
TDOACntnr
FDOACntnr
SignalCntnr
CInfoObject
(from Information)
1
1
1
1
n
n
n
n
Info
Geolocation
InfoTDOA_
Meas
InfoFDOA_
Meas
InfoDigital
Signal
GeoLocation
(from Measurement Data)
TDOA_Meas
(from Measurement Data)
FDOA_Meas
(from Measurement Data)
DigitalSignal
(from Information)
Measurement
(from Measurement Data)

APPENDIX C

This appendix contains the source code for several of the core classes whose diagrams appear in Appendix B. The source code was printed from Microsoft Visual C++, Version 2.2.

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: GEOLOCATOR.CPP        $
//    $Revision:: 21                    $
//    $Date:: 2/23/96 2:22p             $
//    $Modtime:: 2/22/96 9:26a          $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include <math.h>

// Geolocator

#include "CholeskyD.h"
#include "DSymMatrix.h"
#include "constants.h"
#include "reference.h"
#include "TDOACntnr.h"
#include "FDOACntnr.h"
#include "TrackCntnr.h"
#include "FlxSim.h"

#include "GeoLocator.h"


IMPLEMENT_FLEX_SERIAL( Geolocator, CFlexObject, 1, TRUE)

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

Geolocator::Geolocator()
: m_TDOARef(NULL),
  m_FDOARef(NULL),
  m_geoLoc(NULL),
  m_TrackListRef(NULL)
{
    m_name = "Geolocator";
}// end constructor //////////////////////

Geolocator::~Geolocator()
{
    m_TDOARef = NULL;
    m_FDOARef = NULL;
    m_geoLoc = NULL;
    m_TrackListRef = NULL;
}// end destructor ///////////////////////


void Geolocator::flexInit()
{
   FTime eventTime = FlexSim::c_Clock.time() + Period( 10.0 );
   FlxEvent* firstEvent = new FlxEvent( eventTime, this,
                                EVENT_METHOD(Geolocator,process) );

   FlexSim::c_EventCalendar.schedule( firstEvent );

}// end flexInit ////////////////////////////////////////

void Geolocator::process()
{
   unsigned i, j;
   GeoLocation locEst;
   TDOA_Set tdoas;
   FDOA_Set fdoas;

   unsigned numTDOAs = m_TDOARef->size();

   if( numTDOAs == 0 ) return;
```

```
// Build the TDOA Set...
   for( i=0; i<m_TDOARef->size(); ++i )
   {
      InfoTDOA_Meas* tdoaInfo = (InfoTDOA_Meas*)m_TDOARef->at(i);

      TDOA_Meas* tdoaMeas =  (TDOA_Meas*)tdoaInfo;
      tdoas.add( tdoaMeas );

   }


// Build the FDOA Set
   if( m_FDOARef != NULL )
   {
      for( i=0; i<m_FDOARef->size(); ++i )
      {
         InfoFDOA_Meas* fdoaInfo = (InfoFDOA_Meas*)m_FDOARef->at(i);
         FDOA_Meas* fdoaMeas =  (FDOA_Meas*)fdoaInfo;
         fdoas.add( fdoaMeas );
      }
   }

// Build the TDOA covariance matrix and invert...
   DSymMatrix tdoaCov( numTDOAs );

   for( i=1; i<=numTDOAs; ++i )
   {
      TDOA_Meas* tdoa_i = tdoas[i];

      for( j=1; j<=numTDOAs; ++j )
         tdoaCov(i-1,j-1) = tdoa_i->covWith( *(tdoas[j]) );

   }// end for i

// Initialize the geolocation estimate...
   locEst.initializeWith( tdoas );


// Perform Marquardt algorithm...
   doMarquardt( locEst, tdoas, fdoas, tdoaCov );

// Add to TrackCntnr
   m_TrackListRef->addTrack( new InfoGeoLocation(locEst) );

// here remove all tdoas and fdoas
   if ( m_FDOARef != NULL )
      m_FDOARef->removeAllFDOAs();
   m_TDOARef->removeAllTDOAs();

// Convert ECR state and covariance to lat/long...

}// end process ////////////////////////////////////////


DoubleMatrix Geolocator::computeDeltaTs( const Vec3& geoLocPos,
                                         const TDOA_Set& tdoaSet )
{
   unsigned i;
   unsigned numTDOAs = tdoaSet.size();
   double d1(0), d2(0);
   DoubleMatrix dt(numTDOAs), dtdoa(numTDOAs);

   for( i=1; i<=numTDOAs; ++i )
   {
      TDOA_Meas* tdoa_i = tdoaSet[i];

      d1 = (geoLocPos - tdoa_i->firstSensorPos()).magnitude();
      d2 = (geoLocPos - tdoa_i->refSensorPos()).magnitude();

      dt(i-1) = (d1 - d2) / C;
      dtdoa(i-1) = tdoa_i->delTime();
```

```
   }// end for i

   DoubleMatrix dts = dtdoa - dt;

   return dts;

