Agenda Item 2: **Implementation of systems for Aeronautical Information Data Exchange and Aeronautical Data (AIXM)**

GREPECAS Project G2

(Presented by the Secretariat)

SUMMARY

This working paper presents to the Meeting the current implementation status of GREPECAS Project G2 and the new proposal for the revision of AIXM Project.

REFERENCES

- Annex 15 to the Convention of International Civil Aviation
- SAM/AIM Multilateral Meetings
- Report of the Fourth Meeting of the Programmes and Projects Review Committee (PPRC/4)

ICAO Strategic Objectives	*A – Safety* *B – Air Navigation Capacity and Efficiency* *E – Environmental protection*

1. **Background**

1.1 The implementation of GREPECAS Project G2, after having a Coordinator, presented developments in the programme of activities.

1.2 During all SAM/AIM meetings the importance of Quality Certification to guarantee the quality of the data to be transferred was emphasized, as well as the importance of AIM training to optimize the results.

1.3 In PPRC/4 Meeting the continuation of Project G2 was analyzed.

2. **Analysis**

2.1 During PPRC/4 Meeting the project was reconsidered taking into account the items not yet completed. The terms of the work of Project G2 are described in **Appendix A** to this working paper.

2.2 The Meeting shall recall that in SAM/AIM/8 Meeting, as part of AIXM products to be analyzed, the Project Coordinator selected four documents to be considered in the phase of development. In this regard, four documents of EUROCONTROL were translated, which are detailed in the following paragraphs.

2.3 During the SAM/AIM/9 Meeting, four documents from EROCONTROL as a starting point to analyze the operation of AIXM systems were approved, which are *Temporality Model* (**Appendix B**), *AIXM Conceptual Model* (**Appendix C**), *AIXM Application Schema Generation*

(**Appendix D**) and *Feature Identification and Reference* (**Appendix E**). These documents should be followed as guides for AIXM implementation and thus move forward to transit to interoperability.

2.4 In addition, the SAM/AIM/9 Meeting considered important to make AIXM tests on 30 March 2017, between Argentina and Panama, and on 27 April 2017, between Brazil and Peru. Unfortunately, changes in the AIM officers of Panama and Argentina, as well as difficulties in the infrastructures of Brazil and Peru, prevented them from developing the tests. The Meeting should consider the possibility to develop other exchange tests between the States that have already implemented the AIXM.

2.5 Considering the programme of activities and the objective of Project G2, it is necessary to inquire States on the Implementation Action Plans of the data exchange systems. The Secretariat has prepared a template to consult States on the current AIXM implementation situation, which is presented in **Appendix F** to this working paper. It would be important to update the template in order to follow up AIXM implementation in the SAM Region.

2.6 A very important issue is the training of AIM technical personnel in the use the formats, as well as the information to be exchanges. An initial, as well as recurrent, training should be considered, in order to strengthen the knowledge regarding the use of the data exchange model.

3. Conclusion

3.1 In view of the above, the States should be aware that AIXM implementation is a very important step to achieve data and systems interoperability.

3.2 In addition, the need to have professional personnel to assist the AIM on the use of these formats is evident, and operators should be trained to include the information to be exchanged.

3.3 It is also evident that professional personnel to assist the AIM in the use of these formats is required, as well as to train the operators to include the information to be exchanged.

3.4 The Meeting should agree with the Project Coordinator on the dates in which the exchange tests would be made. For this purpose, States which are in conditions to make the tests and the data to be exchanged should be taken into account.

3.5 The Meeting shall agree with the Project Coordinator on the dates in which the exchange tests should be made. For this purpose, the States that are ready to make the tests and the data to be exchanged should be taken into account.

4. Suggested action

4.1 The Meeting is invited to:

a) analyze the information contained in Appendix A and to update Project G2 as necessary;

b) establish the dates for the data exchange tests between the States that are ready;

c) provide the implementation action plan of the data exchange systems of those States that have not done so.

- - - - - -

APPENDIX A

SAM Region	PROJECT DESCRIPTION (DP)	DP N° G2	
Programme	**Title of the Project**	**Start**	**End**
AIM (ICAO Programme Coordinator: Jorge Armoa Cañete)	G2: Implementation of Aeronautical Information Exchange Systems (AIXM) (SAM) Project coordinator: Eng. Karina Calderón Experts contributing to the project: SAM/AIM/IG	01/03/12	01/12/17
Objective	Prepare an action plan to be implemented by States for the application of the aeronautical information/data exchange model.		
Scope	The scope of the project contemplates the evaluation and identification of automation levels associated to the integration of the aeronautical information and data exchange model in the Region, through surveys, the identification of database providers, and the follow-up on the development of SARPs on this matter.		
Metrics	Number of States that have implemented an Action Plan for data exchange systems.		
Goals	Complete all the documentation needed by States before 31/12/16. Achieve AIXM implementation in 40% of States for 2018. Achieve AIXM implementation in 75% of States for 2019.		
Strategy	Project activities will be coordinated among project members, the Project Coordinator, and the Programme Coordinator, mainly through teleconferences (GoToMeeting application). Seminars/meetings are scheduled in accordance with work programme activities. The Project Coordinator will coordinate with the Programme Coordinator for the inclusion of additional experts, if warranted by the tasks and work to be performed. Coordination will take place between the CAR and SAM Regions. The results of the work done will be submitted to the consideration and review of State experts in the form of a final consolidated document for analysis, review, and approval, and for presentation to the GREPECAS PPRC by the Programme Coordinator.		

Rationale	Integrate aeronautical information so as to permit the interoperability of ATM systems while preserving safety, applying the information exchange models.
Related projects	This project is related to Project G3 “*Implementation of the Quality Management Systems in the AIM units in SAM States*”.

Project deliverables	**Relationship with the performance-based regional plan (PFF)**	**Responsible party**	***Status of Implementation**	**Delivery date**	**Comments**
Survey of the provision of IAIP, using a table.	D-ATM	ICAO coordinator		16/03/12	Finalised on schedule at the SAM/AIM meeting.
Circulation of IAIP survey to States	D-ATM	ICAO coordinator		16/03/12	Finalised on schedule at the SAM/AIM meeting.
Collection and updating	D-ATM	ICAO coordinator		16/03/12	Finalised on schedule at the SAM/AIM meeting.
Collection of experiences in SAM States with the electronic AIP	D-ATM	ICAO coordinator		16/03/12	Finalised on schedule at the SAM/AIM meeting.
Develop AIXM action plan	D-ATM	ICAO coordinator		24/04/15	Finalised on schedule.
AIXM documentation collection	D-ATM	ICAO coordinator		22/05/15	Finalised on schedule.
AIXM documentation translation	D-ATM	ICAO		10/07/15	Finalised on schedule.

AIXM documentation revision	D-ATM	ICAO coordinator		21/08/15	Finalised on schedule.
Documentation validation	D-ATM	ICAO coordinator		30/11/16	
Develop document describing AIXM tests steps	D-ATM	ICAO coordinator		28/02/17	
AIXM tests	D-ATM	ICAO coordinator		30/11/17	
Transmission and reception of tests results data	D-ATM	ICAO coordinator		19/05/17	
AIXM seminar	D-ATM	ICAO coordinator		02/10/15	Finalised on schedule.
AIXM management concept guidance material	D-ATM	ICAO coordinator		27/12/16	
Resources required	Designation of experts in the execution of some of the deliverables. Commitment by States to support the coordinators and experts.				

**Grey*	*Task not started*
Green	*Activity underway as scheduled*
Yellow	*Activity started with some delay but expected to be completed on time*
Red	*It has not been possible to implement this activity as scheduled; mitigating measures are required*

APPENDIX B

AIXM 5

TEMPORALITY MODEL

AIXM 5

Temporality Model

Aeronautical Information Exchange Model (AIXM)

Copyright: 2010 - EUROCONTROL and Federal Aviation Administration

All rights reserved.

This document and/or its content can be download, printed and copied in whole or in part, provided that the above copyright notice and this condition is retained for each such copy.

For all inquiries, please contact:

Navin VEMBAR - Navin.Vembar@faa.gov

Eduard POROSNICU - eduard.porosnicu@eurocontrol.int

Part Edition No.		Part Edition Issue Date	Part Author	Reason for Change
0.1	Draft	24 APR 2007	Design Team	Initial Draft
0.2	Draft	04 JUN 2007	Design Team	Updated after discussions in St. Louis and Frankfurt.
0.3	Draft	10 JUN 2007	Design Team	Updated after comments from AIXM FG #8 Meeting and from Edna.
0.4	Proposed	15 JUL 2007	Design Team	Removed "Static" Time Slices from the model. Re-organised the presentation of the different kinds of Time Slices.
0.5	Proposed	12 NOV 2007	Design Team	Clean-up for first public version.
0.6	Proposed	01 FEB 2010	Design Team	Describe PropertiesWithSchedule concept introduced in AIXM 5.1 (see chapter 2.8) Include UML diagrams from AIXM 5.1 model
1.0	Released	15 SEP 2010	Design Team	Released version, to be used as baseline for AIXM 5.1 implementations.

Table of Contents

1. The need for a temporality model 4
2. Building the Temporality Model 5
2.1 (step 1) Time varying properties 5
2.2 (step 2) The Time Slice model 6
2.3 (step 3) Temporary events – digital NOTAM 7
2.4 (step 4) Current Status - SNAPSHOT Time Slices 9
2.5 (step 5) Data exchange – need for PERMDELTA Time Slices 9
2.6 (step 6) Data exchange – corrections 11
2.7 Properties with schedule 13
2.8 Temporality applied to the Abstract Model 17
3. Application aspects 19
3.1 BASELINE Time Slices with undetermined end of validity 19
3.2 SequenceNumber values 19
3.3 Feature end of life 20
3.4 "Delta" for complex properties 21
3.5 "Delta" for multi-occurring properties 22
3.6 Identifying the feature affected by a DELTA Time Slice 22
3.7 Canceling a Time Slice (abandoned changes) 23
3.8 Overlapping TimeSlices and corrections 24
3.9 Other implementation considerations 25
4. Usage examples 26
4.1 Navaid example 26
4.2 Feature creation (commissioning) 27
4.3 Permanent change 28
4.4 Digital NOTAM 28
4.5 End of life (decommissioning) 29
4.6 Complete feature histories 30

1. The need for a temporality model

Time is an essential aspect on the aeronautical information world, where change notifications are usually made well in advance of their effective dates. Aeronautical information systems are usually requested to store and to provide both the current situation and the future changes. The expired information needs to be archived for legal investigation purposes.

