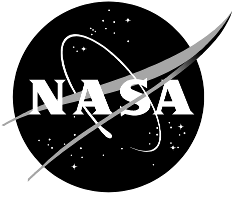
NASA/TM—20205001252



Air Traffic Management TestBed Data Exchange Model

Chok Fung Lai
Ames Research Center, Moffett Field, California

May 2020

NASA STI Program ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

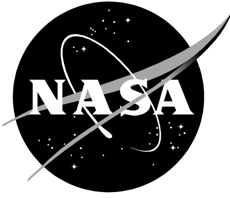
- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM—20205001252



Air Traffic Management TestBed Data Exchange Model

Chok Fung Lai
Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, CA 94035-1000

May 2020

Acknowledgments

The author would like to thank the other Air Traffic Management TestBed team members, Alan Lee, Phu Huynh, Huu Huynh, Jimmy Nguyen, David Wood, and Yun Zheng, for their contributions to the design and development of the data exchange model. The author would also like to thank Mohamad Refai, Kee Palopo, Gregory Wong, Confesor Santiago, and Katharine Lee for reviewing this technical memorandum.

This report is available in electronic form at

http://ntrs.nasa.gov/

Abstract

The Air Traffic Management (ATM) TestBed is a Platform as a Service that is being developed by the National Aeronautics and Space Administration (NASA) to help design, configure, integrate, run, and monitor air traffic simulations. The platform is designed to provide cloud services including back-end, big-data analytics tools, on-demand computing resource management, data storage, and communication middleware. The ATM TestBed reduces the time to test concepts and technologies, supports interactions among various methods such as human-in-the-loop and automation-in-the-loop simulations, and enables collaborative simulations by sharing technologies and tools in the ATM community. In order to allow easier access to simulation components, TestBed provides a messaging support layer for connectivity using a consistent set of input/output interfaces. In addition, a standard data format is introduced to facilitate communication between the components. The data exchange model, supported in the messaging support layer, standardizes the format of the information to be exchanged among the components. This document describes the messaging data model currently developed in TestBed and provides data dictionaries for references to component developers as well as simulation engineers.

This page intentionally left blank.

Table of Content

1. Introduction ... 9
2. Naming Conventions ... 10
   2.1. Data Structures ... 10
   2.2. Units of Measurement ... 10
   2.3. Package Names ... 11
3. Messaging Data Model ... 12
   3.1. SNDEM ... 13
      3.1.1. Meta Information ... 14
      3.1.2. User Defined Data ... 14
      3.1.3. Registry ... 16
4. Data Exchange Model ... 17
   4.1. Attributes ... 17
   4.2. Binary Data ... 19
   4.3. Flight Conflict ... 20
      4.3.1. Conflict Data ... 20
      4.3.2. Separation ... 21
   4.4. Flight Plan ... 22
      4.4.1. Flight Plan Type ... 24
   4.5. Image Data ... 24
   4.6. Multi-Purpose Interface Messages ... 25
   4.7. Resolution ... 27
      4.7.1. Waypoint ... 28
   4.8. Task ... 28
      4.8.1. Task Status ... 29
   4.9. Track ... 30
   4.10. Trajectories ... 31
      4.10.1. Trajectory ... 31
   4.11. Vehicle State ... 32
   4.12. Vehicle True State ... 32
      4.12.1. Vehicle True State, Version 1 ... 33
      4.12.2. Vehicle True State, Version 2 ... 37

Appendix A. Version History ... 37

5. References ... 39

List of Figures

Figure 1.1. Simulation setup: (a) without and (b) with TestBed ... 9
Figure 3.1. Hierarchy of the messaging data model ... 12
Figure 3.2. Flow diagram of a track message ... 13
Figure 3.3. Connectivity among two live SWIM feeds, Fuser, and Traffic Viewer ... 15
Figure 3.4. Engine data bundled in track messages via UDD ... 15
Figure 4.1. Relationships of data exchange model, user defined data, attributes, and registry . 18
Figure 4.2. Separation parameters ... 21
Figure 4.3. Screen sharing using image data ... 24
Figure 4.4. Connectivity among components using MPI messages: (a) without and (b) with TestBed ... 25
Figure 4.5. Task messages between Task Provider and Task Consumer ... 28
Figure 4.6. Flow diagram between Task Provider and Task Consumer ... 29
Figure 4.7. Connectivity between Air Traffic Operations Laboratory and ATM TestBed Laboratory ... 32

List of Listings

Listing 3.1. Engine data bundled in track message ... 16
Listing 3.2. Class definitions of Registry and Value Map ... 17
Listing 4.1. Example of an Attributes instance ... 17
Listing 4.2. Engine data model class ... 18
Listing 4.3. Engine data defined in attributes message ... 19

List of Tables

Table 2.1. Units of measurement ... 10
Table 2.2. Data structure names and their package names ... 11
Table 3.1. Data dictionary of SNDEM ... 13
Table 3.2. Data dictionary of Meta Information ... 14
Table 3.3. Data dictionary of User Defined Data ... 15
Table 4.1. Data dictionary of Attributes ... 17
Table 4.2. Data dictionary of Binary Data ... 20
Table 4.3. Data dictionary of Flight Conflict ... 20
Table 4.4. Data dictionary of Conflict Data ... 20
Table 4.5. Data dictionary of Separation ... 21
Table 4.6. Data dictionary of Flight Plan ... 22
Table 4.7. Data dictionary of Flight Plan Type ... 24
Table 4.8. Connectivity to components using MPI messages ... 25
Table 4.9. Supported MPI messages in TestBed ... 26
Table 4.10. Supported MPI messages in LVCGW ... 27
Table 4.11. Data dictionary of Resolution ... 27
Table 4.12. Data dictionary of Waypoint ... 28
Table 4.13. Data dictionary of Task ... 29
Table 4.14. Data dictionary of Task Status ... 30
Table 4.15. Data dictionary of Track ... 30
Table 4.16. Data dictionary of Trajectories ... 31
Table 4.17. Data dictionary of Trajectory ... 31

Table 4.18. Data dictionary of Vehicle State 32
Table 4.19. Data dictionary of Vehicle True State 33
Table 4.20. Data dictionary of Vehicle True State, Version 1 33
Table 4.21. Data dictionary of Position 3D 36
Table 4.22. Data dictionary of Position 2D 36
Table 4.23. Data dictionary of Body Orientation 37
Table 4.24. Updated data dictionary of Vehicle True State, Version 2 37
Table A.1. Version history 37
Table A.2. Example track messages 38

This page intentionally left blank.

1. Introduction

The Air Traffic Management (ATM) TestBed, formerly known as the Shadow Mode Assessment using Realistic Technologies for the National Airspace System (SMART-NAS) Testbed, has been under active development to enable collaborative simulations by sharing technologies and simulation components in the ATM community [1, 2, 3]. The envisioned way to achieve effective simulations is to provide access to components, namely realistic air transportation data, air traffic control operational systems, and simulation tools. Inter-connecting components requires data exchanges and therefore formats. However, components having distinct data exchange formats makes design, setup, execution and extension complicated, error prone and time consuming. Figure 1.1(a) shows a setup of a component, *A*, connecting to three components, *X*, *Y*, and *Z*, where the directed links (↔) represent data connectivity. Assume these components use distinct data exchange formats. In order for component *A* to connect to the other components, its developers must implement three data converters, denoted by *A⇄X*, *A⇄Y*, and *A⇄Z*, so that the format used in component *A* can be converted into the other formats and back. This approach is not scalable especially if the creation of a custom data converter requires software development, testing and maintenance. In the example, the developers have to create yet another data converter *X⇄Y* to run an extended setup involving connectivity between components *X* and *Y*.

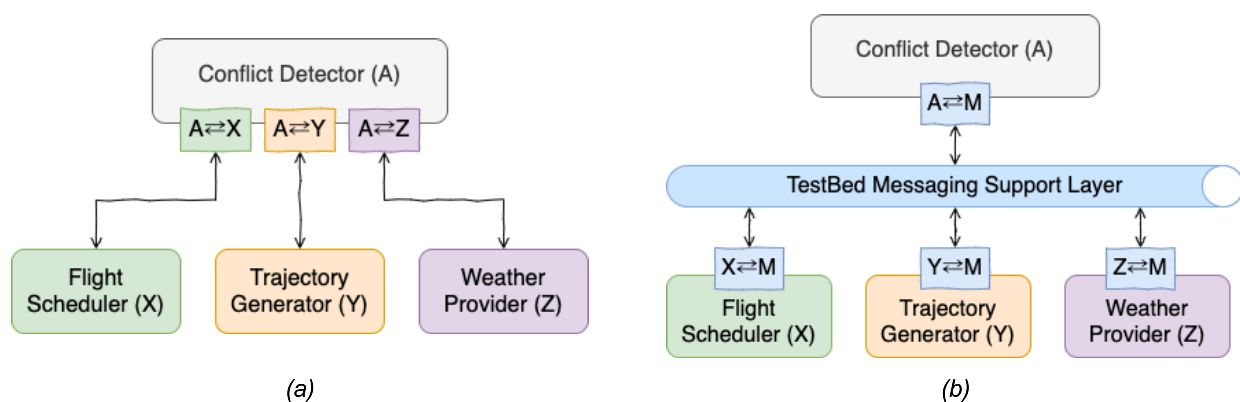


Figure 1.1. Simulation setup: (a) without and (b) with TestBed

In order to allow easier access to the components and to facilitate communication between the components, TestBed provides a messaging support layer for connectivity using a consistent set of input/output interfaces. The process introduces a standard data format called SMART-NAS Data Exchange Model (SNDEM). In Figure 1.1(b), component *A* no longer connects directly to the other components; instead, all the components are connecting to the messaging support layer. Each component has its own data converter, also known as an adapter, that converts its data format into the data exchange model, *M*, and back. The TestBed platform allows new components to be added, and existing components replaced or removed in a plug-and-play manner. In general, if *P* components need to connect to *Q* components, then *P*×*Q* custom data converters are needed; with the introduction of a common data exchange model, only *P*+*Q* generic data converters are needed.