}// end computeDeltaTs ////////////////////////////////////////


double Geolocator::computeFdoaChiSqrContrib( const GeoLocation& geoLocEst,
                                             const FDOA_Set& fdoaSet )
{
   double fdoaContrib(0.0), v1(0.0), v2(0.0), d1(0.0), d2(0.0);
   double deltaF( 0.0 ), c( 299792458.0 );

   unsigned i;

   Vec3 sensorPos,sensorVel,refSenPos,refSenVel;

   Vec3 locPos = geoLocEst.position();
   Vec3 locVel = geoLocEst.velocity();

   double numFDOAs = fdoaSet.size(),diff,avgFreq( 0.0 );

   if( numFDOAs > 0 )
   {

      for(i=1; i<= numFDOAs; i++)
         avgFreq += fdoaSet[i]->cntnrFreq()/numFDOAs;

      for( i=1; i<= numFDOAs; i++ )
      {
         FDOA_Meas* fdoa_i = fdoaSet[i];

         sensorPos = fdoa_i->firstSensorPos();
         sensorVel = fdoa_i->firstSensorVel();
         refSenPos = fdoa_i->refSensorPos();
         refSenVel = fdoa_i->refSensorVel();

         d1 = ( locPos - sensorPos ).magnitude();
         d2 = ( locPos - refSenPos ).magnitude();

         v1 = ( (locPos - sensorPos).dotProduct(locVel - sensorVel) ) / d1;

         v2 = ( (locPos - refSenPos).dotProduct(locVel - refSenVel) ) / d2;

        deltaF = ( avgFreq/C )*( v2 - v1 );

        diff = fdoa_i->deltaFreq() - deltaF;

        fdoaContrib += ( (fdoa_i->deltaFreq() - deltaF)/fdoa_i->sigmaSPF() ) *
                       ( (fdoa_i->deltaFreq() - deltaF)/fdoa_i->sigmaSPF() );

        deltaF = 0.0;

      }// end i for


   }// end if

   return fdoaContrib;

}// end computeFdoaChiSqrContrib ////////////////////////////

void Geolocator::addFdoaContrib(const GeoLocation& geoLocEst,
                                const FDOA_Set& fdoaSet,
                                DoubleMatrix& grad,
                                DSymMatrix& locEstInvCov )
{

   double fdoaContrib(0.0), v1(0.0), v2(0.0), d1(0.0), d2(0.0);
   double deltaF( 0.0 ), c( 297992458.0 ),fdivC(0.0),avgFreq(0.0);

   Vec3 sensorPos,sensorVel,refSenPos,refSenVel;
```

```
Vec3 locPos = geoLocEst.position();
Vec3 locVel = geoLocEst.velocity();

unsigned size = locEstInvCov.dim(),i,j,k;

DoubleMatrix pArray( size, size ), pVec( size ),tempCovInv( size , size );

double numFDOAs = fdoaSet.size(),z,before,after,after_before;

if( numFDOAs > 0 )
{

   for( i=0; i<size; i++ )
      for( j=0; j<size; j++ )
         tempCovInv( i,j ) = locEstInvCov( i,j );

   for(i=1; i<= numFDOAs; i++)
      avgFreq += fdoaSet[i]->cntnrFreq()/numFDOAs;

   for( i=1; i<= numFDOAs; i++ )
   {
      FDOA_Meas* fdoa_i = fdoaSet[i];

      sensorPos = fdoa_i->firstSensorPos();
      sensorVel = fdoa_i->firstSensorVel();
      refSenPos = fdoa_i->refSensorPos();
      refSenVel = fdoa_i->refSensorVel();

      d1 = ( locPos - sensorPos ).magnitude();
      d2 = ( locPos - refSenPos ).magnitude();

      v1 = ( (locPos - sensorPos).dotProduct(locVel - sensorVel) ) / d1;

      v2 = ( (locPos - refSenPos).dotProduct(locVel - refSenVel) ) / d2;

     fdivC = avgFreq / C;

     deltaF = fdivC*( v2 - v1 );

     fdoaContrib = ( (fdoa_i->deltaFreq() - deltaF)/
                     (fdoa_i->sigmaSPF()*fdoa_i->sigmaSPF()) );


     for( j=0; j< 3; j++ )
     {
        pVec( j ) = fdivC*( ( (locVel( j ) - refSenVel( j ))/d2 ) -
                            ( v2*( locPos( j ) - refSenPos( j ) )/(d2*d2) ) -
                            ( (locVel( j ) - sensorVel( j ))/d1 ) +
                            ( v1*( locPos( j ) - sensorPos( j ) )/(d1*d1) ) );
        z=pVec(j);
        z=pVec(j)*fdoaContrib;
        z=pVec(j)*fdoaContrib;
     }

     for( j=0; j<3; j++ )
        for( k=0; k<3; k++ )
        {
           pArray(j,k) = pVec(j)*pVec(k);
           z=pArray(j,k);
           z=pArray(j,k);
        }


   tempCovInv += (1.0/(fdoa_i->sigmaSPF()*fdoa_i->sigmaSPF()))*pArray;

   grad += fdoaContrib*pVec;