For operational[1] reasons, a distinction is usually made between:

- ***permanent changes*** (the effect of which will last until the next permanent change or until the end of the lifetime of the feature) and
- ***temporary status*** (changes of a limited duration that are considered to be overlaid on the permanent state of the feature).

A temporary change includes the concepts of overlay and reversion. The temporary change is overlaid on the permanent feature state. When the temporary change ends, the temporary changes no longer apply and we revert back to the permanent feature state.

Note that, from an operational point of view, “temporary status” also includes the concept of “temporary features”. However, from the AIXM point of view, temporary features are in no way different from normal features. The feature is created and withdrawn, just that the life span is shorter than usual.

In order to satisfy the temporal requirements of aeronautical information systems, AIXM must include an exhaustive temporality model, which enables a precise representation of the states and events of aeronautical features. In particular, this shall enable the development and the implementation of **digital NOTAM**. By digital NOTAM we mean replacing the free text contained in a NOTAM message with structured facts, which enable the automated processing of the information.

A general temporal model should be uniformly applied to all aeronautical feature types and the temporality concept should be abstracted from the task of modeling object properties. At the conceptual level, the model should describe the temporal evolution of the features, as they occur in the real world. This shall be done in compliance with the following rules:

- Completeness - all temporal states must be representable;
- Minimalism - use of minimal number of elements;
- Consistency - no reuse of elements with different meaning;
- Context-free - meaning of (atomic) elements independent of context; no functional dependency of (atomic) elements at the data encoding level;

The data exchange specification shall support the conceptual model. In addition, convenience elements (“views”) may be introduced in the data exchange specification in order to facilitate the operations. This means that the data exchange specification may deviate from the “minimalism” rule.

[1] For example, systems that produce printed aeronautical documentation (AIP, charts) tend to ignore temporary status information; only the static data is represented on such printed products.

2. Building the Temporality Model

In order to explain the AIXM Temporality Model, this chapter follows a step-by-step approach, in which the elements that compose this model are added progressively in order to satisfy the operational needs.

2.1 (step 1) Time varying properties

There are two levels at which aeronautical feature instances are affected by time:

- Every feature has a start of life and an end of life;
- The properties of a feature can change within the lifetime of the feature; this includes the possibility for a property to not be defined over a time period.

The start of life and the end of life may also be considered as feature properties (attributes). This gives the following high-level list of properties for any AIXM feature:

- a global unique identifier;
- the start of life (date and time);
- the end of life (date and time);
- attributes and associations that qualify, quantify or relate in some form that feature.

It is considered that any feature property may change in time, except for the global unique identifier. This is a key assumption of the AIXM Temporality model.

The first step in the construction of the AIXM temporality model is represented by the diagram below, which shows the values of a feature's properties (P1, P2, … P5) along a timeline.

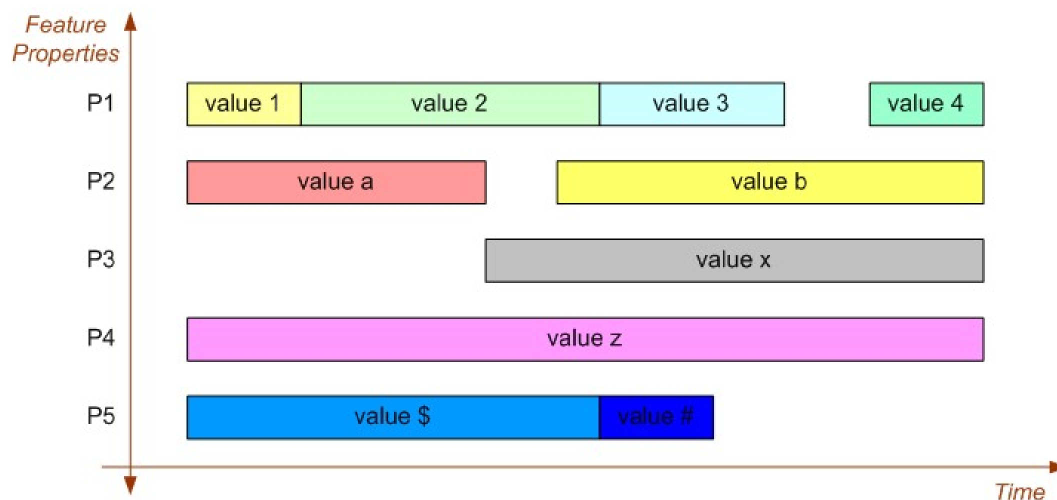


Figure 1 – Time varying properties

Discussion: ***Can the start of life and the end of life properties of a feature vary in time?***

At first sight, probably not. A feature is created at a moment in time and will cease to exist at another moment in time. But this is true only when considering the already known history of a feature. When exchanging data about the future, there might be situations where the start/end of life is planned to happen at a certain date/time and this date might change.

Therefore, we have to include the start/end of life of a feature in the time varying properties list.

2.2 (step 2) The Time Slice model

The temporality model adopted2 by AIXM describes feature events and states. An event is a change of one or more feature properties. A state is the feature property set valid over a time period. An event occurs at the transition between states. In the diagram below events are located at the vertical cuts while states are represented as the feature property set between events.

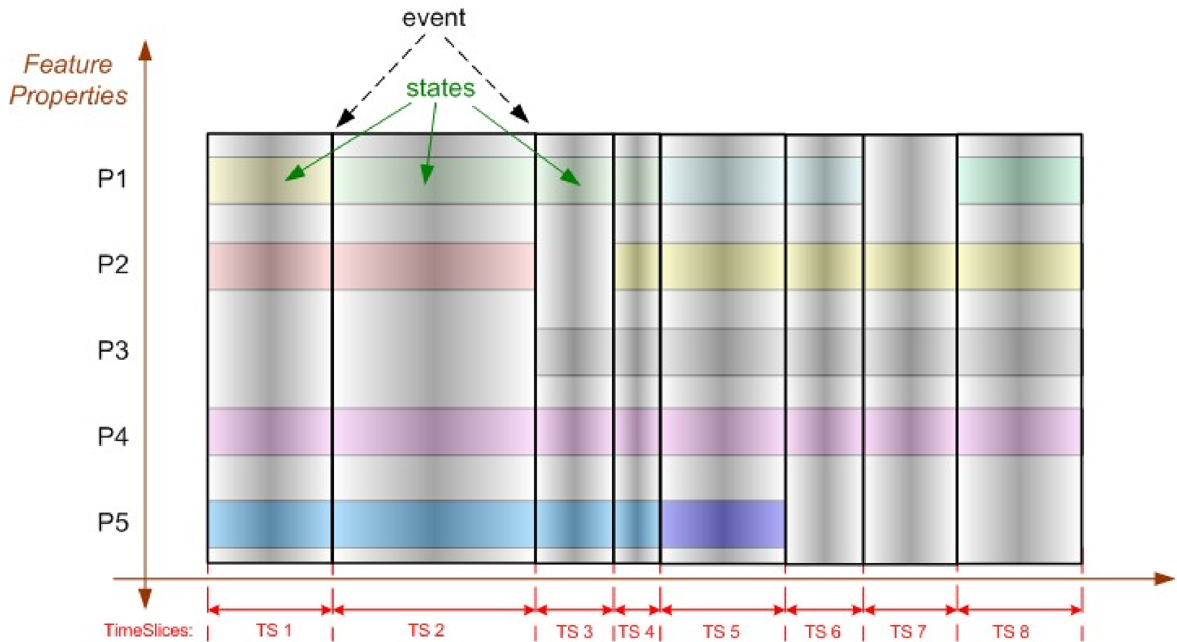


Figure 2 – Events and States

In order to describe the feature properties during states and events, the time varying properties of every feature are encapsulated in a container called "Time Slice". The history of the feature is described with "state" Time Slices, each containing the values of the time varying properties between two consecutive changes (events). Each Time Slice has maximum one value for each property and one specified validity period. In an UML diagram, the basic Time Slice concept is represented as below:

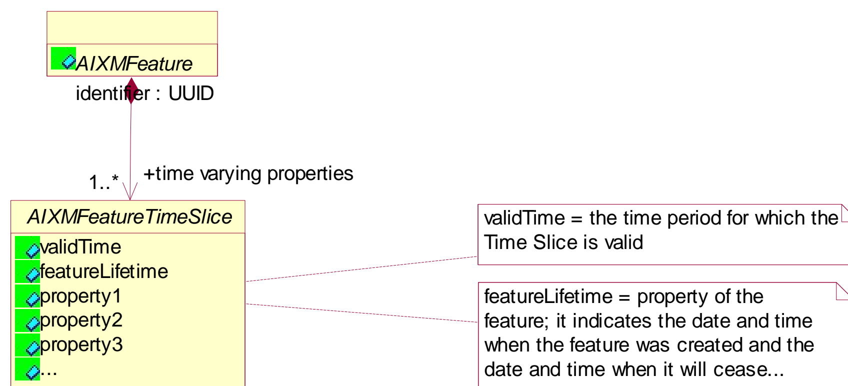


Figure 3 – AIXMFeatureTimeSlice

[2] The AIXM Time Slice model is based on the ISO 19136 (GML) timeslice concept.

Discussion: ***Why not a validity period for each property?***

Instead of grouping property values in Time Slices, another approach could be a temporal model where every property gets it own validity period.

The first argument against this approach is that, in general, the properties of a feature do not change independently from each other. There exist operational constraints that link the values of some properties with the values of other properties. Therefore, several properties would have anyhow to be grouped together, with a common validity period.