The development of TestBed is based on a use case driven approach to support technology exploration, research experiments and flight tests including:

1. Connectivity test among ATM TestBed Laboratory, Distributed Simulation Research Laboratory [4], and *FutureFlight* Central [5] at NASA Ames Research Center
2. Connectivity test between NASA Ames and Langley Research Centers
3. Live flight data connection from the Federal Aviation Administration's (FAA's) System Wide Information System (SWIM) via the NASA's Sherlock ATM Data Warehouse [6]
4. *Autoresolver* [7] evaluation during the Boeing 2018 *ecoDemonstrator* [8] flight test

5. Tailored Arrival Manager [9] evaluation and datalink connectivity during the Boeing 2020 ecoDemonstrator flight test with the FAA
6. Urban Air Mobility (UAM) [10] human-in-the-loop experiments with Uber

The purpose of this document is to provide detailed information of the Data Exchange Model currently developed in TestBed and to provide data dictionaries of each data model for references to component developers and simulation engineers. Though the data models have been created to support the use cases mentioned above, there are general mechanisms exchanging custom data that are not defined in the current model.

Naming conventions and units of measurement are presented in Section 2. Section 3 describes the messaging data model used in the messaging support layer. Data dictionaries of the data models currently defined in TestBed are detailed in Section 4. Finally, version history of the data exchange model is documented in Appendix A. Since the data exchange model will continue to evolve, please contact the ATM TestBed Development Team (email: chok.f.lai@nasa.gov) to obtain the latest version of this document. The information on how to connect and use TestBed will be documented in a User Guide and Developer Guide.

2. Naming Conventions

The data exchange model has been developed using Java Standard Edition (SE) Development Kit (SDK) version 8 [11]. The naming conventions in the TestBed codebase follow the Google Java Style Guide [12] and the class names are in camel case. For example, `VehicleTrueState` is the name of the class *Vehicle True State*.

2.1. Data Structures

The primitive types, arrays, *enum* types, and the class *String* are described in the Java Language Specification [13]. Two collection data structures, *List* and *Map*, are also used:

1. A `List<E>` is a collection of elements with a generic type E.
2. A `Map<K,V>` is a collection of mappings of key to value. The keys and values have generic types K and V, respectively.

2.2. Units of Measurement

Table 2.1 lists the units of measurement used in the data exchange model.

Table 2.1. Units of measurement

Symbol	Unit	Unit Of	Description
%	Percentage	--	The dimensionless unit of a fraction of 100. One percentage equals one-hundredth, i.e., $1\% = 1/100$.
deg	Degree	Angle	For headings, values are between 0 and 360, inclusive. Note that headings of 0 degrees and 360 degrees are the same. For latitudes, values are in World Geodetic System 1984 (WGS84) reference coordinate system [14]. Latitude values are between −90 and +90, inclusive. Positive values are north of the equator (N), and negative values are south of the equator (S). For longitudes, values are in WGS84 reference coordinate system. Longitude values are between −180 and +180, inclusive. Positive values are east of the prime meridian (E), and negative values are west of the prime meridian (W).

ft	Foot	Length	The unit of vertical distance in air navigation.
inHg	Inch of mercury	Pressure	The unit of pressure in an altimeter setting.
kt	Knots	Speed	Positive values indicate forward movement, and negative values indicate backward movement.
lbs	Pound	Weight	The unit of weight in the flight data model.
min	Minute	Time	The unit of time in the flight data model. One minute equals 60 seconds.
ms	Millisecond	Time	The unit of time. One millisecond equals one-thousandth of a second, i.e., $1\, ms = 1/1{,}000\, sec$. Time is measured by the number of milliseconds elapsed since January 1, 1970, 00:00:00 GMT. Duration is measured by taking the difference between two timestamps.
nmi	Nautical mile	Length	The unit of horizontal distance in air navigation. One nautical mile equals 1,852 meters.
persons	Person	People	Number of people.
sec	Second	Time	The unit of time in the flight data model. One second equals 1,000 milliseconds.

2.3. Package Names

Table 2.2 lists the data structure names and their package names used in this document. The fully qualified name is the package name followed by the data structure name, e.g., `java.util.List` for `List`.

Table 2.2. Data structure names and their package names

Data Structure Name	Package Name
Attributes	gov.nasa.sntb.messagingdatamodels.sndem.attributes
Binary Data	gov.nasa.sntb.messagingdatamodels.sndem.binary
Class	java.lang
Conflict Data	gov.nasa.sntb.messagingdatamodels.sndem.flightstate
Conflict Flight Data	nasa.arc.aac.aacinterface
Flight Conflict	gov.nasa.sntb.messagingdatamodels.sndem.flightstate
Flight Plan	gov.nasa.sntb.messagingdatamodels.sndem.flightplan
Flight Plan Type	gov.nasa.sntb.messagingdatamodels.sndem.flightplan
Image Data	gov.nasa.sntb.messagingdatamodels.sndem.binary
Linked Hash Map	java.util
List	java.util
Map	java.util
Meta Info	gov.nasa.sntb.messagingdatamodels.sndem
Mpi Message	gov.nasa.sntb.messagingdatamodels.sndem.mpi.message
Registry	gov.nasa.sntb.messagingdatamodels.sndem.attributes
Resolution	gov.nasa.sntb.messagingdatamodels.sndem.resolution

Separation	gov.nasa.sntb.messagingdatamodels.sndem.flightstate
Snccm	gov.nasa.sntb.messagingdatamodels.snccm
Sndem	gov.nasa.sntb.messagingdatamodels.sndem
Snhmm	gov.nasa.sntb.messagingdatamodels.snhmm
Snmdm	gov.nasa.sntb.messagingdatamodels.snmdm
String	java.lang
Task	gov.nasa.sntb.messagingdatamodels.sndem.task
Task Status	gov.nasa.sntb.messagingdatamodels.sndem.task
Track	gov.nasa.sntb.messagingdatamodels.sndem.flightstate
Trajectories	gov.nasa.sntb.messagingdatamodels.sndem.trajectory
Trajectory	gov.nasa.sntb.messagingdatamodels.sndem.trajectory
User Defined Data	gov.nasa.sntb.messagingdatamodels.sndem
Vehicle State	gov.nasa.sntb.messagingdatamodels.sndem.flightstate
Vehicle True State	gov.nasa.sntb.messagingdatamodels.sndem.simuniverse
Waypoint	gov.nasa.sntb.commoninterfacesupport.utilities.data

3. Messaging Data Model

SMART-NAS Messaging Data Model is a unified data representation in the TestBed messaging support layer and standardizes the format of the information to be exchanged among simulation components. Currently, three concrete models have been developed:

1. Command and Control Model—contains messages for controlling components such as startup and shutdown.
2. Data Exchange Model—contains messages for data exchanges between components.
3. Health and Monitor Model—contains messages for monitoring components.

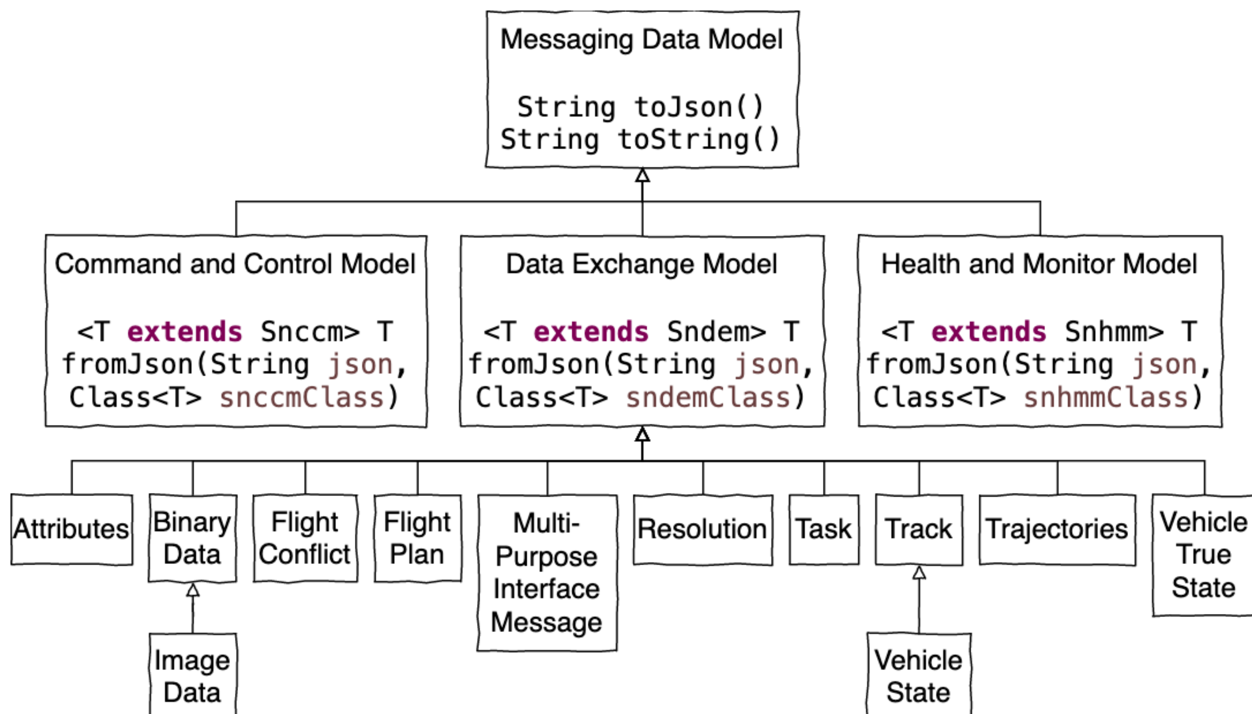


Figure 3.1. Hierarchy of the messaging data model

The hierarchy of these models is shown in Figure 3.1 where the up arrow (↑) indicates extension. Thus, a class at the tail of an arrow extends the class at the head of the arrow. The models provide application programming interfaces for converting messages to and from the format being used in the messaging support layer. This document focuses on the data exchange model. The two other models are internal to the TestBed platform and are out of the scope.

To ease development effort, each message is currently converted into a JavaScript Object Notation (JSON) [15] string using Google *Gson* library [16]. Figure 3.2 is a flow diagram illustrating how a track instance is transmitted from component *A* to component *B*. First, the track instance is passed to an adapter's publisher which calls the method `Sndem.toJson()` to convert the instance into a JSON string. Second, the JSON string is transmitted to a subscriber via the messaging support layer. Finally, the subscriber calls the method `Sndem.fromJson(String)` to convert the JSON string back into a track instance.