}// end i for

for(  i=0; i<size; i++ )
   for(  j=0; j<size; j++ )
   {
      before = locEstInvCov(i,j);
      after = tempCovInv(i,j);
```

```
            locEstInvCov( i,j ) = tempCovInv( i,j );
            after = locEstInvCov( i,j );
            after_before = after - before;
            z = locEstInvCov( i,j );
         }

   } // end if


}// end addFdoaContrib ***************************************

double Geolocator::computeChi2( const GeoLocation& geoLocEst,
                                const TDOA_Set& tdoaSet,
                                const FDOA_Set& fdoaSet,
                                const DSymMatrix& tdoaCov )
{
   Vec3 locPos = geoLocEst.position();

   DoubleMatrix dts = computeDeltaTs( locPos, tdoaSet );

   double fdoaContrib = computeFdoaChiSqrContrib( geoLocEst, fdoaSet );


   CholeskyD covDC( tdoaCov );
   DoubleMatrix y(dts.rows());

   covDC.solve( dts, y );

   DoubleMatrix chiSquare = transpose(dts) * y;

   double zeroElem = chiSquare(0);

   //fdoaContrib = 0.0;

   double chiSqr = zeroElem + fdoaContrib;

   return chiSqr;

}// end computeChi2 ////////////////////////////////////

void Geolocator::doMarquardt( GeoLocation& geoLocEst,
                              const TDOA_Set& tdoaSet,
                              const FDOA_Set& fdoaSet,
                              const DSymMatrix& tdoaCov )
{
 // Initialize.
    unsigned iter(0), nstate(3), maxIter(1000);
    double lambda(1.0e-6), step(1.0), angle(0.0),error(0.0);
    unsigned numTDOAs = tdoaSet.size();
    int converged(0);
    DSymMatrix stateCovInv(nstate), stateCov(nstate);
    DoubleMatrix grad(nstate), del(nstate);
    Vec3 locPos = geoLocEst.position();
    Vec3 locVel = geoLocEst.velocity();
    Vec3 locPos_p = locPos;
    Vec3 locVel_p = locVel;
    Vec3 delVec;


// Compute an initial chi-square...
    double chiSqrOld = computeChi2( geoLocEst, tdoaSet, fdoaSet, tdoaCov );

    double chiSqrNew,z,y;

    unsigned i,j,k;

    while( !converged )
    {
    // Construct inverse covariance matrix and correction
    //     gradient...


       grad.fill( 0.0 );
       stateCovInv.fill( 0.0 );
```

```
addTdoaContrib( geoLocEst, tdoaSet, tdoaCov, grad, stateCovInv );


addFdoaContrib( geoLocEst, fdoaSet, grad, stateCovInv );


do
{
// Modify inverse covariance with lambda and invert...
  //stateCov = stateCovInv;
  // stateCov *= (1.0 + lambda);
   for(i=0; i<nstate; i++)
      stateCovInv(i,i) = stateCovInv(i,i)*(1.0 + lambda);
   stateCov = inverse( stateCovInv );

// Determine new state estimate and new chi-square....
   del = stateCov * grad;
   locPos_p += Vec3( del(0), del(1), del(2) );
   geoLocEst.setPosition( locPos_p );
   if( nstate == 6 )
   {
      locVel_p += Vec3( del(3), del(4), del(5) );
      geoLocEst.setVelocity( locVel_p );
   }// end if
   chiSqrNew = computeChi2( geoLocEst, tdoaSet, fdoaSet, tdoaCov );

// Check for improvement...
   if( chiSqrNew < chiSqrOld )
   {
      locPos = locPos_p;
      goto label1;
   }

// Determine angle between gradient and search
//   directions...
   double gmag(0.0), dmag(0.0), gdotd(0.0);
   for( i=0; i<nstate; i++ )
   {
      gmag += grad(i)*grad(i);
      dmag += del(i)*del(i);
      gdotd += grad(i) * del(i);
   }

   angle = ::acos( gdotd / (::sqrt(gmag) * ::sqrt(dmag)) );
   double degAngle = angle*57.2958;
   if( angle > Pi )
      angle = 2.0 * Pi - angle;

// Check angle...
   if( angle > Pi/4.0 )
   {
      lambda *= 10.0;
      continue;
   }
   else
      break;

} while( lambda < 10.5 );