The second reason is that changes in the aeronautical world are regulated by the AIRAC cycle. This imposes that significant operational changes occur at predefined dates, in order to ensure the predictability of the aeronautical environment and to allow time for the users to accommodate with the changes. In general, aeronautical features have stable property values between AIRAC cycle dates. Therefore, grouping together the properties in Time Slice with a unique validity period is a simplified temporal model, which supports well the operational requirements.

2.3 (step 3) Temporary events – digital NOTAM

Aeronautical features may be affected by temporary events, such as a navaid being out of service, a runway being closed, a restricted area becoming active, etc. All such events generate temporary changes in the values of one or more feature properties. At the end of the temporary event, the values of these properties are reversed to their static values.

In order to model temporary events, we need to specialise the basic temporality model defined at step 2 by differentiating between two kinds of Time Slices:

- ***Baseline*** = a kind of Time Slice that describes the feature state (the set of all feature's properties) as result of a permanent change.
- ***Temporary*** = a kind of Time Slice that describes the transitory overlay of a feature state during a temporary event.

From a "payload" point of view, there exists an essential difference between Baseline and Temporary Time Slices:

- A Baseline Time Slice includes the values of all time varying feature properties that are defined for the time of validity of the Time Slice; for example, in the diagram below, TS2 will include the values of p1, p2, p4 and p5;
- A Temporary Time Slice includes just the values of the properties that are temporarily changed; for example, in the diagram below, TS "temp" will include just p4="value w". For this reason, temporary Time Slices are called "Temporary Delta" Time Slices.

Note: a temporary change could also consist in a feature property becoming temporarily undefined (no value). For this purpose, feature properties can also get a 'nil' value.

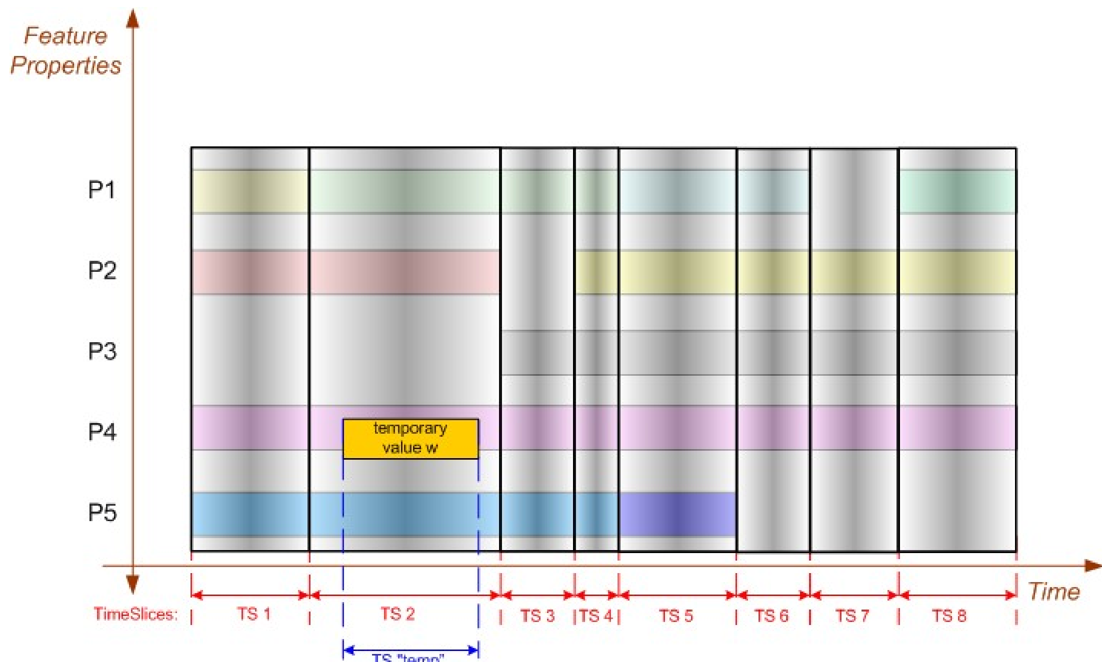


Figure 4 - BASELINE and TEMPDELTA TimeSlices

One reason for temporary Time Slices to contain strictly the modified properties is to avoid confusions that could result from overlapping temporary events. When several temporary delta overlap in time, complicated rules would be necessary in order to decide which values to include for not affected properties. Should the values of the baseline Time Slice be included? Or should the other temporary changes be considered? Therefore, the model is clearer if only the affected properties are included in Temporary Delta Time Slices.

With regard to the UML model, as the Temporary Delta Time Slices need to be distinguished from the baseline ones, an additional attribute is necessary in the AIXMFeatureTimeSlice class. This is named "interpretation" and indicates the type of Time Slice - BASELINE or TEMPDELTA, as shown in the figure below.

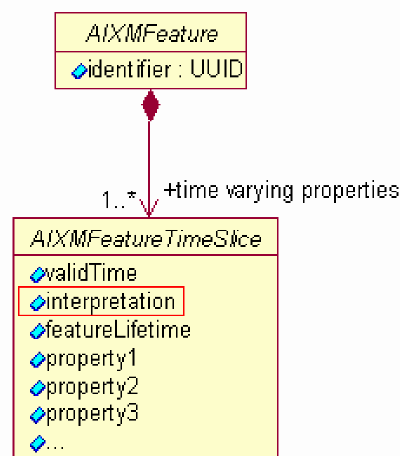


Figure 5 – FeatureTimeSlice with "interpretation" property

The essential benefit brought by TEMPDELTA Time Slices is that they enable the encoding of "digital NOTAM". A TEMPDELTA Time Slice will contain the values of all feature properties that are overlaying for a limited time period the baseline values.

Discussion: ***Why not considering the temporary change as a sequence of two permanent changes?***

Using a Time Slice model with BASELINEs only, the interval TS2 would have to be split into 3 new Time Slices, for example TS2a, TS2b and TS2c. In this approach, the temporary situation would be modeled as a sequence of two permanent changes. The disadvantage of this solution is that the information about the temporary nature of the value "w" would be lost. There exist aeronautical applications, such as charting and AIP production, which normally ignore the temporary changes. Such applications need to know if a value is temporary or part of the baseline.

Also, temporary events, such as the activation of a restricted area, have a life of their own: first the activation is requested, than planned for a time interval maybe different from what was requested, than active for maybe a shorter time than planned, etc. In order to correctly model the life of temporary events, they need to be modeled as such and not hidden behind fictitious permanent changes.

2.4 (step 4) Current Status - SNAPSHOT Time Slices

The temporality model described up to this point complies with the rules for completeness, minimalism, consistency and context-free mentioned at the end of section 1. Using BASELINE and TEMPDELTA Time Slices it is possible to describe the temporal evolution of the time varying properties of aeronautical features, covering both permanent states and temporary events.

However, the model is slightly inconvenient for a real life implementation, because it lacks the possibility to communicate the current status of a feature, which results when merging the baseline data with any temporary data that is effective at that moment in time. For convenience, we need an additional kind of Time Slice to be included in the model. This will be named "SNAPSHOT" and will carry the result of merging the effective BASELINE information with all overlaying TEMPDELTA that are effective at a that moment in time. Typically, a SNAPSHOT Time Slice will have a Time Instant as validTime.

- SNAPSHOT = A kind of Time Slice that describes the state of a feature at a time instant, as result of combining the actual BASELINE Time Slice effective at that time instant with all TEMPDELTA Time Slices that are effective at that time instant.

Note that for a SNAPSHOT, the correctionNumber and the sequenceNumber properties shall not be populated.

2.5 (step 5) Data exchange – need for PERMDELTA Time Slices

Another kind of Time Slice that will be introduced for convenience is in support to systems that need to notify the clients about data updates. There exist two types of applications:

1. "Pull" Systems - provide an interface by which a client can query the aeronautical information and extract the results of the query;
2. "Push" Systems - generate and transmit to the client notifications about aeronautical information changes.

For "push" systems, it is difficult to use only these three kinds of Time Slice for communicating (generating and transmitting) information about the future. For example, how to communicate information about the end of life (decommissioning) of a feature?

Using BASELINE Time Slices for this purpose would require communicating an 'update' of at least the last sent Time Slice, with an updated value of the 'endOfLife' property (encoded as featureLifetime/endPosition). This would also require interpretation rules such as "if the endOfLife

is equal with the end of validity of the Time Slice, then this means that the feature is permanently withdrawn". The postponement of a withdrawn, which is operationally possible, would require a second update of the endOfLife, which might become complicated to interpret.

A more convenient solution is to include in the temporality model a Time Slice that represents permanent change events. This will be called Permanent Delta (PERMDELTA).

- PERMDELTA = A kind of Time Slice that describes the difference in a feature state as result of a permanent change.

The ***end of life can now be communicated with a PERMDELTA Time Slice in which the featureLifetime/endPosition gets a value***,. Symmetrically, the start of life can also be communicated with a PERMDELTA Time Slice, in which the featureLifetime/startPosition property and the other feature properties get their initial values. Being modeled as formal events, the start of life and the end of life can relatively simply be postponed or advanced (this requires a mechanism for updating an 'event' Time Slice, which will be discussed later in this paper).

A second advantage of PERMDELTA Time Slices is that client systems no longer need to compare the previously received BASELINE Time Slice with the new one in order to detect the changed attributes. This process may be time consuming and even error prone. The data originator is the best positioned to know the list of changed properties and the PERMDELTA Time Slice gives the possibility to communicate this information to interested clients. This facilitates the implementation of systems that are not interested in changes of certain feature properties. For example, charting applications - a PERMDELTA affecting properties that do not appear on the chart will easily be ignored.

From a conceptual point of view, a PERMDELTA Time Slice occurs at the edge between any two consecutive BASELINE Time Slices and it contains values strictly for the changed properties.

All these kinds of Time Slices are described in Figure 6.

Conceptually, there exists a direct dependence between PERMDELTA and BASELINE Time Slices. However, this does not mean that the BASELINE Time Slice needs to be effectively instantiated after each PERMDELTA. In an implementation, it is possible, for example, to "accumulate" PERMDELTA Time Slices. The instantiation of a new BASELINE might occur, for example, after each third PERMDELTA affecting a feature.