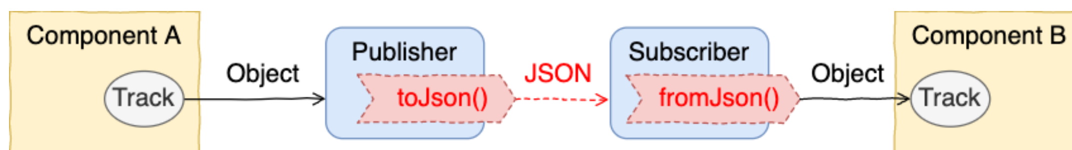


Figure 3.2. Flow diagram of a track message

Note that converting a messaging data model into a JSON string and back (the red part in the diagram) is considered internal in the TestBed platform. In the future, the JSON format may be replaced with another format that supports additional features, such as compression and encryption, to accommodate better network transfer rates as well as data protection without affecting the existing adapter implementations.

3.1. SNDEM

`Sndem` is the abstract, base class for all the SNDEM (SMART-NAS Data Exchange Model) classes defined in the TestBed messaging support layer. Table 3.1 lists the data dictionary of this data structure using the following column definitions:

1. Field—name of the field defined in the data model or structure.
2. Type—name of the data type; complex data types will be listed in subsections.
3. Unit—unit of measurement, if available, of the field value (see Section 2.2).
4. Description—brief description of the field as well as optional information including value ranges, name aliases defined in the previous versions, and an example value.
5. Since—first TestBed version supporting the field. The version history is documented in Appendix A.

Note: throughout this document, for readability, long field names, type names and units of measurement in the data dictionary tables are split into multiple lines.

Table 3.1. Data dictionary of SNDEM

Field	Type	Unit	Description	Since
meta	Meta Info		Meta-information of the data exchange model instance (see Section 3.1.1). Field alias: `metaInfo`.	1.0a
_udd	User Defined Data		Optional user-defined data storing name-value pairs based on given types (see Section 3.1.2). Value is null if not available. Field alias: `userDefinedData`.	2.0a

3.1.1. Meta Information

A `Meta Info` instance stores metadata and information about a data exchange model including the identifier of the adapter's publisher, version of the message, time when the message was published, and time when the message was received. Table 3.2 lists the data dictionary of the meta information.

Table 3.2. Data dictionary of Meta Information

Field	Type	Unit	Description	Since
src	String		Source of the data exchange model message, i.e., who publishes the data. The format of this value is `[ComponentTitle].[BlockID]`. The value will be automatically set in an adapter's publisher if it is not programmatically set by a developer. Field alias: `source`. Example: `Traffic Viewer.2`.	0.9b
ver	String		Version of the data exchange model. This is also the version of the TestBed Software Development Kit. Value is null if not available. Field alias: `version`. Example: `2.0a`.	2.0a
tpub	long	ms	Publication time when the data exchange model message was published by a component to the messaging layer. It is possible that multiple messages have the same publication time when they were published within the same millisecond. Value range is `[0, +inf]` or zero (0) if not available. Field alias: `timePublished`. Example: `1512429016457`.	0.9b
tsub	long	ms	Subscription time when the data exchange model was received by a component from the messaging layer. It is possible that multiple messages have the same subscription time if they were received within the same millisecond. Value range is `[0, +inf]` or zero (0) if not available. Field alias: `timeArrived`, `timeSubscribed`. Example: `1518546378849`.	0.9b

3.1.2. User Defined Data

A `User Defined Data (UDD)` instance stores user specific name-value pairs that are not defined in the data exchange model. The name-value pairs can be grouped by a type that is, by convention, a fully qualified class name of the user specific data structure. Table 3.3 lists the data dictionary of the UDD. Internally, a UDD instance uses a Registry (see Section 3.1.3) data structure to store custom data.

Table 3.3. Data dictionary of User Defined Data

Field	Type	Unit	Description	Since

registry	Registry		Mappings of data type to name-value pairs:
			• Key = type of the data
			• Value = mappings of data name to value:
			  • Key = name of the data
			  • Value = value of the data as string
				0.9b

The UDD provides flexibility for component adapter developers to store custom and extended data in any existing data exchange model. Figure 3.3 shows a setup demonstrating the usage of the UDD and connectivity among four components:

1. SWIM: Airport Surface Detection Equipment—Model X (ASDE-X)
2. SWIM: Traffic Flow Management Data (TFMData)
3. Fuser from the Airspace Technology Demonstration 2 (ATD-2) [17]
4. TestBed Traffic Viewer [18]

The UDD associated with the track messages consists of the following information:

- Globally Unique Flight Identifier (GUFI) of the system that produced the track
- Identifier of the airport that is responsible for the track
- Identifiers of the departure and destination airports

and the UDD associated with the flight plan messages includes the following information:

- GUFI of the system that produced the flight plan
- Estimated and scheduled times of arrival to arrival fix, landing, and gate arrival
- Engine class (jet, turbo, or piston), equipment and weight class qualifiers

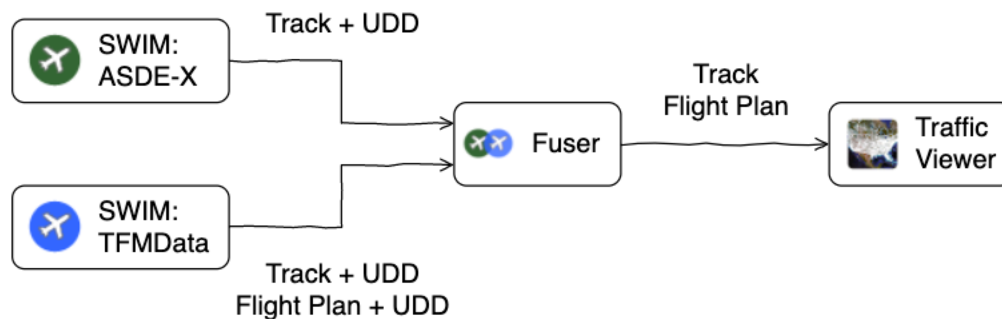


Figure 3.3. Connectivity among two live SWIM feeds, Fuser, and Traffic Viewer

Here is an example demonstrating the usage of the UDD. Suppose that two component adapters, My Publisher and My Subscriber, need to exchange track messages with aircraft engine data. Even though no engine data model is currently defined in the TestBed data exchange model, the engine data can still be bundled in the track messages, as illustrated in Figure 3.4.

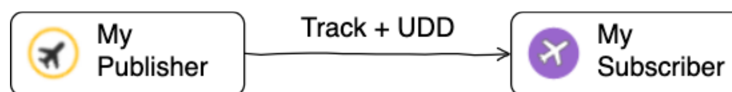


Figure 3.4. Engine data bundled in track messages via UDD

Assume the aircraft engine data model has a type string "`com.example.model.Engine`" and consists of two fields: a string representing the engine model ("`model`") and an integer number representing the rotations per minute ("`rpm`"). Listing 3.1 lists code snippets, in the Java programming language, demonstrating two use cases:

(a) A publisher component publishes a track message with the engine data bundled, and
(b) A subscriber component receives the bundled engine data from a track message.

Listing 3.1. Engine data bundled in track message

(a)

```
package com.example.component.MyPublisher;

import gov.nasa.sntb.messagingdatamodels.sndem.UserDefinedData;
import gov.nasa.sntb.messagingdatamodels.sndem.flightstate.Track;
import gov.nasa.sntb.simcomponent.adapter.PluginAdapter;

public class MyPublisher extends PluginAdapter {
    @Override
    public void execute() {
        Track track = new Track("NASA123", 37.77f, -122.42f, 1200f, 0);
        String type = "com.example.model.Engine";
        UserDefinedData engine = track.getUserDefinedData().get(type);
        engine.setString("model", "boost");
        engine.setInt("rpm", 123456);
        publish(track);
    }
}
```

(b)

```
package com.example.component.MySubscriber;

import gov.nasa.sntb.architectblueprint.interfaces.datasets.TrackDataset;
import gov.nasa.sntb.messagingdatamodels.sndem.UserDefinedData;
import gov.nasa.sntb.messagingdatamodels.sndem.flightstate.Track;
import gov.nasa.sntb.simcomponent.adapter.PluginAdapter;

public class MySubscriber extends PluginAdapter implements TrackDataset {
    @Override
    public void processTrackDataset(String key, Track dataset) {
        String type = "com.example.model.Engine";
        UserDefinedData engine = dataset.getUserDefinedData().get(type);
        String model = engine.getString("model");
        int rpm = engine.getInt("rpm");
    }
}
```

3.1.3. Registry

A `Registry` instance is a specialized map data structure storing mappings of type to `Value Map`, which itself is a map. The keys are types and the values are mappings of name to value. Thus, each value can be uniquely identified by a pair of type and name. Listing 3.2 lists the class definitions of the Registry and Value Map. The purpose of the Registry data structure is to provide a standardized way to store and query values by types and names. Note that registry instances are referenced in both the User Defined Data (see Section 3.1.2) and the Attributes (see Section 4.1).

Listing 3.2. Class definitions of Registry and Value Map

```
package gov.nasa.sntb.messagingdatamodels.sndem.attributes;

public class Registry extends LinkedHashMap<String, ValueMap> {
}

public class ValueMap extends LinkedHashMap<String, String> implements Delegate {
}
```

4. Data Exchange Model

Each data exchange model class extends the abstract class `Sndem` (see Section 3.1). The extension allows new capabilities, such as data compression and data encryption, to be added in the future without modifications of any existing subclasses. The following subsections detail each data exchange model defined in TestBed.

4.1. Attributes

The `Attributes` model provides a means for exchanging data that are not currently defined in the data exchange model. This allows current and future airspace concepts to be rapidly prototyped even if a limited number of data exchange models (presented in Section 4) are implemented in TestBed. Table 4.1 lists the data dictionary of the attributes model. Internally, an attributes instance uses a Registry (see Section 3.1.3) data structure to store custom data.

Table 4.1. Data dictionary of Attributes

Field	Type	Unit	Description	Since
`type`	`String`		Current type of the attributes. Example: `com.example.model.Engine`.	`0.9b`

`registry`	`Registry`		Mappings of attribute type to name-value pairs:
			• Key = type of the attribute
			• Value = mappings of attribute name to value:
			  • Key = name of the attribute
			  • Value = value of the attribute as string
				`0.9b`

Listing 4.1 is an example of an Attributes data structure containing two types, `type1` and `type2`, where the first and second types have *M* and *N* pairs of attributes, respectively.