// Decrease step size to obtain improvement...
   delVec = Vec3( del(0), del(1), del(2) );
   do
   {
      locPos_p += delVec * step;
      geoLocEst.setPosition( locPos_p );
      if( nstate == 6 )
      {
         locVel_p += Vec3( del(3), del(4), del(5) );
         geoLocEst.setVelocity( locVel_p );
      }// end if
      chiSqrNew = computeChi2( geoLocEst, tdoaSet, fdoaSet, tdoaCov );
      if( chiSqrNew < chiSqrOld )
         break;
```

```
            else
               step /= 2.0;
         }while( step > 0.01 );

      // If step too small exit...
         if( step > 0.01 )
            locPos = locPos_p;
         else
            break;

label1 :
      // Check for convergence...
         if( ((chiSqrOld - chiSqrNew)/chiSqrOld < 0.0001) || (iter == maxIter) )
         {
            converged = TRUE;
            locPos = locPos_p;
         }
         else
         {
            chiSqrOld = chiSqrNew;
            lambda = lambda / 10.0;
            iter++;
            geoLocEst.setPosition( locPos_p );
         } // end if

      }// end while

      // Load the state estimate...
      double chiSqrRat =   (chiSqrOld - chiSqrNew)/chiSqrOld;
      geoLocEst.setPosition( locPos );


}// end doMarquardt ////////////////////////////////////

void Geolocator::addTdoaContrib( const GeoLocation& geoLocEst,
                                 const TDOA_Set& tdoaSet,
                                 const DSymMatrix& tdoaCov,
                                 DoubleMatrix& grad,
                                 DSymMatrix& locEstInvCov )
{
   unsigned i;
   unsigned numTDOAs = tdoaSet.size();
   double d1(0), d2(0);
   DoubleMatrix dt(numTDOAs), dtdoa(numTDOAs);
   Vec3 locPos = geoLocEst.position();
   Vec3 s1, s2;
   Vec3 pVec;
   DoubleMatrix P(grad.rows(),numTDOAs);

   for( i=1; i<=numTDOAs; ++i )
   {
      TDOA_Meas* tdoa_i = tdoaSet[i];
      s1 = tdoa_i->firstSensorPos();
      s2 = tdoa_i->refSensorPos();
      d1 = (locPos - s1).magnitude();
      d2 = (locPos - s2).magnitude();

      dt(i-1) = (d1 - d2) / C;
      dtdoa(i-1) = tdoa_i->delTime();

      pVec = 1.0/C * ( (locPos - s1)/d1 - (locPos - s2)/d2 );

      P(0,i-1) = pVec.x();
      P(1,i-1) = pVec.y();
      P(2,i-1) = pVec.z();

   }// end for i

   DoubleMatrix dts = dtdoa - dt;

   DSymMatrix tdoaInvCov = inverse( CholeskyD( tdoaCov) );

   locEstInvCov = (P * tdoaInvCov) * transpose(P);
```

```
   grad = (P * tdoaInvCov) * dts;


}// end addTdoaContrib ////////////////////////////////

CObject* Geolocator::copy()
{
   return ((CObject *) new Geolocator(*this));

}// end copy ////////////////////////////

CObList* Geolocator::GetAttributes()
{
   CObList* pList = CFlexObject::GetAttributes();

   if( pList == NULL )
      pList = new CObList();

// Add local attributes...

// Return the accumulated attributes...
   return pList;

}// end GetAttributes ////////////////////////////////

CObList* Geolocator::GetReferences()
{
   CObList* ans = CFlexObject::GetReferences();

   if( ans == NULL )
      ans = new CObList();

    // Add the local ports and return...
    ans->AddTail(new Reference(RUNTIME_CLASS(TDOACntnr),
                               (CFlexObject**)(&m_TDOARef)));
    ans->AddTail(new Reference(RUNTIME_CLASS(FDOACntnr),
                               (CFlexObject**)(&m_FDOARef)));
    ans->AddTail(new Reference(RUNTIME_CLASS(TrackCntnr),
                               (CFlexObject**)(&m_TrackListRef)));

   return ans;

}// end GetReferences ////////////////////////////////

BOOL Geolocator::ReferencesResolved()
{
    // Returns TRUE if the references of the Geolocator have been set.
    //   Otherwise, return FALSE

    // First check any inherited ports
    BOOL answer = CFlexObject::ReferencesResolved();

    // Check the local references
    if( ( m_TDOARef == NULL) ||
        ( m_TrackListRef == NULL))
        answer = FALSE;

    return answer;

}// end ReferencesResolved ////////////////////////////////////

void Geolocator::Serialize( FlxArchive& anArc )
{
    // First, get any inherited members...
    CFlexObject::Serialize( anArc );

    if( anArc.IsStoring())
    {
        anArc << m_TDOARef;
        anArc << m_FDOARef;
        anArc << m_TrackListRef;
    }
    else
    {
```