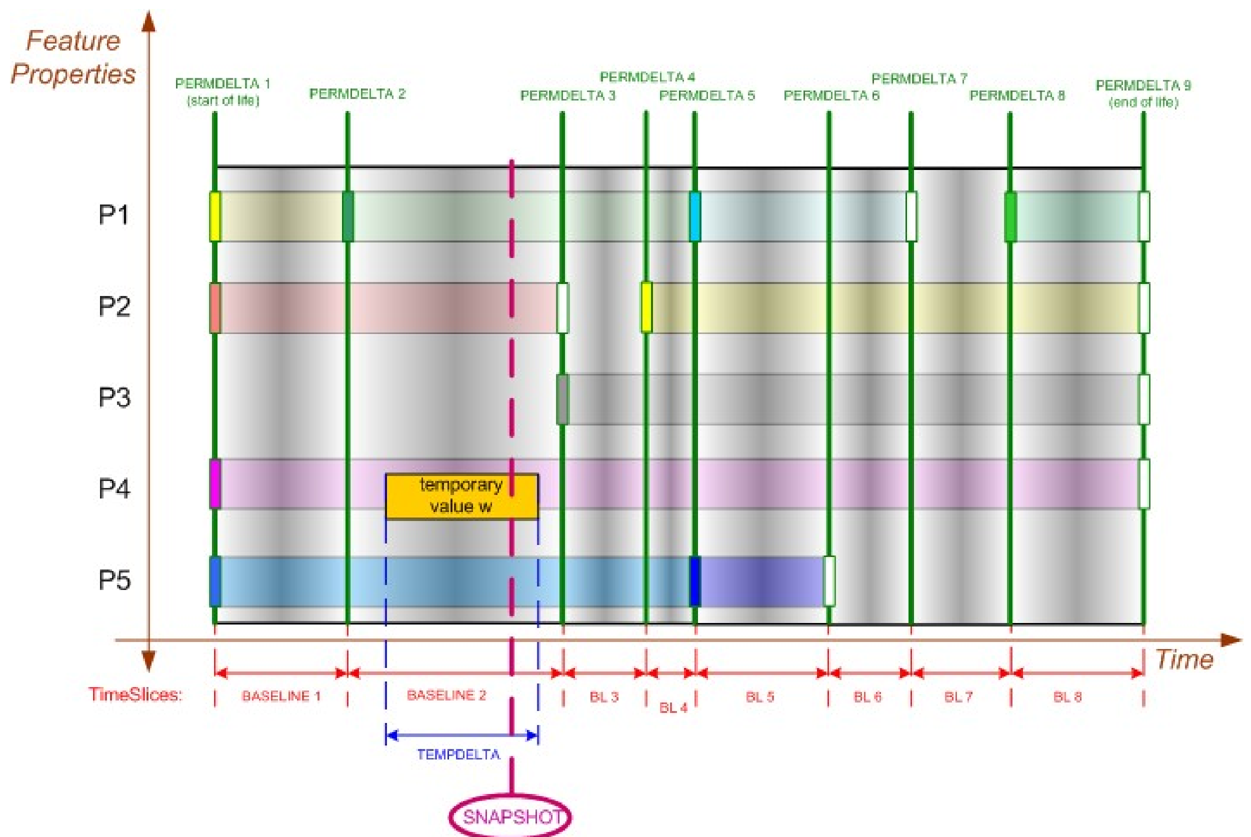


Figure 6 – Four types of TimeSlices

2.6 (step 6) Data exchange – corrections

In the aeronautical world we need to communicate information about events that are planned to take place in future. Inevitably, the reality might be different from the initial planning and it might be necessary to update the already communicated information.

As in AIXM the properties of a feature are encapsulated in Time Slices, this means that we need a mechanism for updating/correcting a previously communicated feature Time Slice. First, a key is necessary for the identification of the Time Slice concerned. For this purpose, a ***"sequence number" attribute is introduced in the model, playing the role of unique identifier for each Time Slice*** inside a feature.

If necessary to correct a previously communicated Time Slice, an update of the Time Slice will be provided, having the same sequence number but a higher correction number. As a consequence, if there exist more than one Time Slice with the same sequence number related to a given feature, the one with the higher correction number will be considered valid.

The UML representation of the final AIXM 5 Feature Time Slice model is provided below:

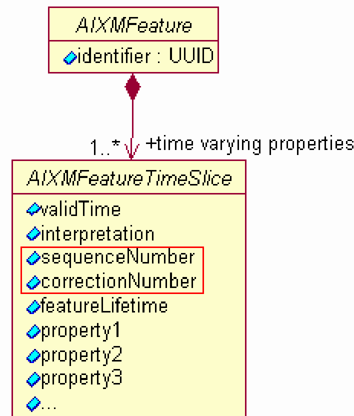


Figure 7 – Complete AIXMFeatureTimeSlice model

To summarise, the following Time Slice types are used in the AIXM:

- ***BASELINE = a kind of Time Slice that describes the feature state (the set of all feature's properties) as result of a permanent change.***
- ***PERMDELTA = A kind of Time Slice that describes the difference in a feature state as result of a permanent change.***
- ***TEMPDELTA = a kind of Time Slice that describes the overlay of a fe***ature state during a temporary event.
- ***SNAPSHOT*** = A kind of Time Slice that describes the state of a feature at a time instant, as result of combining the actual BASELINE Time Slice (valid at that time instant) with all eventual TEMPDELTA Time Slices that are effective at that time instant.

Discussion: ***What was the temporality model of past AIXM versions?***

AIXM 3.x and 4.x provide limited temporality support. It is possible to encode the feature state at a point in time (AIXM-Snapshot message) and to communicate baselines (AIXM-Update). AIXM 3.x and 4.x do not support the direct encoding of the temporary status information; it would have to be done as a sequence of two baselines, one changing the properties and the second one reverting to the previous situation. But this does not allow distinguishing between real permanent changes and temporary status information.

In addition, AIXM 3.x and 4.x embed temporality in the exchange message rather than in the feature itself. Consequently, temporality was a property of the message rather than a property of the aeronautical feature. The message properties describe how receiving systems should interpret the message content.

The limited capabilities to transmit temporal information using Update and Snapshot messages in AIXM 3.x and 4.x have led to the development of this more complete AIXM 5 Temporality Concept, at feature level.

2.7 Properties with schedule

The Temporality Model described up to this point works well for features that have properties with constant values during their time of validity. In some cases, one or more properties of a feature may have their own cyclic variation in time according to an established schedule. For example, a navaid can be operational during day time and unserviceable during night time; a restricted airspace could be active every day from 09:00 till 17:00; etc.

To model such situations, the concept of "properties with schedule" has been introduced in AIXM 5.1. The idea is to associate the properties that have cyclic varying values with a "Timesheet" that describes the times when each value is applicable for those attributes. The concept of Timetable/Timesheet already existed in AIXM 3.x and 4.x. It was inherited as such in AIXM 5.0, where it was found to sometimes overlap with the TimeSlice concept. Therefore, a re-thinking of the role of Timesheets was necessary in AIXM 5.1 (see Change Proposal 5.1-35, which provides a more detailed analysis of the need for schedules in AIXM).

Discussion: ***Does the model really need to support schedules?***

It is obvious that schedules exist in the aeronautical data domain. The question is whether the TimeSlice concept is sufficient or not for also coverings such situations.

Theoretically, the TimeSlice model without schedules can be used for a feature (such as a navaid) that has a property (such as operational hours) that changes cyclically (such as operational every day from 06:00 – 22:00). But this means that either a dedicated BASELINE or a TEMPDELTA is encoded each time when the operationalStatus changes from operational to unserviceable. This would generate either 730 BASELINE TimeSlices or one BASELINE Timeslice and 365 TEMPDELTA TimeSlices in a year, which is a significant inconvenience. In addition, the "cyclical" aspect would not be immediately visible.

Therefore, the TimeSlice model needs to be complemented with a "schedules" concept, which enables to model directly the cyclic variation of the values of one or more feature properties.

At the feature level, all the properties that change according to an established schedule must be isolated in a separate class, as illustrated below with class NavaidOperationalStatus. This class inherits from an abstract class called "PropertiesWithSchedule".

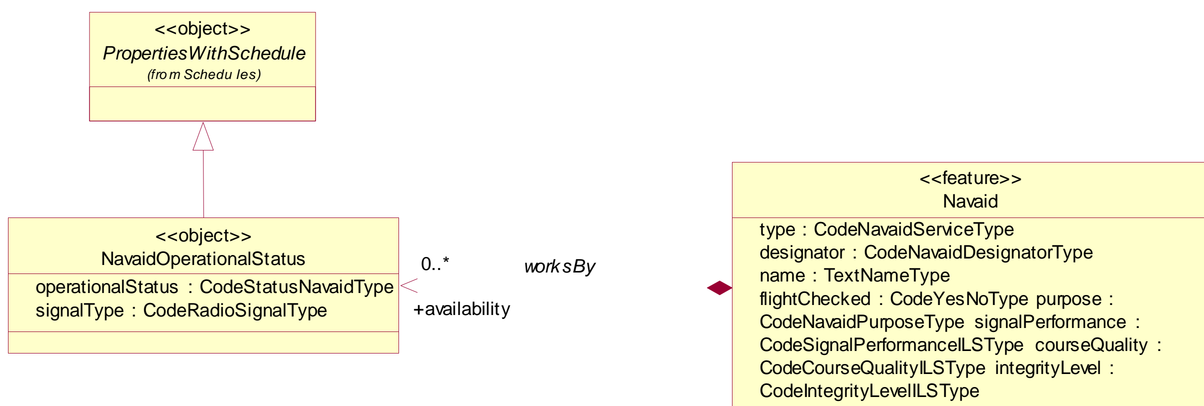


Figure 8 - Properties with schedule model

The abstract class PropertiesWithSchedule is then associated with Timesheet(s).