Listing 4.1. Example of an Attributes instance

```
Attributes {
  type1 = {
    name1 = value1,
    name2 = value2,
    ...
    nameM = valueM
  },
  type2 = {
    ...
    nameN = valueN
  }
}
```

One main difference between the User Defined Data and the Attributes is the data hierarchy. The former one is a data structure used to include extra fields in an existing data exchange model, while the latter one is a standalone data exchange model. Figure 4.1 shows a relationship diagram where the boxes represent the classes, the white-triangle link (◁−) represents the "is a" relationship, and the diamond links (♦−) represent the "has a" relationship:

1. The Attributes class is a subclass of the Data Exchange Model class.
2. A Data Exchange Model instance has a User Defined Data instance.
3. An Attributes instance has a Registry instance that stores attribute values.
4. A User Defined Data instance has a Registry instance that stores custom values.

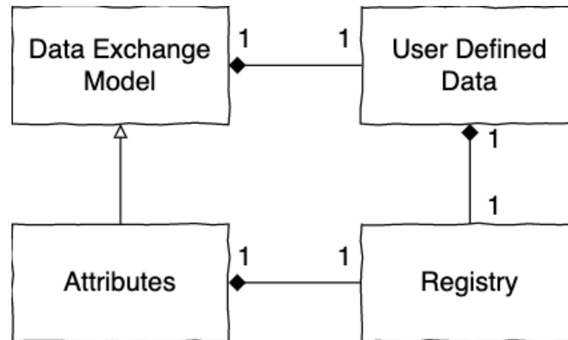


Figure 4.1. Relationships of data exchange model, user defined data, attributes, and registry

To illustrate the usage of the Attributes model, consider that one adapter needs to publish engine data to another adapter. The engine data may be exchanged by executing the following steps:

1. Create a data model class named Engine.
2. Add a constructor accepting an Attributes parameter in the class Engine.
3. Add a method `toAttributes()` to convert the engine data into an Attributes instance.

Listing 4.2. Engine data model class

```
1   package com.example.mode;
2
3   import gov.nasa.sntb.messagingdatamodels.sndem.attributes.Attributes;
4
5   public class Engine {
6       private static final String TYPE = Engine.class.getName();
7       private String model;
8       private int rpm;
9
10      public Engine(String model, int rpm) {
11          this.model = model;
12          this.rpm = rpm;
13      }
14
15      public Engine(Attributes attributes) {
16          Attributes engine = attributes.getAttributes(TYPE);
17          model = engine.getString("model");
18          rpm = engine.getInt("rpm");
19      }
20
21      public Attributes toAttributes() {
22          Attributes engine = new Attributes(TYPE);
23          engine.setString("model", model);
```

```
24          engine.setInt("rpm", rpm);
25          return engine;
26      }
27
28      public String getModel() {
29          return model;
30      }
31
32      Public int getRpm() {
33          return rpm;
34      }
35  }
```

Listing 4.2 is a sample implementation of the class Engine. Lines 10-13 define the constructor. Lines 16-18 demonstrate how to get the engine data via the supplied `attributes` instance. Lines 22-25 demonstrate how to create an `Attributes` instance based on the current engine data. Lines 28-34 are the getter methods. Listing 4.3 lists (a) a publisher component publishing an attributes message with the engine data, and (b) a subscriber component receiving the engine data from an attributes message.

Listing 4.3. Engine data defined in attributes message

(a)

```
package com.example.component.MyPublisher;

import com.example.model.Engine;
import gov.nasa.sntb.messagingdatamodels.sndem.attributes.Attributes;
import gov.nasa.sntb.simcomponent.adapter.PluginAdapter;

public class MyPublisher extends PluginAdapter {
    @Override
    public void execute() {
        Engine engine = new Engine("boost", 123456);
        publish(engine.toAttributes());
    }
}
```

(b)

```
package com.example.component.MySubscriber;

import com.example.model.Engine;
import gov.nasa.sntb.architectblueprint.interfaces.datasets.AttributesDataset;
import gov.nasa.sntb.simcomponent.adapter.PluginAdapter;

public class MySubscriber extends PluginAdapter implements AttributesDataset {
    @Override
    public void processAttributesDataset(String key, Attributes dataset) {
        Engine engine = new Engine(dataset);
        String model = engine.getModel();
        int rpm = engine.getRpm();
    }
}
```

4.2. Binary Data

The `Binary Data` model stores binary data as a byte array for data exchange. The model is useful for exchanging data that are computer-readable but not human-readable such as images (see Section 4.5) and weather data. The format of a binary data content is identified by a

Multipurpose Internet Mail Extensions (MIME) [19] type. Table 4.2 lists the data dictionary of the binary data model.

Table 4.2. Data dictionary of Binary Data

Field	Type	Unit	Description	Since
key	String		Key of the data. This will be used as the topic key. Example: `pi`.	0.9b
mimeType	String		MIME type of the binary data. Example: `application/octet-stream`.	0.9b
data	byte[]		Array of bytes representing the binary data. Example: `[3,1,4,1,5,9]`.	0.9b

4.3. Flight Conflict

The `Flight Conflict` model is a container for the conflict information in Autoresolver. A conflict represents a detected, predicted loss of separation between two flights at a future time. Table 4.3 lists the data dictionary of the flight conflict model.

Table 4.3. Data dictionary of Flight Conflict

Field	Type	Unit	Description	Since
detectionTime	long	ms	The time, in milliseconds, when the conflict detection occurs. Example: `1518548215902`.	0.9b
conflictTime	long	ms	The time, in milliseconds, when the first loss-of-separation conflict is predicted to occur. Example: `1518548275902`.	0.9b
firstFlight	Conflict Data		The conflict data (see Section 4.3.1) pertaining to the first flight.	0.9b
secondFlight	Conflict Data		The conflict data pertaining to the second flight.	0.9b

4.3.1. Conflict Data

A `Conflict Data` instance contains information related to a particular flight involved in a loss-of-separation conflict. It originated from the class `Conflict Flight Data` in the Advanced Airspace Concept (AAC) [7]. Table 4.4 lists the data dictionary of the conflict data.

Table 4.4. Data dictionary of Conflict Data

Field	Type	Unit	Description	Since
callsign	String		The callsign of the flight. Example: `AAL123`.	0.9b
latitude Degrees	double	deg	The latitude value, in degrees, of the trajectory point at the conflict. Value range is `[-90, +90]`.	0.9b

			Example: 37.67.	
longitude Degrees	double	deg	The longitude value, in degrees, of the trajectory point at the conflict. Value range is [-180, +180]. Example: -122.26.	0.9b
altitudeFeet	double	ft	The altitude, in feet, of the trajectory point at the conflict. The altitude type depends on the conflict detectors. It may be an indicated altitude of aircraft above mean sea level (MSL) on a standard day, or a pressure altitude. Example: 23987.65.	0.9b
trueHeading Degrees	double	deg	The true heading, in degrees, of the trajectory point at the conflict. Value range is [0, 360]. Example: 120.45.	0.9b
required Separation	Separation		The required separation parameters that have been violated at the conflict (see Section 4.3.2).	0.9b
type	String		The algorithm-specific type string describing the nature of the trajectories such as "AAC," "dead-reckon," "flight plan" or "maneuver." Example: dead-reckon.	0.9b

4.3.2. Separation

A Separation instance stores separation parameters used in a conflict detection algorithm. The parameters describe a right circular cylinder, as shown in Figure 4.2, where the center represents the location of a flight, the radius represents the horizontal separation distance, and the height represents *twice* the vertical separation distance.

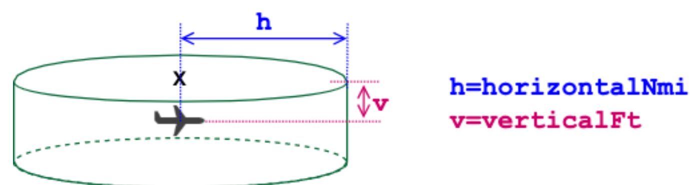


Figure 4.2. Separation parameters

Table 4.5 lists the data dictionary of the separation.

Table 4.5. Data dictionary of Separation

Field	Type	Unit	Description	Since
horizontalNmi	double	nmi	Required horizontal separation distance, in nautical miles. Value range is [0, +inf] or zero (0) if not available. Example: 5.0.	0.9b

verticalFt	double	ft	Required vertical separation distance, in feet. Value range is [0, +inf] or zero (0) if not available. Example: 960.0.	0.9b

4.4. Flight Plan

The Flight Plan model stores flight plan related data fields defined in FAA Form 7233-1 [20]. The model is created for parsing live data feeds from the FAA's SWIM via the NASA's Sherlock ATM Data Warehouse. Table 4.6 lists the data dictionary of the flight plan model.

Table 4.6. Data dictionary of Flight Plan

Field	Type	Unit	Description	Since
aircraftId	String		Complete aircraft callsign or identification including the prefix "N" if applicable. Example: AAL123.	0.9b
flightPlanType	Flight Plan Type		Type of the flight plan to be used (see Section 4.4.1). Value is null if not available.	0.9b
aircraftType	String		Designator of the aircraft, i.e., aircraft type or special equipment. Example: BE9L.	0.9b
trueAirspeed Knots	int	knots	True airspeed, in knots. Value range is [0, +inf] or zero (0) if not available. Example: 456.	0.9b
departure Airport	String		Departure airport identifier code, or if unknown, the name of the airport. Note that this field may include the city name (or even the state name) if needed for clarity. Value is null if not available. Example: KSFO.	0.9b
proposed DepartureTime	long	ms	Proposed departure time, in milliseconds. If airborne, specify the actual or proposed departure time as appropriate. Value range is [0, +inf] or zero (0) if not available. Example: 1511983543000.	0.9b
actual DepartureTime	long	ms	Actual departure time, in milliseconds. If airborne, specify the actual or proposed departure time as appropriate. Value range is [0, +inf] or zero (0) if not available. Example: 1511983552000.	0.9b

cruiseAltitudeFeet	int	ft	Appropriate Visual Flight Rules altitude, in feet, to assist the briefer in providing weather and wind information. Value range is `[0, +inf]` or zero (0) if not available. Example: `29000`.	0.9b
routeOfFlight	String		The route of flight defined by using Navigational Aid (NAVAID) identifier codes and airways. Value is null if not available. Example: `KSFO.SSTIK3.EBAYE..AVE..KLAX`.	0.9b
destinationAirport	String		Destination airport identifier code, or if unknown, the airport name. Note that this may include city name (or even the state name) if needed for clarity. Value is null if not available. Example: `KLAX`.	0.9b
estimatedEnRouteDuration	long	ms	Estimated en route duration, in milliseconds. Value range is `[0, +inf]` or zero (0) if not available. Example: `10800000`.	0.9b
remarks	String		Remarks pertinent to Air Traffic Control or to the clarification of other flight plan information, such as the appropriate radiotelephony (callsign) associated with the designator field in aircraft ID. Items of a personal nature are not accepted. Value is null if not available. Example: `No over water`.	0.9b
fuelOnBoardDuration	long	ms	Duration of fuel on board, in milliseconds. Value range is `[0, +inf]` or zero (0) if not available. Example: `14400000`.	0.9b
alternateAirports	String		Alternate airport(s) if desired. Value is null if not available. Example: `KSAN, KPHX`.	0.9b
pilotInformation	String		Pilot’s information such as name, address, and telephone number. This may include sufficient information to identify home base, airport, or operator. Since this field value contains pilot identifying information such as name, address, and telephone number, the value is neither stored nor transmitted by TestBed.	0.9b