```
        anArc >> m_TDOARef;
        anArc >> m_FDOARef;
        anArc >> m_TrackListRef;
    }

}// end Serialize ////////////////////////////////////

// EOF
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: flxtdoa_generator.cpp $
//    $Revision:: 3                     $
//    $Date:: 1/17/96 3:52p             $
//    $Modtime:: 1/17/96 3:45p          $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include <math.h>
#include "FlxTDOA_Generator.h"
#include "Environment.h"
#include "attrib.h"
#include "FlxSim.h"
#include "reference.h"


IMPLEMENT_FLEX_SERIAL( FlxTDOA_Generator, CFlexObject, 1, TRUE)

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


FlxTDOA_Generator :: FlxTDOA_Generator()
   : CFlexObject( "TDOA Generator" )
{
}

// ***************************************************************************

CObject* FlxTDOA_Generator::copy()
{
   return ((CObject *) new FlxTDOA_Generator(*this));
}

// ***************************************************************************

CObList* FlxTDOA_Generator::GetAttributes()
{
    // Get any inherited attributes
    CObList* pList = CFlexObject::GetAttributes();
    if( pList == NULL)
        pList = new CObList();

    // Add local attributes
    pList->AddTail( new LongAttr( &m_Seed, " Measurement Error Seed", Range(1,1000000) ) );

    return pList;
}

// ***************************************************************************

CObList* FlxTDOA_Generator::GetReferences()
{
    // Return a list containing the references of the receiver

    // First, get any inherited ports, if any
    CObList* pList = CFlexObject::GetReferences();

    // If no inherited references, create a new object list
    if( pList == NULL)
        pList = new CObList();

    // Add the local references and return
    pList->AddTail( new Reference( RUNTIME_CLASS(SignalCntnr),
                                   (CFlexObject**)(&m_signalsRef)));
    pList->AddTail( new Reference( RUNTIME_CLASS(TDOACntnr),
                                   (CFlexObject**)(&m_TDOARef)));
    return pList;
```

```
}

// ****************************************************************************

BOOL FlxTDOA_Generator::ReferencesResolved()
{
    // Returns TRUE if the references of the receiver have been set.
    //   Otherwise, return FALSE

    // First check any inherited ports
    BOOL answer = CFlexObject::ReferencesResolved();

    // Check the local references
    if( ( m_signalsRef == NULL) ||
        ( m_TDOARef == NULL))
        answer = FALSE;

    return answer;
}

// ****************************************************************************

void FlxTDOA_Generator::Serialize( FlxArchive& anArc )
{
    // First, get any inherited members...
    CFlexObject::Serialize( anArc );

    if( anArc.IsStoring())
    {
        anArc << m_signalsRef;
        anArc << m_TDOARef;
        anArc << m_Seed;
    }
    else
    {
        anArc >> m_signalsRef;
        anArc >> m_TDOARef;
        anArc >> m_Seed;
    }
}

// ****************************************************************************

void FlxTDOA_Generator::flexInit()
{
    //
    // Initialize the TDOA generator and place its first event on
    // the event calendar.
    //

    FTime eventTime = FlexSim::c_Clock.time() + Period( 5.0);
    FlxEvent *firstEvent = new FlxEvent(eventTime, this,
                                EVENT_METHOD(TDOA_Generator,Process));
    FlexSim::c_EventCalendar.schedule(firstEvent);
}

// ****************************************************************************


// EOF
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: TDOA_GENERATOR.CPP    $
//    $Revision:: 9                     $
//    $Date:: 1/18/96 5:33p             $
//    $Modtime:: 1/18/96 5:25p          $
//
////////////////////////////////////////////////
#include "stdafx.h"

#include "gauss.h"

#include <math.h>
#include "TDOA_Generator.h"
#include "Environment.h"
#include "FlxSim.h"
#include "reference.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// ****************************************************************************

TDOA_Generator::TDOA_Generator()
    : m_signalsRef( NULL),
      m_TDOARef( NULL),
      m_Seed( 1 )
{
}

// ****************************************************************************

TDOA_Generator::~TDOA_Generator()
{
    m_signalsRef = NULL;
    m_TDOARef = NULL;
}

// ****************************************************************************


void TDOA_Generator::Process()
{
   unsigned numSignals = m_signalsRef->size();
   unsigned i;
   Period dt = 0.0;

   GaussianRandGen errorDist( m_Seed );


   if( numSignals < 2 ) return;

// For now, assume zero atmospheric effects...
   double sigmaATMb = 0.0;
   double sigmaATMr = 0.0;