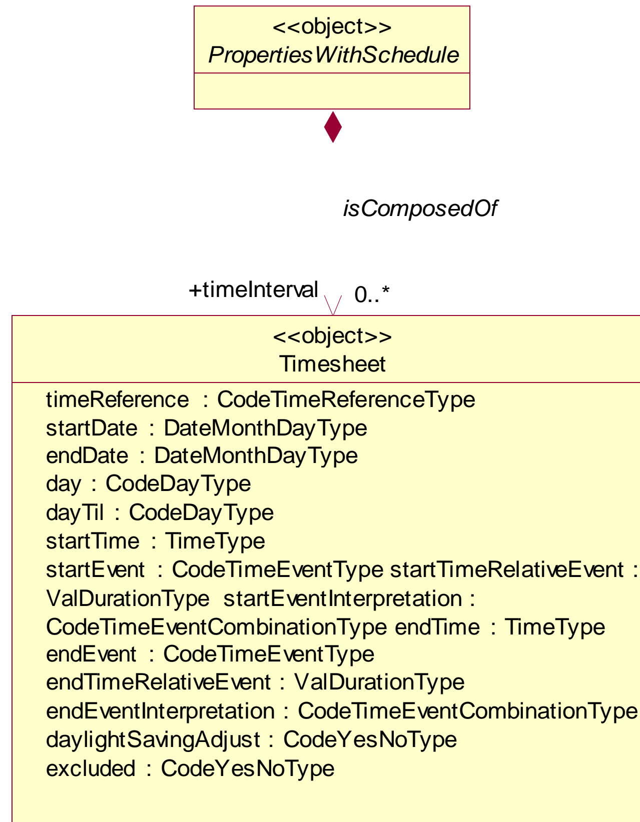


Figure 9 - Timesheet class

The Timesheet class contains the time reference system (UTC-12 to UTC +14), daylight saving indicator and provides the possibility to include/exclude specified dates and times. It can, for instance, represent:

- a single repetitive time period, such as "each Monday from 10:00 to 16:00";
- a single time block spreading over several days, such as "From each Monday 10:00 till Thursday at sunset"
- a date range, such as "every year from 15 OCT to 15 MAY";
- etc…

Discussion: ***Is there any alternative to introducing the "properties with schedule" concept?***

Another solution could be to include "schedules" in the TimeSlice concept and make a schedule usable for any feature. That would have two disadvantages.

If an attribute, such as the value of a declared distance, has one value during day and another value during night, each of the two values would need to be part of a different Baseline. Each of the two Baseline would have a schedule that would indicate when they are applicable. But the two Baseline would have overlapping validity times. This would significantly complicate the Temporality concept of AIXM. The analysis also shows that, frequently, schedules really concern just one or two attributes. Having the schedule at the level of the feature would hide this important aspect.

Therefore, the introduction of the attribute with schedule concept is considered the most convenient approach.

The introduction of the PropertiesWithSchedule requires clear rules for interpreting the various combinations that can occur between TimeSlice types, their validity and property schedules. The risk is that the value of a property may be undefined if the schedules associated with a BASELINE leaves "holes". In the AIS operations of today, for properties that change values according to a schedule, it is quite common to specify only the "main" value, such as "operational", "active", etc.. For example, it is indicated that the "navaid operates every day from 06:00 – 22:00", but it is not explicitly indicated what is the status of the navaid between 22:00 – 06:00. Operational people will assume that the navaid is not operating between 22:00 – 06:00.

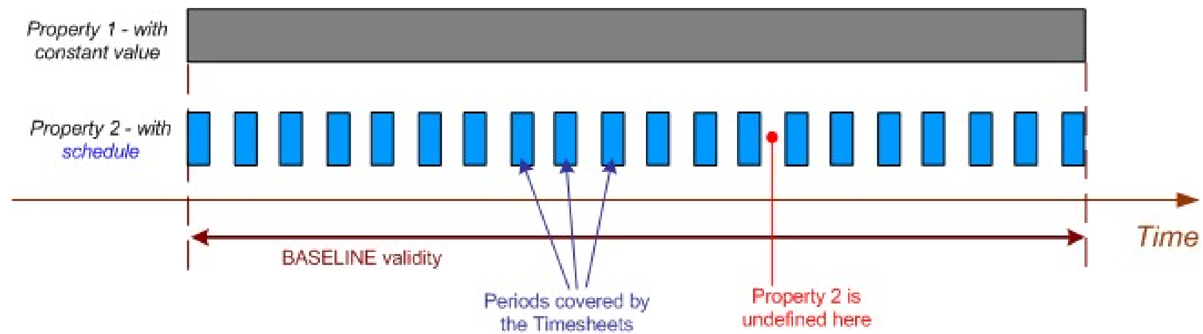


Figure 10 - Schedule with holes

As machines cannot make "assumptions", for digital data processing, it is safer if the "non-operating" times are also stated explicitly, so that schedules associated with the values of the property do not leave any holes. Therefore, it is recommended that BASELINE Timeslices contain only fully defined properties with schedule, which indicate explicitly what the property value is at every moment within the validity time of the TimeSlice. If one or more Timesheets are associated with a property with schedule, than the value of the property shall be considered as undefined at any moment not covered by a Timesheet.

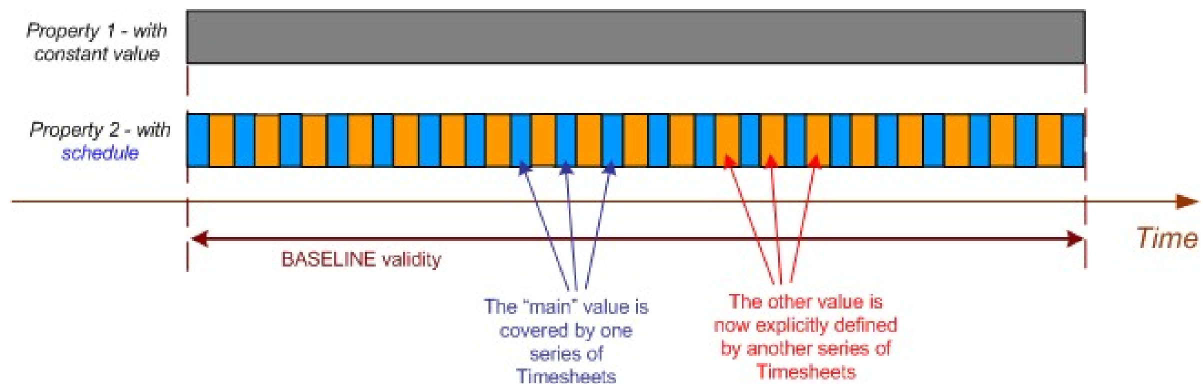


Figure 11 - Fully defined property with schedule

Timesheets that leave gaps can also occur for TEMPDELTA TimeSlices. By definition, any property contained in a TEMPDELTA overrides the value of the equivalent BASELINE property, for the duration of validity of the TEMPDELTA. Therefore, as a general principle, the times encoded in Timesheets contained in TEMPDELTA TimeSlices also replace in full the times encoded in the equivalent BASELINE Timesheets.

The situation is simple and clear if the TEMPDELTA does not have Timesheets or if these Timesheets cover the whole period of applicability of the TEMPDELTA, as presented in the following diagrams:

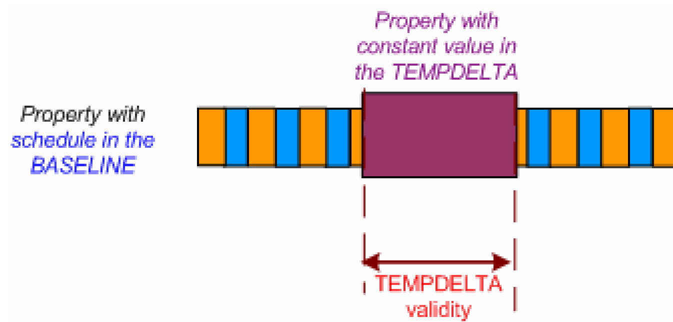


Figure 12 - TEMPDELTA constant value overrides the BASELINE schedule

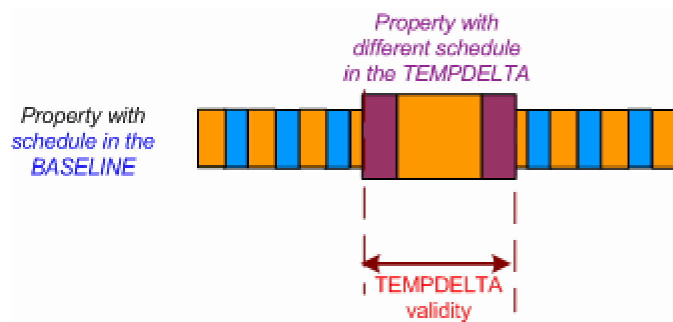


Figure 13 - TEMPDELTA schedule overrides the BASELINE schedule

A situation that can lead to difficulties in interpretation is when the Timesheets associated with the TEMPDELTA leave gaps (times where the value of the property is not explicitly specified), as in the following diagram:

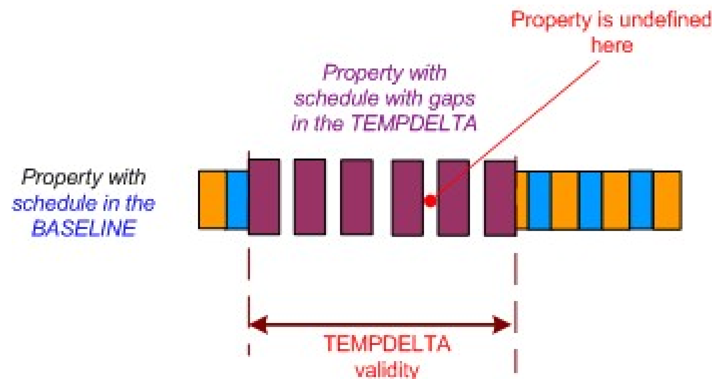


Figure 14 - TEMPDELTA schedule gaps leave the property undefined

The temptation could be to consider that the BASELINE situation applies in the gaps left by the Timesheets associated with the TEMPDELTA. But this would be in conflict with the general principle that the TEMPDELTA values replace the BASELINE values in full. Therefore, if a TEMPDELTA schedule leaves gaps (periods for which the value is not explicitly provided), then it shall be considered that the property has an unspecified value at those time periods.