			Value is null if not available. Example: `John Doe, Moffett Field, (650) 555-1234`.	
numberAboard	int	persons	Total number of persons on board. Value range is `[0, +inf]` or zero (0) if not available. Example: `100`.	0.9b
colorOf Aircraft	String		Predominant colors. Value is null if not available. Example: `White`.	0.9b

4.4.1. Flight Plan Type

A `Flight Plan Type` instance indicates one of the types used in the flight plan model:

- Visual Flight Rules (VFR)
- Instrument Flight Rules (IFR)
- Defense Visual Flight Rules (DVFR)

Note that it is possible to have a composite flight plan type by enabling multiple types, e.g., “VFR/IFR.” Table 4.7 lists the data dictionary of the flight plan type.

Table 4.7. Data dictionary of Flight Plan Type

Field	Type	Unit	Description	Since
vfr	boolean		Flag indicates whether the VFR are used.	0.9b
ifr	boolean		Flag indicates whether the IFR are used.	0.9b
dvfr	boolean		Flag indicates whether the DVFR are used.	0.9b

4.5. Image Data

The `Image Data` model stores image specific binary data (see Section 4.2). A use case of doing image data exchange is to share a screen from a host computer to one or multiple client computers. Figure 4.3 shows a nominal setup that a Screen Capture adapter on the left-hand side captures a screen image every second and then publishes the image data to TestBed’s messaging support layer. The subscriber on the right-hand side runs a Screen Viewer adapter that displays the captured image on screen.

Figure 4.3. Screen sharing using image data

Refer to Table 4.2 for the data dictionary of the image data model. Common MIME types for the images are:

1. Graphics Interchange Format: `image/gif`
2. Joint Photographic Experts Group: `image/jpeg`
3. Portable Network Graphics: `image/png`

4.6. Multi-Purpose Interface Messages

The Multi-Purpose Interface (MPI) is a data exchange format used by the Aeronautical Datalink and Radar Simulator (ADRS) and external components [21, 22, 23]. To evaluate, demonstrate and support connectivity among components running at NASA Ames Research Center laboratories using the TestBed platform, a subset of the MPI messages has been implemented and included in the TestBed data exchange model. Table 4.8 lists the components as well as the laboratories used.

Table 4.8. Connectivity to components using MPI messages

	Component	Laboratory
1.	Multi-Aircraft Control System (MACS) [24]	• ATM TestBed Laboratory • Distributed Simulation Research Laboratory
2.	Time-Based Flow Management (TBFM) [25]	• ATM TestBed Laboratory
3.	Live, Virtual, Constructive Gateway (LVCGW) [26]	• ATM TestBed Laboratory • Distributed Simulation Research Laboratory
4.	Reconfigurable Image Generator (RiG) [27]	• ATM TestBed Laboratory • FutureFlight Central

In order to meet project-specific needs, software development teams may add new capabilities to the ADRS and some of the components by modifying their source code. As a result, certain components may no longer be able to connect to the same ADRS for data exchange due to incompatibility. Figure 4.4(a) illustrates this issue in an experiment run by connecting four components, MACS, TBFM, LVCGW, and RiG, via a modified ADRS, but the TBFM is no longer compatible with the modified ADRS. One potential solution is to update the TBFM so that it is compatible with the modified ADRS. However, this solution is not desired especially if a change in the ADRS would break all the connecting components.

The TestBed platform addresses this concern by connecting each component to a single ADRS and each ADRS connects to the messaging support layer. Figure 4.4(b) illustrates a nominal network diagram connecting the four components to the TestBed messaging support layer via individual ADRS instances. Since these components are connected to their compatible versions of ADRS, modifying a component and its connected ADRS, e.g., MACS and ADRS 1, will not affect the connectivity of the other components.

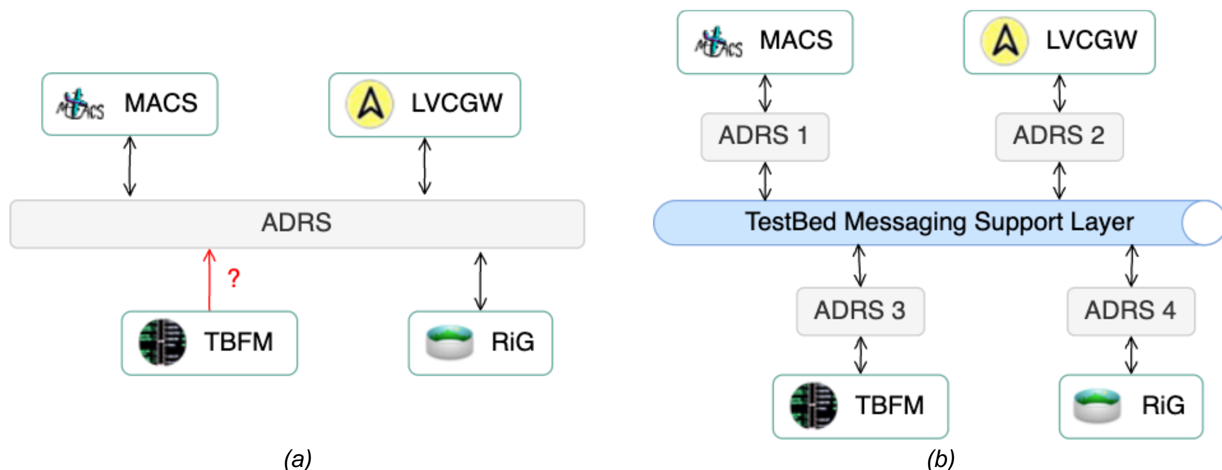


Figure 4.4. Connectivity among components using MPI messages: (a) without and (b) with TestBed

Table 4.9 lists the MPI messages in the ADRS that have been implemented in TestBed. The TestBed class names use camel case and the pattern AdrsMpi<XXX>MsgSt where the <XXX> values are listed in the first column, while the corresponding ADRS C data structure names use lowercase and the pattern adrs_mpi_<YYY>_msg_st where the <YYY> values are listed in the second column.

Table 4.9. Supported MPI messages in TestBed

TestBed Class Name[1] **AdrsMpi<XXX>MsgSt**	**ADRS C Data Structure** **adrs_mpi_<YYY>_msg_st**	**Description**
AcControl	ac_control	Aircraft control message.
AkRoute	ak_route	Host AK route message.
Arinc702	arinc_702	Aeronautical Radio, Incorporated (ARINC) 702 message defined in the flight management system.
Broadcast	broadcast	Broadcast message.
DateTime	date_time	Date time message.
DeleteAc	delete_ac	Delete aircraft message.
ExtendedFlightData	extended_flight_data	Extended flight data message.
ExtendedFlightPlan	extended_flight_plan	Extended flight plan message.
FlightData	flight_data	Flight data message.
FlightPlan	flight_plan	Flight plan message.
FlightState	flight_state	Flight state message.
Ident	ident	Identification message.
InitialClient	initial_client	Initial client message.
InitialServerResponse	initial_server_response	Initial server response message.
InitialState	initial_state	Initial state message.
MacsConfig	macs_config	MACS configuration message.
MacsControl	macs_control	MACS control message.
PilotInput	pilot_input	Pilot input message.
Request	request	Request message.
Scenario	scenario	Scenario message.
Track	track	Track message.
Trajectory	trajectory	Trajectory message.
Transaction	transaction	Transaction message.
XmlPayload	xml_payload	Extensible Markup Language (XML) payload message.

In addition, Table 4.10 lists the MPI messages in the LVCGW that have been implemented in TestBed. The TestBed class names use the pattern Mpi<XXX>Message where the <XXX> values

[1] In the package gov.nasa.sntb.messagingdatamodels.sndem.mpi.message.

are listed in the first column, while the corresponding LVCGW class names use the pattern `Msg<YYY>` where the `<YYY>` values are listed in the second column.

Table 4.10. Supported MPI messages in LVCGW

TestBed Class[2] `Mpi<XXX>Message`	LVCGW Message Class `Msg<YYY>`	Description
`AcTrackState`	`AcTrackState`	Track state of the intruder aircraft message.
`AcTrackStateOwnship`	`AcTrackStateOS`	Track state of the *ownship* aircraft message.
`FlightPlan`	`FlightPlan`	Flight plan message.
`FlightState`	`FlightState`	Flight state message.
`Handshake`	`Handshake`	Handshake message.
`Heartbeat`	`Heartbeat`	Heartbeat message.
`SaaBands`	`SaaBands`	Sense And Avoid (SAA) bands message.
`SaaFlightState`	`SaaFlightState`	SAA flight state message.
`SaaThreatResults`	`SaaThreatResults`	SAA threat results message.
`WellClearRecovery`	`WellClearRecovery`	Well clear recovery message.

4.7. Resolution

The `Resolution` model stores conflict resolution information of an Autoresolver's flight conflict. Table 4.11 lists the data dictionary of the resolution model.

Table 4.11. Data dictionary of Resolution

Field	Type	Unit	Description	Since
`resolutionId`	`String`		The resolution identifier. Example: `AAL123`.	`0.9b`
`maneuver`	`String`		Textual description of the maneuver an aircraft has to perform to avoid the flight conflict. Example: `Turn left by 10 degrees`.	`0.9b`
`altitudeFt`	`float`	`ft`	Target altitude, in feet. `NaN`[3] if not available. Example: `28000.0`.	`0.9b`
`distanceNmi`	`float`	`nmi`	Distance, in nautical miles, to hold. Value range is `[0, +inf]` or zero (0) if not available. Example: `10.0`.	`0.9b`

---

[2] In the package `gov.nasa.sntb.lvcgwadapter.mpi.message`.