   InfoDigitalSignal* refDigSig = (InfoDigitalSignal*)(m_signalsRef->at(0));
   double bandwidth = refDigSig->bandwidth();
   double sampleRate =  refDigSig->sampleRate();
   double snr = refDigSig->snr();
   double sigmaClock = refDigSig->clockError();
   double sigmaSPT = ::sqrt(3.0)  / (Pi * bandwidth * sqrt( bandwidth * sampleRate * snr));
   double varBias = sigmaClock*sigmaClock + sigmaATMb*sigmaATMb;
   double varTDOA = sigmaSPT*sigmaSPT + sigmaATMr*sigmaATMr + 2.0 * varBias;

// Create TDOA measurements from each sensor pair...
   TDOA_Meas tdoa;
   for( i=1; i<numSignals; ++i )
   {
```

```
      InfoDigitalSignal* firstDigSig = (InfoDigitalSignal*)(m_signalsRef->at(i));

       // warning: the order here is critical and must match the way the partials
       // and dt is computed when computing chisqr and state covariance
       dt = firstDigSig->refTime() - refDigSig->refTime();
       double meanVal = errorDist.randVal( dt, ::sqrt( varBias ) );
       Period measurement( errorDist.randVal( meanVal, ::sqrt( varTDOA ) ) );

       tdoa.setMeasTime( refDigSig->refTime() );
       tdoa.setTime( measurement );
       Vec3 refSensorPos = refDigSig->sensorPos();
       tdoa.setRefSensorPos( refDigSig->sensorPos() );
       tdoa.setSensor1Pos( firstDigSig->sensorPos() );
       tdoa.setRefSensorNum( 0 );
       tdoa.setfirstSensorNum( i );
       tdoa.setVar( varTDOA );
       tdoa.setBias( varBias );

       // Add new TDOA to the output collection...
       m_TDOARef->add( new InfoTDOA_Meas( tdoa) );

    }// for

}// end Process ////////////////////////////////////////////////////////////

// EOF
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: TDOACNTNR.CPP          $
//    $Revision:: 5                      $
//    $Date:: 12/04/95 10:49a            $
//    $Modtime:: 12/04/95 9:41a          $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include "attrib.h"
#include "TDOACntnr.h"


IMPLEMENT_FLEX_SERIAL( TDOACntnr, InfoCollection, 1, TRUE)

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// ****************************************************************************

TDOACntnr::TDOACntnr()
{
    m_name = "TDOA Container";
}

// ****************************************************************************

TDOACntnr::~TDOACntnr()
{
   removeAllTDOAs();
}

// ****************************************************************************

CObList* TDOACntnr::GetAttributes()
{
    // Get inherited attributes...
    CObList *pList = InfoCollection::GetAttributes();
    if (pList == NULL)
        pList = new CObList();

    // Add local attributes...
    if(! isEmptyTDOACntnr())
    {
        Iterator itr( m_contents);
        while( itr++)
        {
            InfoNode* node = ( InfoNode*)itr();
            InfoTDOA_Meas* aTDOA = ( InfoTDOA_Meas*)node->m_infoObj;
            pList->AddTail( new ObjectAttr( aTDOA, "InfoTDOA_Meas"));
        }
    }


    // Return the accumulated attributes...
    return pList;
}

// ****************************************************************************

void TDOACntnr::addTDOA( InfoTDOA_Meas* value)
{
    InfoNode* node = new InfoNode();

    node->m_infoObj = value;

    m_contents.add( *node);
}
```

```
// ********************************************************************************

void TDOACntnr::removeTDOA( InfoTDOA_Meas* value)
{
    Iterator itr( m_contents);

    while( itr++)
    {
        InfoNode* iNode = ( InfoNode*)itr();
        InfoTDOA_Meas* aTDOA = (InfoTDOA_Meas*)iNode->m_infoObj;
        if( aTDOA->isEqual( value))
        {
            m_contents.remove( *itr());
            delete value;
        }
    }
}

// ********************************************************************************

InfoTDOA_Meas* TDOACntnr::nextTDOA( )
{
    InfoNode* node = ( InfoNode*)m_contents.first();
    return ( InfoTDOA_Meas*)node->m_infoObj;

}

// ********************************************************************************

BOOL TDOACntnr::isEmptyTDOACntnr( )
{
    if( !m_contents.isEmpty())
    {
        Iterator itr( m_contents);
        if (itr++ == NULL)
            return TRUE;
        else
            return FALSE;
    }
        return TRUE;
}

// ********************************************************************************

void TDOACntnr::removeAllTDOAs( )
{
    if( !isEmptyTDOACntnr())
    {
        Iterator itr( m_contents);
        while( itr++)
        {
            InfoNode* node = ( InfoNode*)itr();
            InfoTDOA_Meas* aTDOA = ( InfoTDOA_Meas*)node->m_infoObj;
            delete aTDOA;
            delete node;
        }
    }
    m_contents.removeAll();