From the examples above, it is therefore recommended that TEMPDELTA schedules do not leave unspecified periods (gaps) within the time of applicability of the TEMPDELTA.

2.8 Temporality applied to the Abstract Model

The AIXM UML model contains a set of abstract classes that are used as templates for the features and objects defined in AIXM. When applying the Time Slice concept, as described in this document, this would trigger the split of every UML class that represents a feature into a main class and a “FeatureTimeSlice” class, as shown in the following diagram.

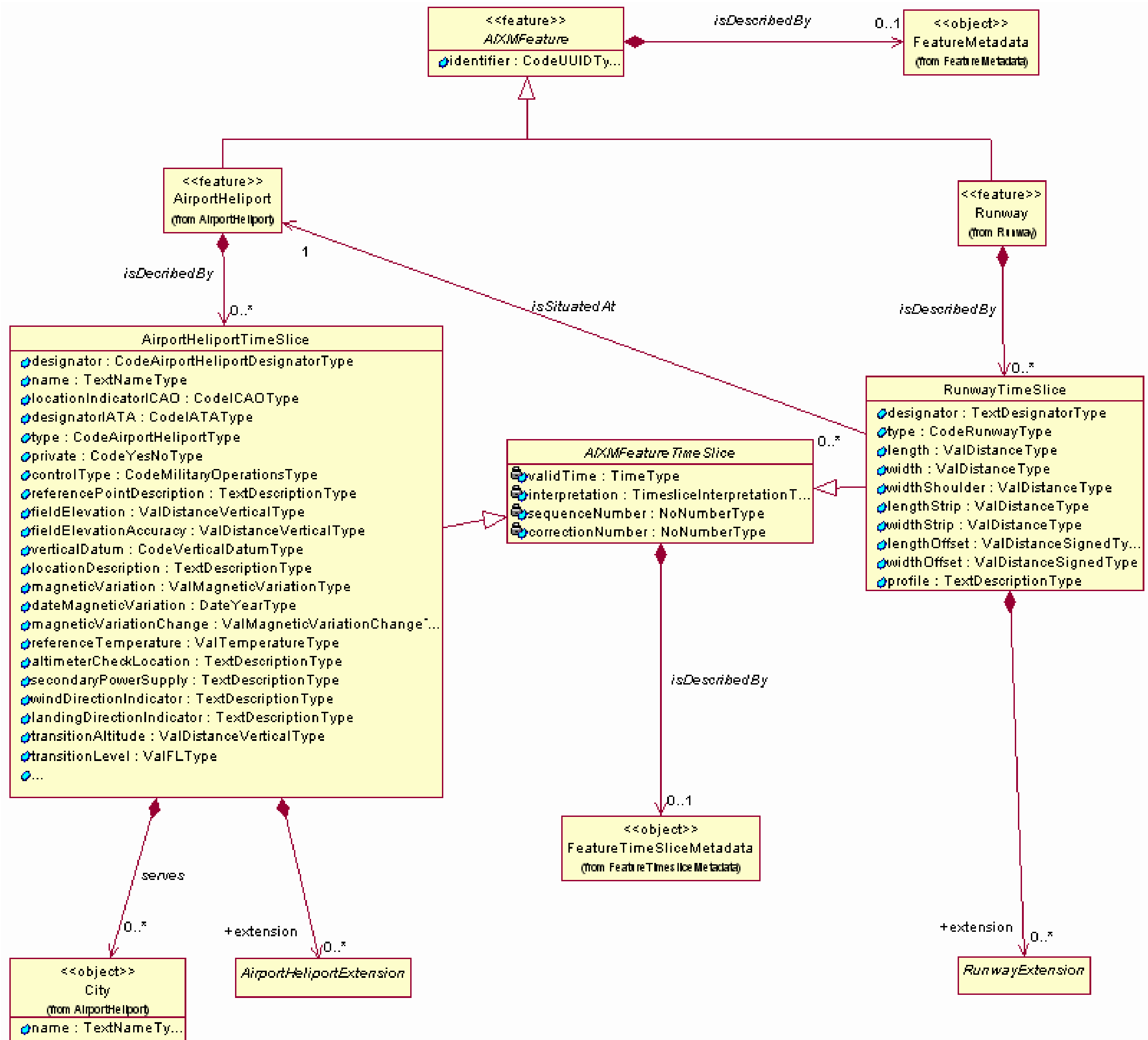


Figure 15 - Model expanded with explicit TimeSlice classes

The UML diagram shows how each and every `<<feature>>` inherits from the abstract AIXMFeature class. The concrete features are described by TimeSlices which have properties. The TimeSlice inherits from the abstract AIXMFeatureTimeSlice class.

The diagram also shows that each AIXM Feature may have FeatureMetadata and each TimeSlice may have FeatureTimeSliceMetadata. Finally, each TimeSlice may contain an Extension. The Extension mechanism allows each user of AIXM 5 to define and use his own specific attributes and classes, in addition to the core AIXM ones.

The diagram above is quite complex. If applied to the whole set of AIXM classes, it might undermine the readability of the UML diagrams, as a separate "TimeSlice" class and the necessary associations would have to be added for each <<feature>> class. **Therefore, the Design Team has decided to provide a simplified AIXM UML model, without visible inheritance of all features from the abstract AIXMFeature and without visible *SomeFeature*TimeSlice classes**. However, the split and into SomeFeatureTimeSlice classes is assumed to exist, when converting from the UML model to the XML Schema of AIXM.

3. Application aspects

3.1 BASELINE Time Slices with undetermined end of validity

The operationally significant changes in the aeronautical information domain are regulated by the AIRAC cycle. Usually, when a permanent change is communicated, it is unknown when the next permanent change will take place. Therefore, it triggers the encoding of a BASELINE with an unknown end of validity. This is expressed in GML as “<gml:endPosition indeterminatePosition="unknown"/>”. This BASELINE will cover the period until the next permanent change. Implicitly, when the next change occurs, the previous BASELINE gets an end of validity and needs to be updated/corrected.

The situation may be represented as in the diagram below. The first BASELINE, created at the start of life of the feature, initially has an unknown end of validity. It is represented on this diagram as “BASELINE 1”, assuming that it has sequenceNumber=1.

When the permanent change “PERMDELTA 2” occurs, the validity of the initial BASELINE ends and a new BASLINE takes over. To completely represent the history of the feature, a corrected version of the first BASELINE is instantiated (having the same sequenceNumber=1 and also a correctionNumber=1), this time with a known end of validity. The newly created BASELINE has sequenceNumber=2 and no correction yet.

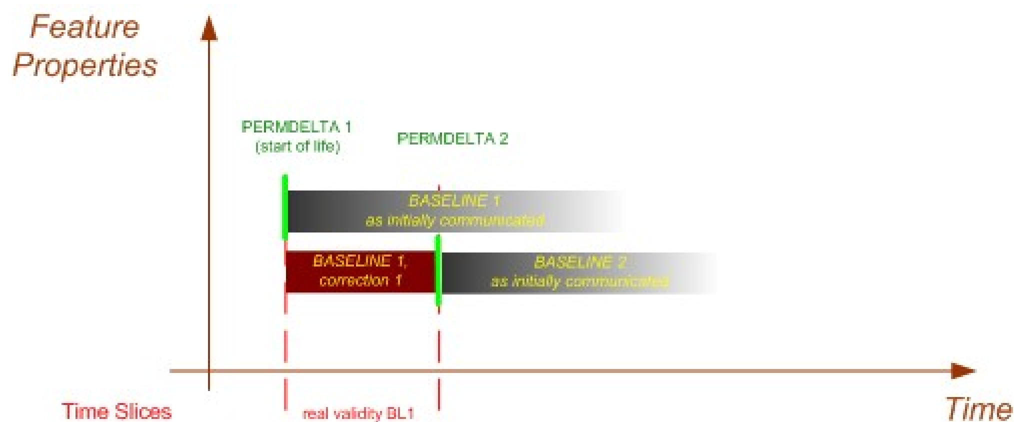


Figure 16 – Previous BASELINE correction as result of a PERMDELTA

3.2 SequenceNumber values

As explained in 2.6, the sequenceNumber is primarily used as an identifier of the TimeSlice, in order to apply a correction. Therefore, the sequenceNumber shall be unique per type of TimeSlice and should be persistent. It is not allowed to change the sequenceNumber of a TimeSlice because this could break the link with a correction TimeSlice and there is no mechanism in AIXM that would enable notifying the change of a sequenceNumber.

A secondary aspect is that sequenceNumbers can also be used to give some chronological information about the time when that TimeSlice was issued (not in which order it becomes valid!).

Therefore, it is recommended that sequenceNumbers are allocated starting from “1” and are incremented by 1 unit (“2”, “3”, “4”, etc.) each time that a new TimeSlice of that kind is encoded:

- The initial PERMDELTA that creates the feature to have sequenceNumber=1 and the first BASELINE that results to also have sequenceNumber=1;
- The second PERMDELTA (the first change of the feature after its creation) to have sequenceNumber=2 and the resulting BASELINE to also have sequenceNumber=2, etc.
- Then, the first TEMPDELTA that occurs to have sequenceNumber=1, the next one sequenceNumber=2, etc.

The result of this recommendation is visible in Figure 17 – Complete history of a feature.

3.3 Feature end of life

As explained in 2.5, the PERMDELTA TimeSlices have been introduced in order to facilitate the notification of the end of life / decommissioning / withdraw of a feature. This shall be encoded as a PERMDELTA that changes the featureLifetime/../endPosition property (of the BASELINE TimeSlice valid at the withdraw time) from "undetermined" into a precise date and time value. The effective date of the PERMDELTA shall be equal to the end of life value. No other property of the feature is included in the PERMDELTA in this case, as this PERMDELTA will not result into the establishment of a new BASELINE, just in a correction of the last active BASELINE.Extended to the complete history of the feature, the correction of the initially communicated BASELINES up to the end of life of the feature can be represented as in the following diagram.