[3] `NaN` represents a Not-a-Number value of types float or double.

headingDeg	float	deg	Target heading, in degrees. Value range is `[0, 360]` or `NaN` if not available. Example: `350.0`.	0.9b
speedKts	float	knots	Target speed, in knots. Value range is `[0, +inf]` or `NaN` if not available. Example: `254.80046`.	0.9b
waypoints	List<Waypoint>		List of waypoints to be used (see Section 4.7.1). Value is null if not available.	0.9b

4.7.1. Waypoint

A `Waypoint` instance represents a named, two-dimensional location. Table 4.12 lists the data dictionary of the waypoint data structure.

Table 4.12. Data dictionary of Waypoint

Field	Type	Unit	Description	Since
name	String		The name of the waypoint, which can be a fix name, an airport name, or a fix name with radial and distance. Example: `PUW257044`.	0.9b
latitude	double	deg	Latitude in degrees. Value range is `[-90, +90]` or `NaN` if not available. Example: `46.504688749082106`.	0.9b
longitude	double	deg	Longitude in degrees. Value range is `[-180, +180]` or `NaN` if not available. Example: `-118.2609576981683`.	0.9b

4.8. Task

The `Task` model represents a user task to be processed, either sequentially or in parallel, by a component. The model is developed for experiments requiring an automated simulation capability [28]. Figure 4.5 shows a nominal setup demonstrating the usage of the task model between two components, Task Provider and Task Consumer.

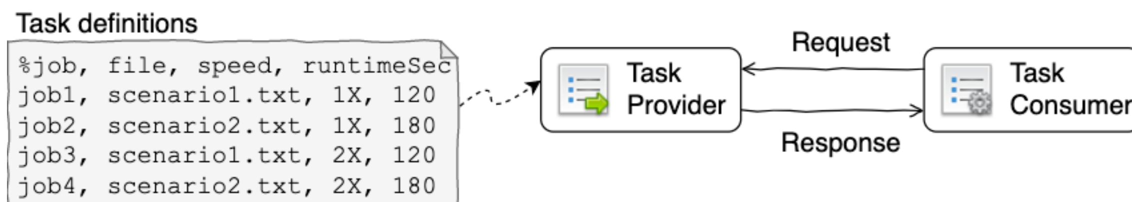


Figure 4.5. Task messages between Task Provider and Task Consumer

During an experiment startup, the task provider reads task definitions from a configurable property into a task list. Whenever a message requesting a task is received from the task consumer, the task provider will reply the next available task from the task list as a response

message. The request-response process continues until there are no more tasks available in the task list. This design supports running tasks in a sequential mode when a single task consumer is used, or concurrent modes when multiple task consumers are connected to the same task provider. Table 4.13 lists the data dictionary of the task model.

Table 4.13. Data dictionary of Task

Field	Type	Unit	Description	Since
`processorId`	`String`		Identification of the processor that handles this task. The value is null if the field is not available. Example: `abed3702.1.task-consumer`.	`0.9b`
`status`	`Task Status`		Status of this task (see Section 4.8.1). Example: `REQUESTED`.	`0.9b`
`parameters`	`Map<String, String>`		Mapping of parameter name to value to be passed to the task for execution. Keys are parameter names and values are parameter values.	`0.9b`

4.8.1. Task Status

A `Task Status` instance represents a stage, as an enumeration value, of a task execution. Here is the sequence of task status:

1. `REQUESTED`
2. `INITIALIZED`
3. `STARTED`
4. `CANCELLED` or `COMPLETED`

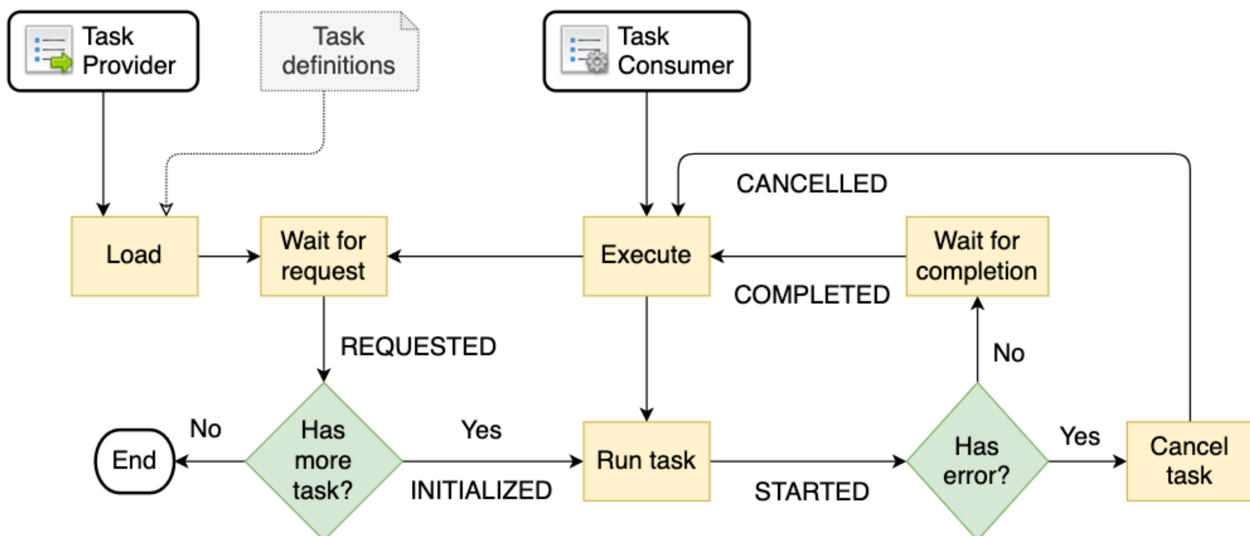


Figure 4.6. Flow diagram between Task Provider and Task Consumer

Figure 4.6 shows a flow diagram of the task status values between the task provider and the task consumer. Table 4.14 lists the data dictionary of the task status enumeration values.

Table 4.14. Data dictionary of Task Status

Enum	Description	Since
CANCELLED	Task has been cancelled.	0.9b
COMPLETED	Task is completed.	0.9b
INITIALIZED	Task is initialized but not yet started.	0.9b
REQUESTED	New task is requested.	0.9b
STARTED	Task has started and is running.	0.9b

4.9. Track

The `Track` model represents a single, three-dimensional location of a vehicle obtained by radar or sensor. Table 4.15 lists the data dictionary of the track model.

Table 4.15. Data dictionary of Track

Field	Type	Unit	Description	Since
vid	String		Vehicle identifier such as aircraft callsign. Value is null if not available. Field alias: `vehicleId`. Example: `AAL123`.	0.9b
latDeg	float	deg	Latitude, in degrees, of the vehicle's location. Value range is `[-90, +90]` or `NaN` if not available. Field alias: `latitudeDegrees`. Example: `34.907051`.	0.9b
lonDeg	float	deg	Longitude, in degrees, of the vehicle's location. Value range is `[-180, +180]` or `NaN` if not available. Field alias: `longitudeDegrees`. Example: `-117.620305`.	0.9b
altFt	float	ft	Indicated altitude, in feet, of the vehicle above mean sea level (MSL) on a standard day. Value is `NaN` if not available. Field alias: `altitudeFeet`. Example: `27059.473`.	0.9b
time	long	ms	Time, in milliseconds, when the track information was created. Value is zero (0) if not available. Example: `1512411933075`.	0.9b
gsKt	float	knots	Groundspeed, in knots, represents movement of the vehicle relative to the ground. Value is `NaN` if not available. Field alias: `groundspeedKnots`.	0.9b

			Example: `104.0`.	
crsDeg	float	deg	Flight course in degrees. By default, values are in degrees from true north. Value range is [0, 360] or NaN if not available. Field alias: courseDegrees. Example: `130.78125`.	0.9b
vsFpm	float	ft/min	Vertical speed in feet per minute. Positive values indicate climbing, and negative values indicate descending. Value is NaN if not available. Field alias: verticalSpeedFpm. Example: `16.0`.	0.9b

4.10. Trajectories

The Trajectories model stores mappings of type to trajectory that are used in Autoresolver. The trajectory types are algorithm-specific strings describing the nature of the trajectories such as “AAC,” “dead-reckon,” “flight plan,” or “maneuver.” Table 4.16 lists the data dictionary of the trajectories model.

Table 4.16. Data dictionary of Trajectories

Field	Type	Unit	Description	Since
map	Map< String, Trajectory >		Mappings of type to trajectory (see Section 4.10.1). Field alias: trajectoryMap.	0.9b
vid	String		Vehicle identifier such as aircraft callsign. Field alias: vehicleId. Example: `AAL123`.	0.9b

4.10.1. Trajectory

A Trajectory instance stores a list of tracks of a vehicle. The instance may represent a predicted path of a specific type such as dead reckoning and tracks are ordered by time values in ascending order. Table 4.17 lists the data dictionary of the Trajectory data structure.

Table 4.17. Data dictionary of Trajectory

Field	Type	Unit	Description	Since
vehicleId	String		Identification of the vehicle, e.g., aircraft callsign. Example: `AAL123`.	0.9b
tracks	List <Track>		List of tracks in the trajectory (see Section 4.9).	0.9b

4.11. Vehicle State

The `Vehicle State` model represents a single state of a vehicle. This model is an extension of the track model (see Section 4.9) with additional fields. Table 4.18 lists the data dictionary of the additional fields in the vehicle state model.

Table 4.18. Data dictionary of Vehicle State

Field	Type	Unit	Description	Since
`trueHeading`	`float`	`deg`	True heading in degrees. Value range is `[-90, +90]`. Example: `45.17`.	`0.9b`
`trueAirspeed`	`float`	`knots`	True airspeed in knots. Example: `326.45`.	`0.9b`
`indicatedAirspeed`	`float`	`knots`	Indicated airspeed in knots. Example: `301.12`.	`0.9b`

4.12. Vehicle True State

The `Vehicle True State` model represents a true state of a vehicle obtained from an Aircraft Simulation for Traffic Operations Research (ASTOR), a simulated aircraft having six-degree of freedom dynamic models, in the Air Traffic Operations Simulation (ATOS) distributed simulation platform running in the Air Traffic Operations Laboratory (ATOL) at NASA Langley Research Center [29]. Figure 4.7 shows a nominal diagram demonstrating the usage of the vehicle true state model in a connectivity test between NASA Ames and Langley Research Centers.