}// end removeAllTracks ////////////////////////////////////////////////////////

// ********************************************************************************

void TDOACntnr::Serialize( FlxArchive& anArc )
{
    InfoCollection::Serialize( anArc );

    if( !isEmptyTDOACntnr() )
    {
        Iterator itr( m_contents);
        while( itr++)
        {
            InfoNode* node = ( InfoNode*)itr();
```

```
            InfoTDOA_Meas* aTDOA = ( InfoTDOA_Meas*)node->m_infoObj;
            aTDOA->Serialize( anArc);
        }
    }
}

// ********************************************************************************
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: flxfdoa_generator.cpp $
//    $Revision:: 3                     $
//    $Date:: 1/24/96 5:28p             $
//    $Modtime:: 1/24/96 5:27p          $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include "FlxFDOA_Generator.h"
#include "attrib.h"
#include "Environment.h"
#include "FlxSim.h"
#include "reference.h"

IMPLEMENT_FLEX_SERIAL( FlxFDOA_Generator, CFlexObject, 1, TRUE)

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

FlxFDOA_Generator :: FlxFDOA_Generator()
   : FDOA_Generator()
{
   m_name = "FDOA Generator";
}

// ************************************************************

void FlxFDOA_Generator::flexInit()
{
    //
    // Initialize the FDOA generator and place its first event on
    // the event calendar.
    //

    FTime eventTime = FlexSim::c_Clock.time() + Period( 5.0);
    FlxEvent *firstEvent = new FlxEvent(eventTime, this,
                                EVENT_METHOD(FlxFDOA_Generator,Process));
    FlexSim::c_EventCalendar.schedule(firstEvent);
}

// ****************************************************************************

CObject* FlxFDOA_Generator::copy()
{
   return ((CObject *) new FlxFDOA_Generator(*this));
}

// ****************************************************************************

CObList* FlxFDOA_Generator::GetAttributes()
{
    // Get any inherited attributes
    CObList* pList = CFlexObject::GetAttributes();
    if( pList == NULL)
        pList = new CObList();

    // Return the accumulated, local attributes
    pList->AddTail( new LongAttr( &m_Seed, "Measurement Error Seed", Range(1,1000000) ) );

    // Add local attributes

    return pList;
}

// ****************************************************************************

CObList* FlxFDOA_Generator::GetReferences()
```

```
{
    // Return a list containing the references of the receiver

    // First, get any inherited ports, if any
    CObList* pList = CFlexObject::GetReferences();

    // If no inherited references, create a new object list
    if( pList == NULL)
        pList = new CObList();

    // Add the local references and return
    pList->AddTail( new Reference( RUNTIME_CLASS(SignalCntnr),
                                   (CFlexObject**)(&m_signalsRef)));
    pList->AddTail( new Reference( RUNTIME_CLASS(FDOACntnr),
                                   (CFlexObject**)(&m_FDOARef)));
    return pList;
}

// ****************************************************************************

BOOL FlxFDOA_Generator::ReferencesResolved()
{
    // Returns TRUE if the references of the receiver have been set.
    //   Otherwise, return FALSE

    // First check any inherited ports
    BOOL answer = CFlexObject::ReferencesResolved();

    // Check the local references
    if( ( m_signalsRef == NULL) ||
        ( m_FDOARef == NULL))
        answer = FALSE;

    return answer;
}

// ****************************************************************************

void FlxFDOA_Generator::Serialize( FlxArchive& anArc )
{
    // First, get any inherited members...
    CFlexObject::Serialize( anArc );

    if( anArc.IsStoring())
    {
        anArc << m_signalsRef;
        anArc << m_FDOARef;
        anArc << m_Seed;
    }
    else
    {
        anArc >> m_signalsRef;
        anArc >> m_FDOARef;
        anArc >> m_Seed;
    }
}

// ****************************************************************************

// EOF
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: fdoa_generator.cpp     $
//    $Revision:: 5                      $
//    $Date:: 2/23/96 2:21p              $
//    $Modtime:: 2/22/96 2:40p           $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include <math.h>
#include "FDOA_Generator.h"
#include "Environment.h"
#include "FlxSim.h"
#include "reference.h"
#include "gauss.h"


#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// ****************************************************************************

FDOA_Generator::FDOA_Generator()
    : m_signalsRef( NULL),
      m_FDOARef( NULL),
      m_Seed( 1 )
{
}

// ****************************************************************************

FDOA_Generator::~FDOA_Generator()
{
    m_signalsRef = NULL;
    m_FDOARef = NULL;
}

// ****************************************************************************


void FDOA_Generator::Process()
{
   unsigned numSignals = m_signalsRef->size();
   unsigned i;

   GaussianRandGen errorDist( m_Seed );


  if( numSignals < 2 ) return;

// For now, assume zero atmospheric effects...
   double sigmaATMb = 0.0;
   double sigmaATMr = 0.0;