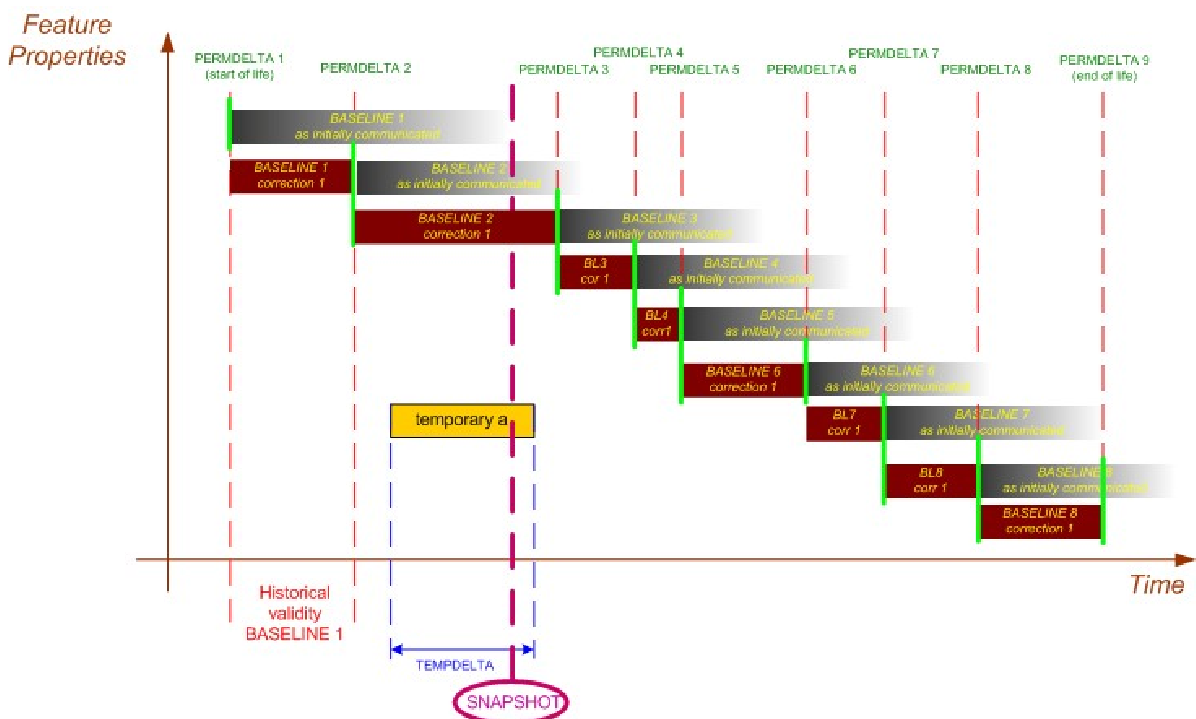


Figure 17 – Complete history of a feature

3.4 "Delta" for complex properties

Many AIXM features have complex properties that are made of zero or more component classes (represented as aggregated classes in the UML model, 0..*). For example, an AirportHeliport has an associated AirportHeliportAvailability which is "composedOf" zero or more AirportHeliportUsage.

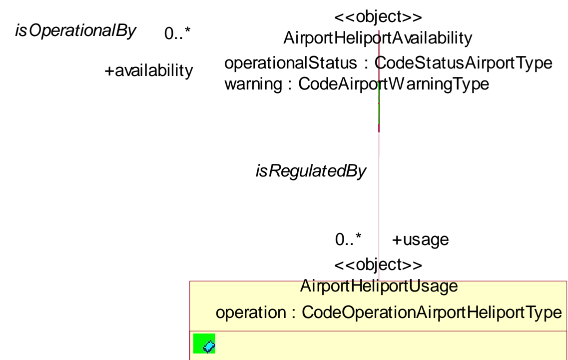


Figure 18

The question is: what should PERMDELTA or TEMPDELTA Time Slices contain for such situations?

By definition, "delta" Time Slices shall contain strictly the values of the affected feature properties and this rule applies only to features. Objects are consider complex types of a feature property and have to be included in full in a "delta" Time Slice, if the encapsulating feature property is changed. This will be explained further down with an example.

Feature properties are all the feature attributes and all the associations for which the feature has the navigability (indicated as an arrow pointing from the feature class towards another class). For example, in the previous class diagram, the properties of the AirportHeliport feature are all attributes (designator, name, …, certificationExpirationDate) and also the "availability" property, given by the role played by class AirportHeliportAvailability in the association isOperationalBy. The "availability" property of the Airspace is a complex one, composed of several AirportHeliportUsage. If a temporary or permanent change occurs inside the AirportHeliportAvailability (for example, a modification of one of its composing AirportHeliportUsage), then the modified AirportHeliportAvailability shall be included in full in the TEMPDELTA or PERMDELTA Time Slice.

3.5 "Delta" for multi-occurring properties

An equivalent rule applies for feature properties that can occur multiple times. In UML, such properties are encapsulated in an Object, which is related with the feature class by a 0..* association. For example, an AirportHeliport may serve 0..* Cities, as indicated in the following diagram. This means that the property "serves" of the AirportHeliport feature is potentially multi-occurring.

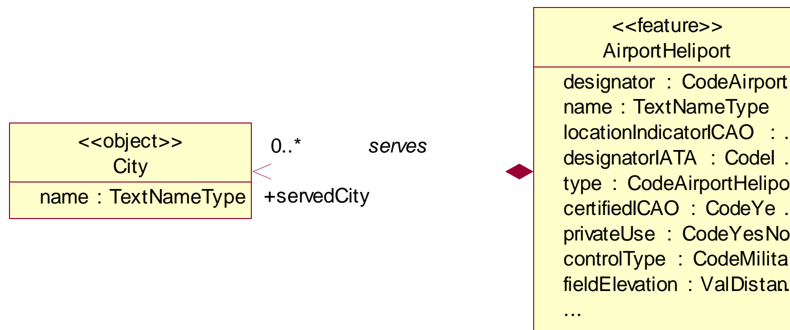


Figure 19

The rule is that, in a PERMDELTA or TEMPDELTA Time Slice, multi-occurring properties shall be provided with all occurrences included. Therefore, if an AirportHeliport had, for example, two served cities and this needs to be permanently changed to three cities, all the three "servedCity" properties have to be included in a PERMDELTA.

3.6 Identifying the feature affected by a DELTA Time Slice

A Time Slice is always encoded as child element of a feature. As every AIXM feature has 'gml:identifier' property, this should be sufficient for this purpose. This is supposed to be a global unique identifier (of type UUID), which provides an unambiguous key for every AIXM Feature.

However, the global unique identifiers are likely to not exist for some time. In this situation, there are two possibilities:

- Either use the gml:identifier property for encoding a local unique identifier (an artificial key), specific to the data originator. In this case, the PERMDELTA and TEMPDELTA Time Slices can be operationally received only from the same originator who has provided the BASELINE data. Using PERMDELTA/TEMPDELTA from another data source would inevitably break the chain, as different identifiers would be used.
- Or, in addition to the PERMDELTA or TEMPDELTA Time Slice, include in the AIXM file a SNAPSHOT Time Slice, which contains some properties (a "natural key") that are sufficient for identifying the feature. The fact that the SNAPSHOT contains just some natural key properties and does not contain all properties is not in conflict with the definition of a SNAPSHOT TimeSlice, because a SNAPSHOT represents the view that a particular system has on that feature, which might be an incomplete view. The recipient of the data will have to query his local system and identify the feature that has the same values at that moment in time, thus being identified as the target for the update.

3.7 Canceling a Time Slice (abandoned changes)

For aeronautical information systems that work in "push" mode, the primary means for generating and providing information about a change is by PERMDELTA and TEMPDELTA Time Slices. The question is what procedures shall be applied in the case of a change in the planning, such as:

- Abandoning the commissioning/decommissioning of a feature (before its effective date)
- Abandoning a permanent change (before its effective date)
- Abandoning a temporary change (before its effective date)

It was already discussed (see 2.6) that the postponement/advancement of an event requires a correction to a TimeSlice, using the sequenceNumber property as key for identifying the Time Slice concerned. The sequenceNumber will also be used for identifying the PERMDELTA or TEMPDELTA that needs to be abandoned. To clearly indicate that the change contained in the TimeSlice has been canceled, the gml:validTime property will be empty and it will have the nilReason attribute set to "inapplicable". For example, if a PERMDELTA for SomeFeature has been provided, with sequenceNumber "23", in order to cancel it, a second PERMDELTA with the same sequenceNumber and a higher correctionNumber has to be issued, as below

TimeSlice (initial)

- validTime = timeInstant…
- *interpretation = PERMDELTA*
- sequenceNumber = 23
- featureLifetime/beginPosition = same timeInstant…
- property 1
- property 2
- property 3
- property 4
- 5

TimeSlice (correction)

- validTime : nilReason="inapplicable
- *interpretation = PERMDELTA*
- sequenceNumber = 23
- correctionNumber = 1
- featureLifetime/beginPosition: nilReason="inapplicable

Note that this Time Slice cancellation does not affect 'pull' systems, such as Web Services or WFS, where the system provides the most current information, following an on-line request by the client. The client is not supposed to refer to or compare the results with the results of a previous query.

3.8 Overlapping TimeSlices and corrections

The sequenceNumber and correctionNumber are used to resolve and interpret overlapping timeSlices. Consider the scenario shown in the figure below where a Feature's Status property is changed repeatedly over several overlapping time intervals. Each temporary change receives a sequenceNumber. In the example, one of the Time Slices is corrected, leading to a duplicated sequenceNumber and different correctionNumber.