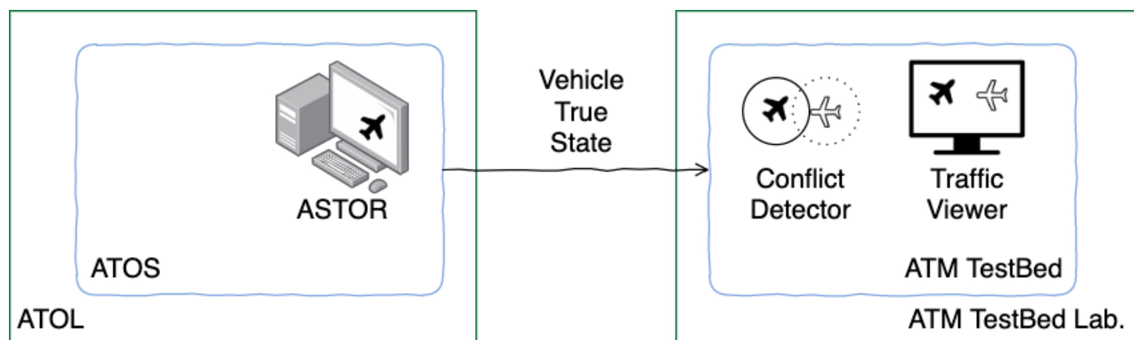


Figure 4.7. Connectivity between Air Traffic Operations Laboratory and ATM TestBed Laboratory

The ASTOR currently supports two versions, 1 and 2, of the vehicle true state models. The two versions are included in the TestBed data exchange model. To query the latest available version being defined in the model, the method `VehicleTrueState`[4]`.getVersion()` may be called and it will return an integer value. Alternatively, the methods `getV001()` and `getV002()` may also be called; supported versions are returned as non-null values. Table 4.19 lists the data dictionary of the vehicle true state model.

---

[4] In the package `gov.nasa.sntb.messagingdatamodels.sndem.simuniverse`.

Table 4.19. Data dictionary of Vehicle True State

Field	Type	Unit	Description	Since
v001	Vehicle True State[5]		Vehicle true state, in version 1 format (see Section 4.12.1). Value is null if not available.	1.5a
v002	Vehicle True State[6]		Vehicle true state, in version 2 format (see Section 4.12.2). Value is null if not available.	1.5a

4.12.1. Vehicle True State, Version 1

A Vehicle True State, defined in the sub-package v001, represents a true state of a vehicle from an ASTOR in the version 1 format. Table 4.20 lists the data dictionary of this data structure. The custom data structures used in this version are defined in the sub-package v001. Note that for numeric fields, -9999.0 represents an unknown value.

Table 4.20. Data dictionary of Vehicle True State, Version 1

Field	Type	Unit	Description	Since
simId	int		Unique identifier for simulated entity. C type: unsigned long. Example: 10895060.	1.5a
callSign	String		Callsign string. C type: char[128]. Example: NASA501.	1.5a
qualifierBits	int		Qualifier Bits. C type: unsigned long. Example: 0.	1.5a
realTime	long	ms	Current epoch time, number of milliseconds since January 1, 1970, 00:00:00 GMT. C type: long long. Example: 1456497096952.	1.5a
simTime	long	ms	Simulation epoch time, number of milliseconds since January 1, 1970, 00:00:00 GMT. C type: long long. Example: 1456497096952.	1.5a

[5] In the package gov.nasa.sntb.messagingdatamodels.sndem.simuniverse.v001.
[6] In the package gov.nasa.sntb.messagingdatamodels.sndem.simuniverse.v002.

truePosition3d	`Position3d`		Position of the vehicle in the three-dimensional space (see Section 4.12.1.1).	`1.5a`
topodetic Velocity	`float[3]`	`ft/ sec`	Wanted velocity vector, North-East-Down (NED) frame, in feet per second. Example: `[-191.0, -84.0, 0.0]`.	`1.5a`
topodetic Acceleration	`float[3]`	`ft/ sec/ sec`	Wanted acceleration, x, y, z, in feet per second per second. Example: `[0.0, 0.0, 0.0]`.	`1.5a`
body Orientation	`Orientation`		Orientation of the vehicle body (see Section 4.12.1.3). Optional.	`1.5a`
bodyAngular Rate	`float[3]`	`deg/ sec`	Angular rate, in degrees per second, of the vehicle body. Optional. Example: `[0.0, 0.0, 0.0]`.	`1.5a`
indicated Altitude	`float`	`ft`	Indicated altitude in feet. This is the value shown on the altimeter. Example: `925.0`.	`1.5a`
pressure Altitude	`float`	`ft`	Pressure altitude in feet. This is the altitude above the standard datum plane; the value shown on the altimeter when it is set to a standard pressure of 29.92 inHg. Optional. Example: `925.0`.	`1.5a`
altimeter Setting	`float`	`inHg`	Altimeter setting in inHg. The value of the atmospheric pressure used to adjust the sub-scale of a pressure altimeter. Optional. Example: `256.0`.	`1.5a`
indicated Airspeed	`float`	`knots`	Airspeed, in knots, read directly from the airspeed indicator. Optional. Example: `126.0`.	`1.5a`
trueAirspeed	`float`	`knots`	The speed, in knots, of the aircraft relative to the airmass in which it is flying. Example: `120.0`.	`1.5a`
calibrated Airspeed	`float`	`knots`	The indicated airspeed, in knots, corrected for instrument and position error. Optional. Example: `122.0`.	`1.5a`

magHeading	float	deg	Magnetic heading in degrees: • Heading of x-body axis with respect to magnetic north. • Headings are angles, range is between 0 and 360. • Angles are in degrees. Optional. Example: `175.13336`.	1.5a
magneticTrack	float	deg	Magnetic track angle in degrees. Direction of projection of velocity vector onto ground with respect to magnetic north. Optional. Example: `175.13336`.	1.5a
magnetic Variation	float	deg	Magnetic variation in degrees. Magnetic variation at three-dimensional position and simulation time. Optional. Example: `10.602112`.	1.5a
aircraftWeight	float	lbs	Aircraft weight in pounds. Optional. Example: `2300.0`.	1.5a
fuelWeight	float	lbs	Fuel weight in pounds. For logging and post-analysis use only. Optional. Example: `210.0`.	1.5a
landingGear	float	%	Landing gear percentage. Optional, only needed for display. Example: `0.0`.	1.5a
speedBrakes	float	%	Speed brakes percentage. Optional, only needed for display. Example: `0.0`.	1.5a
vehicleType	String		Vehicle type. C type: `char[128]`. Optional, only needed for display. Example: `SR22`.	1.5a
vehicle Operator	String		Vehicle operator. C type: `char[128]`. Optional, only needed for display. Example: `NASA`.	1.5a

deadReckoning Algorithm	long		Enumeration of the dead-reckoning algorithm to use. Optional. Example: `0`.	1.5a
lights	boolean		Flag indicates whether lights are on (true) or not (false). Optional, only needed for display. Example: `false`.	1.5a
freeze	boolean		Flag indicates whether the simulator is frozen (true) or not (false). If true, the simulator is not updating, and dead reckoning is suspended. Optional. Example: `false`.	1.5a
svbuffer	String		Packed buffer to be used for topic sub-versioning. C type: `char[320]`. Example: `svbuffer`.	1.5a

4.12.1.1. Position 3D

A `Position 3D` represents a location in the three-dimensional space. Table 4.21 lists the data dictionary of this data structure.

Table 4.21. Data dictionary of Position 3D

Field	Type	Unit	Description	Since
position2d	Position2d		Position of the vehicle in the two-dimensional space (See Section 4.12.1.2).	1.5a
geodetic Altitude	float	ft	Altitude in feet. Example: `5425.0`.	1.5a

4.12.1.2. Position 2D

A `Position 2D` represents a location in the two-dimensional space. Table 4.22 lists the data dictionary of this data structure.

Table 4.22. Data dictionary of Position 2D

Field	Type	Unit	Description	Since
latitude	double	deg	Latitude in degrees. Example: `36.974036693573`.	1.5a
longitude	double	deg	Longitude in degrees. Example: `-76.70776605606079`.	1.5a

4.12.1.3. Body Orientation

A `Body Orientation` represents the orientation of a vehicle body. Table 4.23 lists the data dictionary of this data structure.

Table 4.23. Data dictionary of Body Orientation

Field	Type	Unit	Description	Since
`yaw`	`float`	deg	Yaw in degrees. Example: `164.53125`.	`1.5a`
`pitch`	`float`	deg	Pitch in degrees. Example: `2.0`.	`1.5a`
`roll`	`float`	deg	Roll in degrees. Example: `0.0`.	`1.5a`

4.12.2. Vehicle True State, Version 2

A `Vehicle True State`, defined in the sub-package `v002`, represents a true state of a vehicle from an ASTOR in the version 2 format. When comparing with the version 1, the only change is the `qualifierBits` field that it is declared as `long` type in the version 2 instead of `int` type in the version 1. Table 4.24 lists the data dictionary of the updated field. Note that all the custom data structures used in this version are defined in the sub-package `v002`.

Table 4.24. Updated data dictionary of Vehicle True State, Version 2

Field	Type	Unit	Description	Since
`qualifierBits`	`long`		Qualifier Bits. C type: `long long`. Example: `0`.	`1.5a`

Appendix A. Version History

In order to support high-fidelity simulations among NASA and communities, the SNDEM has been undergoing improvements and modifications. Table A.1 lists the version history of the changes introduced in the data exchange model among TestBed versions. This document focuses on the latest version `2.0a`.

Table A.1. Version history

Version	Changes	Notes
`0.9a`	• None	• Released on February 10, 2016. • This initial version was effective until the Boeing 2018 ecoDemonstrator flight test.
`0.9b`	• Supported field aliases to reduce message sizes, e.g., "`src`" for "`source`" • Omitted long fields with zero (0) values	• Released on June 8, 2017. • A primary goal was to reduce message sizes.