   InfoDigitalSignal* refDigSig = (InfoDigitalSignal*)(m_signalsRef->at(0));
   double bandwidth = refDigSig->bandwidth();
   double sampleRate =  refDigSig->sampleRate();
   double snr = refDigSig->snr();
   double sigmaClock = refDigSig->clockError();
   double sigmaSPF = ::sqrt(3.0)  / (Pi * sampleRate * sqrt( bandwidth * sampleRate * snr));
   double varBias = sigmaClock*sigmaClock + sigmaATMb*sigmaATMb;
   double varFDOA = sigmaSPF*sigmaSPF + sigmaATMr*sigmaATMr + 2.0 * varBias;

// Create FDOA measurements from each sensor pair...
   FDOA_Meas FDOA;
   for( i=1; i<numSignals; ++i )
   {
```

```
      InfoDigitalSignal* firstDigSig = (InfoDigitalSignal*)(m_signalsRef->at(i));

       // warning: the order here is critical and must match the way the partials
       // and dt is computed when computing chisqr and state covariance
       double df = ( firstDigSig->cntnrFreq() - refDigSig->cntnrFreq() );
       double mean = errorDist.randVal( df, ::sqrt( varBias ) );
       double measurement = errorDist.randVal( mean, ::sqrt( varFDOA ) );

       FDOA.setMeasTime( refDigSig->refTime() );
       FDOA.setCntnrFreq( firstDigSig->cntnrFreq() );
       FDOA.setDeltaFreq( df );
       Vec3 refSensorPos = refDigSig->sensorPos();
       FDOA.setRefSensorPos( refDigSig->sensorPos() );
       FDOA.setRefSensorVel( refDigSig->sensorVel() );
       FDOA.setSensor1Pos( firstDigSig->sensorPos() );
       FDOA.setSensor1Vel( firstDigSig->sensorVel() );
       FDOA.setRefSensorNum( 0 );
       FDOA.setfirstSensorNum( i );
       FDOA.setSigmaSPF( sigmaSPF );
       FDOA.setVar( varFDOA );
       FDOA.setBias( varBias );

       // Add new FDOA to the output collection...
       m_FDOARef->add( new InfoFDOA_Meas( FDOA) );

   }// for


}// end Process ////////////////////////////////////////////////////////////

// EOF
```

```
////////////////////////////////////////////////
//
//    NRC FlexSim
//
//    $Workfile:: fdoacntnr.cpp          $
//    $Revision:: 3                      $
//    $Date:: 2/12/96 10:41a             $
//    $Modtime:: 2/05/96 2:33p           $
//
////////////////////////////////////////////////

#include "stdafx.h"

#include "attrib.h"
#include "FDOACntnr.h"


IMPLEMENT_FLEX_SERIAL( FDOACntnr, InfoCollection, 1, TRUE)

#ifdef _DEBUG
#define new DEBUG_NEW
#endif


// ****************************************************************************

FDOACntnr::FDOACntnr()
{
    m_name = "FDOA Container";
}

// ****************************************************************************

FDOACntnr::~FDOACntnr()
{
   removeAll();
}

// ****************************************************************************

CObList* FDOACntnr::GetAttributes()
{
    // Get inherited attributes...
    CObList *pList = InfoCollection::GetAttributes();
    if (pList == NULL)
        pList = new CObList();

    // Add local attributes...
    if(! isEmpty())
    {
        Iterator itr( m_contents);
        while( itr++)
        {
            InfoNode* node = ( InfoNode*)itr();
            InfoFDOA_Meas* aFDOA = ( InfoFDOA_Meas*)node->m_infoObj;
            pList->AddTail( new ObjectAttr( aFDOA, "InfoFDOA_Meas"));
        }
    }


    // Return the accumulated attributes...
    return pList;
}

// ****************************************************************************
void FDOACntnr::Serialize( FlxArchive& anArc )
{
    InfoCollection::Serialize( anArc );

    if( !isEmpty() )
    {
        Iterator itr( m_contents);
        while( itr++)
        {
```

```
            InfoNode* node = ( InfoNode*)itr();
            InfoFDOA_Meas* aFDOA = ( InfoFDOA_Meas*)node->m_infoObj;
            aFDOA->Serialize( anArc);
        }
    }
}

// ***************************************************************************
void FDOACntnr::removeAllFDOAs()
{
    if( !isEmpty())
    {
        Iterator itr( m_contents);
        while( itr++)
        {
            InfoNode* node = ( InfoNode*)itr();
            InfoFDOA_Meas* aFDOA = ( InfoFDOA_Meas*)node->m_infoObj;
            delete aFDOA;
            delete node;
        }
    }
    m_contents.removeAll();

}// end removeAllTracks ////////////////////////////////////////////////////////

// EOF
```

☆U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-47156

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.