	• Converted byte array contents into *Base64* [19] strings • Converted character array contents into Java strings • Supported special floating numbers: `NaN`, `-Infinity`, and `Infinity`	

`1.0a`	• Moved the following `SNDEM` fields to the "`metaInfo`" field:
	  ○ "`source`"
	  ○ "`timePublished`"
	  ○ "`timeSubscribed`"
		• Released on September 27, 2018.
		• This version allowed subclasses to define fields using these three names.
		• The changes allowed TestBed to add new fields without affecting subclasses in the future.

`1.5a`	• None	• Released on May 24, 2019. • No changes to the data exchange model.
`2.0a`	• Added "`userDefinedData`" field to `SNDEM`. • Added "`version`" field to the "`metaInfo`" field.	• Released on September 30, 2019. • This version supported user defined data and message versioning.

Table A.2 lists an example track message, in JSON format, to illustrate the changes among the versions. Note that the message of the version `2.0a` also includes a user-defined data named "engine."

Table A.2. Example track messages

Version	Message	Version	Message

`0.9a`	`{`
	`"vehicleId": "BOE069",`
	`"latitudeDegrees": 48.133335,`
	`"longitudeDegrees": -111.13718,`
	`"altitudeFeet": 27004.1,`
	`"trackTime": 1523829782000,`
	`"groundspeedKnots": 389.2,`
	`"trueHeadingDegrees": 123.22178,`
	`"verticalSpeedFpm": -96.3,`
	`"source": "EcodAdapter.1",`
	`"timePublished": 1520526488734,`
	`"timeArrived": 0`
	`}`
		`0.9b`	`{`
			`"vid": "BOE069",`
			`"latDeg": 48.133335,`
			`"lonDeg": -111.13718,`
			`"altFt": 27004.1,`
			`"time": 1523829782000,`
			`"gsKt": 389.2,`
			`"crsDeg": 123.22178,`
			`"vsFpm": -96.3,`
			`"src": "EcodAdapter.1",`
			`"tpub": 1523829795652`
			`}`

`1.0a`
`1.5a`
	`{`
	`"vid": "BOE069",`
	`"latDeg": 48.133335,`
	`"lonDeg": -111.13718,`
	`"altFt": 27004.1,`
	`"time": 1523829782000,`
	`"gsKt": 389.2,`
	`"crsDeg": 123.22178,`
		`2.0a`	`{`
			`"vid": "BOE069",`
			`"latDeg": 48.133335,`
			`"lonDeg": -111.13718,`
			`"altFt": 27004.1,`
			`"time": 1523829782000,`
			`"gsKt": 389.2,`
			`"crsDeg": 123.22178,`

	"vsFpm": -96.3,
	"meta": {
	"src": "EcodAdapter.1",
	"tpub": 1561586962076
	}
	}
			"vsFpm": -96.3,
			"meta": {
			"src": "EcodAdapter.1",
			"ver": "2.0a",
			"tpub": 1561586962076
			},
			"_udd": {
			"registry": {
			"engine": {
			"model": "boost",
			"rpm": "12345"
			}
			}
			}
			}

5. References

1. Palopo, K., Chatterji, G. B., Guminsky, M. D., and Glaab, P. C. (2015) "Shadow Mode Assessment using Realistic Technologies for the National Airspace System (SMART NAS) Test Bed Development," AIAA Modeling and Simulation Technologies Conference, Dallas, TX, 22-26 June 2015.
2. Robinson, J. E., Lee, A., and Lai, C. F. (2017) "Development of a High-Fidelity Simulation Environment for Shadow-Mode Assessments of Air Traffic Concepts," Royal Aeronautical Society: Modeling and Simulation in Air Traffic Management Conference, London, UK, 14-15 November 2017.
3. Chan, W., Barmore, B., Kibler, J., Lee, P., O'Connor, N., Palopo, K., Thipphavong, D., and Zelinski, S. (2018) "Overview of NASA's Air Traffic Management - Exploration (ATM-X) Project," AIAA 2018-3363, AIAA Aviation Forum, AIAA Aviation Technology, Integration, and Operations Conference, Atlanta, GA, 25-29 June 2018.
4. Aponso, B. L. (2014) "Simulation Technology at NASA," National Academies' Workshop on Opportunities for the Employment of Simulation in U.S. Air Force Training Environments, Dayton, OH, November 17-19, 2014.
5. Dorighi, N. S. and Sullivan, B. T. (2003) "FutureFlight Central: A Revolutionary Air Traffic Control Tower Simulation Facility," AIAA-2003-5598, AIAA Modeling and Simulation Technologies Conference and Exhibit, 11-14 August 2003.
6. Eshow, M. M., Lui, M., and Ranjan, S., (2014) "Architecture and Capabilities of a Data Warehouse for ATM Research," 2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC), Colorado Springs, CO, 2014, pp. 1E3-1-1E3-14.
7. Erzberger, H., Lauderdale, T. A., and Chu Y-C (2010), "Automated Conflict Resolution, Arrival Management and Weather Avoidance for ATM," 27th Congress of International Council of the Aeronautical Sciences, Nice, France, 19-24 September 2010.
8. "Boeing: ecoDemonstrator" (Online). https://www.boeing.com/principles/environment/ecodemonstrator. Boeing. Retrieved 2020/02/11.
9. Coppenbarger, R. A., Mead, R. W., and Sweet, D. N. (2007), "Field Evaluation of the Tailored Arrivals Concept for Datalink-Enabled Continuous Descent Approach," AIAA 2007-7778, 7th AIAA Aviation Technology, Integration and Operations Conference (ATIO), Belfast, Northern Ireland, 18-20 September 2007.
10. Thipphavong, D. P., Apaza, R. D., Barmore, B. E., Battiste, V., Burian, B. K., Dao, Q. V., Feary, M. S., Go, S., Goodrich, K. H., Homola, J. R., Idris, H. R., Kopardekar, P. H., Lachter, J. B., Neogi, N. A., Ng, H. K., Oseguera-Lohr, R. M., Patterson, M. D., and Verma, S. A.

(2018) “Urban Air Mobility Airspace Integration Concepts and Considerations,” AIAA 2018-3676, 2018 Aviation Technology, Integration, and Operations Conference, Atlanta, GA, 25-29 June 2018.
11. “Java SE - Downloads | Oracle Technology Network | Oracle” (Online). https://www.oracle.com/technetwork/java/javase/downloads/. Oracle. Retrieved 2020/02/12.
12. “Google Java Style Guide” (Online). https://google.github.io/styleguide/javaguide.html. Google. Retrieved 2020/02/12.
13. Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A., “The Java® Language Specification – Java SE 8 Edition” (Online). https://docs.oracle.com/javase/specs/jls/se8/html/index.html. Oracle. Retrieved 2020/02/10
14. The National Imagery and Mapping Agency (NIMA) (2000), “Department of Defense World Geodetic System 1984, Its Definition and Relationships With Local Geodetic Systems,” NIMA Technical Report TR8350.2, 3 January 2000.
15. Bray, T., “RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format” (Online). https://tools.ietf.org/html/rfc8259. Internet Engineering Task Force. Retrieved 2020/02/07.
16. “GitHub - google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back” (Online). https://github.com/google/gson. Google. Retrieved 2020/02/12.
17. Ging, A., Engelland, S., Capps, A., Eshow, M., Jung, Y., Sharma, S., Talebi, E., Downs, M., Freedman, C., Ngo, T., Sielski, H., Wang, E., Burke, J., Gorman, S., Phipps, B., Ruszkowski, L. M. (2018) “Airspace Technology Demonstration 2 (ATD-2) Technology Description Document (TDD),” NASA-TM-2018-219767, NASA Technical Memorandum, March 1, 2018.
18. Lai, C. F. (2020) “Air Traffic Management TestBed Traffic Viewer: Developer’s Guide,” NASA-TM-2020-220511, NASA Technical Memorandum, April 16, 2020.
19. Freed, N. and Borenstein, N. (1996) “RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies,” https://doi.org/10.17487/RFC2045. Internet Engineering Task Force. Retrieved 2020/02/07.
20. Cunningham, S. (1982) “Form 7233-1 – Flight Plan Document Information,” https://www.faa.gov/forms/index.cfm/go/document.information/documentid/186159. Federal Aviation Administration. Retrieved 2020/02/12.
21. Jovic, S. (2017), “Live Virtual Constructive (LVC) Interface Control Document for the LVC Gateway,” NASA-TM-2017-219499, NASA Technical Memorandum, January 19, 2017.
22. Prevot, T., Palmer, E., Smith, N., and Callantine, T. (2002), “A Multi-Fidelity Simulation Environment for Human-in-the-Loop Studies of Distributed Air Ground Traffic Management,” AIAA 2002-4679, AIAA Modeling and Simulation Technologies Conference and Exhibit, Monterey, CA, 5-8 August 2002.
23. Prevot, T., Smith, N., Palmer, E., Mercer, J., Lee, P., Homola, J., and Callantine, T. (2006) “The Airspace Operations Laboratory (AOL) at NASA Ames Research Center,” AIAA 2006-6112, AIAA Modeling and Simulation Technologies Conference and Exhibit, Keystone, CO, 21-24 August 2006.
24. Prevot, T. and Mercer, J. (2007) “MACS: A Simulation Platform for Today’s and Tomorrow’s Air Traffic Operations,” AIAA-2007-6556, AIAA Modeling and Simulation Technologies (MST) Conference and Exhibit, Hilton Head, SC, 20-23 August 2007.
25. “Time Based Flow Management” (Online). https://www.faa.gov/nextgen/cip/tbfm/. Federal Aviation Administration. Retrieved 2020/02/05.
26. Murphy, J. R., Jovic, S., and Otto, N. M. (2015), “Message Latency Characterization of a Distributed Live, Virtual, Constructive Simulation Environment,” AIAA SciTech, Kissimmee, FL, 5-9 January 2015.

27. Archdeacon, J. L., Iwai, N. H., Kato, K. H., and Sweet, B. T. (2017) “Reconfigurable Image Generator,” U.S. Patent 9,583,018, February 28, 2017.
28. Arneson, H., Evans, A. D., Li, J., and Wei, M. Y. (2017) “Development and Validation of an Automated Simulation Capability in Support of Integrated Demand Management,” AIAA/Royal Aeronautical Society Flight Simulation, London, UK, 14-16 November 2017.
29. Peters, M. E., Ballin, M. G., and Sakosky, J. S. (2012), “A Multi-Operator Simulation for Investigation of Distributed Air Traffic Management Concepts,” AIAA 2002-4596, AIAA Modeling and Simulation Technologies Conference and Exhibit, Monterey, CA, 5-8 August 2002.