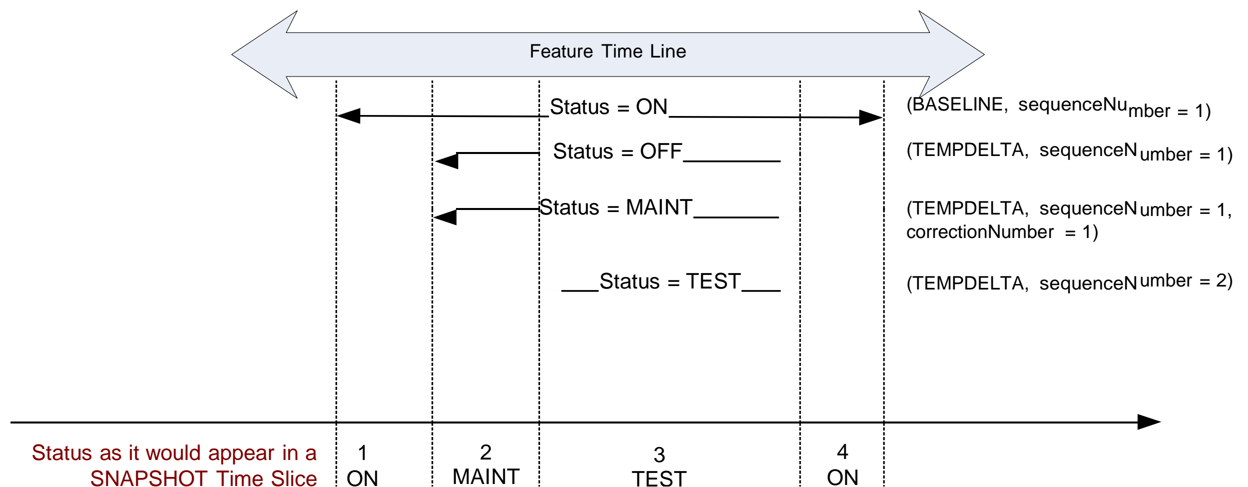


Figure 20 Example of TimeSlice corrections and overlapping

At the edges of each temporal event we can identify transitions to different feature versions. The combination of Time Slice type and sequenceNumber can be used to unambiguously identify the value of the Feature's Status property at each moment in time.

To determine the value of a property at a given time or over a given time interval the following rules should be used:

1. Identify the BASELINE that is valid at that moment in time, by looking at it's validTime. If several exist, they should all have the same sequenceNumber and different correctionNumbers. Take the one with the highest correctionNumber;
2. Identify all TEMPDELTAS that are effective at the specified time ;
3. Sort the TEMPDELTAs by increasing sequenceNumber;
4. Apply the TEMPDELTAs to the feature from low sequenceNumber to high sequenceNumber.
    a. When two or more deltas have the same sequenceNumber apply the delta with the highest correctionNumber.

The possibility to resolve overlapping TEMPDELTAs using the sequenceNumber and correctionNumber shows how cancellations and corrections can be communicated. In this example, TEMPDELTA sequenceNumber = 1 is initially used to communicate that the feature Status = OFF. Later, a Time Slice correction is transmitted using the same sequenceNumber = 1 but with a correctionNumber = 1; it corrects the feature state to Status = MAINT. However, the final status is later given by the TEMPDELTA with sequence number 2 which indicates the feature Status = TEST.

3.9 Other implementation considerations

The conceptual temporal model described in the previous section provides considerable flexibility for systems that implement temporality. A system that tried to fully implement the AIXM temporality model would be very complex. However there is no requirement for systems implementing AIXM to support all kinds of Time Slices. For example:

- Some systems may only store BASELINE Time Slice data and disregard any temporary changes. Examples include AIP publication, paper chart publishers and ARINC 424 based systems.
- Some systems may only transmit and store temporary changes. Examples include the NOTAM systems. However, such systems need to refer to source of BASELINE data.
- Some systems may only require periodic snapshots providing the current state of the system. An example is a passive monitoring system designed to report system status at selected time intervals.
- Some systems may want a new “snapshot” after every change without making a distinction between a temporary and a permanent change. Examples include traffic management and flight plan processing systems.
- Some systems may be developed that can process and interpret all of the temporal components and provide users with Baseline, Deltas and Snapshot Time Slices at any given moment in time.

AIXM contains a complete temporal model; however, as the examples illustrate it is the responsibility of interacting systems to negotiate specific temporal data exchange requirements as well as to integrate temporality into their internal subsystems.

4. Usage examples

4.1 Navaid example

Figure 21 illustrates the temporal model by showing a transmission frequency change for a navigation aid (VOR AML, from 112.0 MHz to 113.2 MHz). Normally, this change should occur at an AIRAC cycle date. Usually, the change requires the navaid to be out of service for a certain time, then to be on test on the new frequency. The temporary status is communicated at present through NOTAM messages.

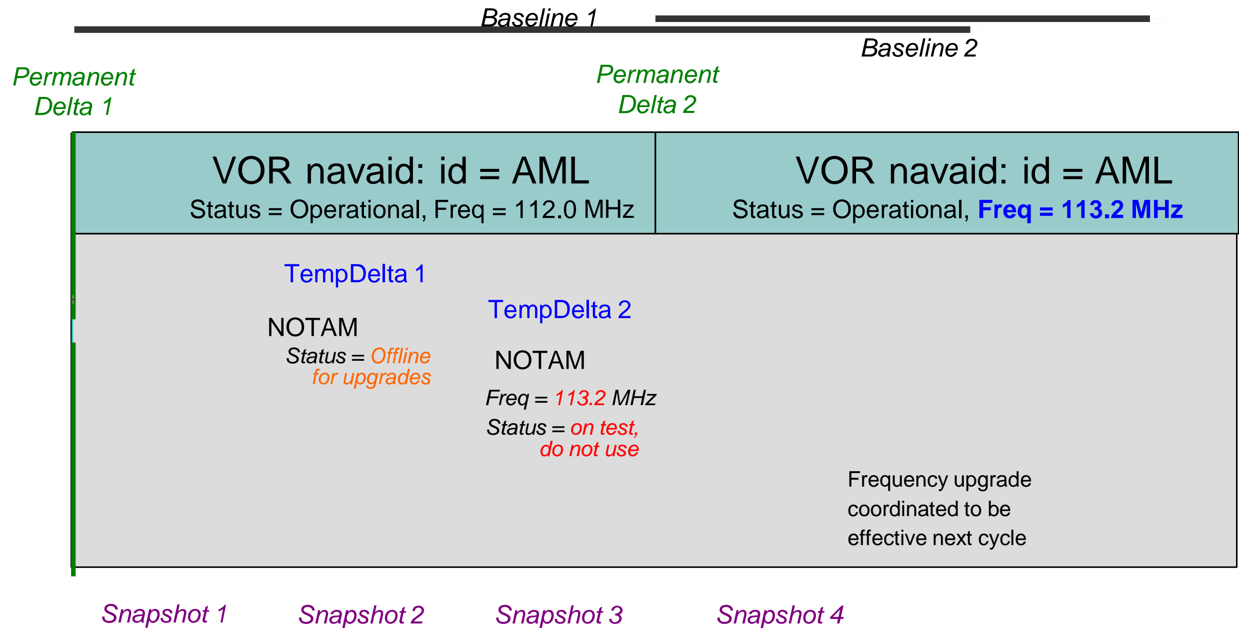


Figure 21

Based on this diagram we can identify the following temporal components:

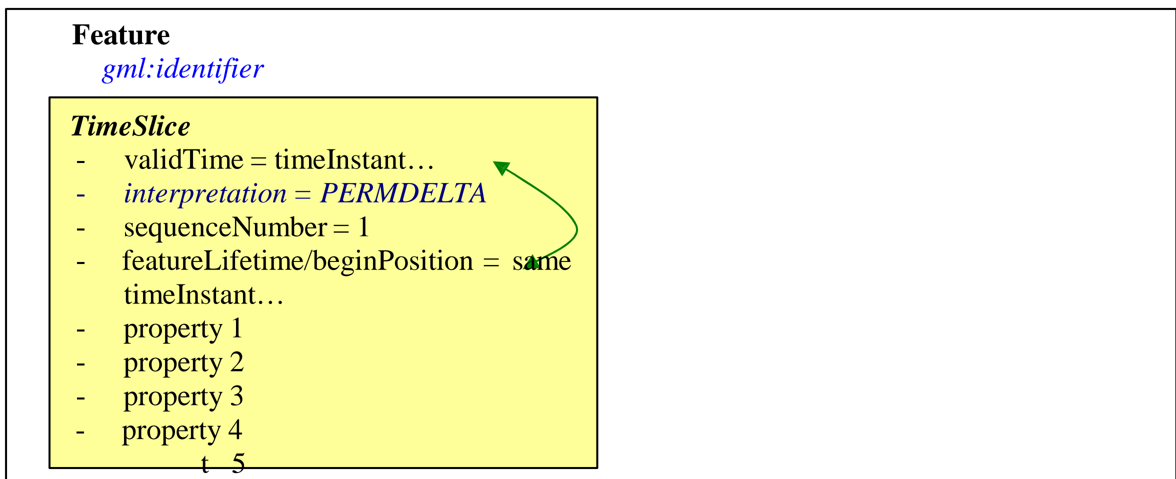
- The diagram shows two BASELINE Time Slices. The first baseline has a NAVAID frequency of 112.0 MHz and is valid since some time in the past; the second baseline has the new frequency of 113.2 MHz and is valid starting from the AIRAC cycle date.
- A PERMDELTA can be used to describe the permanent state change, which is the AML VOR frequency change. For completeness sake, the previous PERMDELTA that has preceded the first BASELINE (1) is also shown.
- Each transitory event can be expressed as a TEMPDELTA that changes the Operational Status of the navaid and eventually the frequency.
- Based on the PERMDELTA and the TEMPDELTA delta Time Slices shown in the diagram, four different versions for the “current status of the feature” may exist. Each “current status” version begins and ends at the boundary of a Permanent or Temporary Delta and may be presented as a Time Slice of type SNAPSHOT.

Depending on the temporal implementation employed by the exchanging systems, different methods can be used to communicate feature changes. In the interest of global standardization, the rest of this

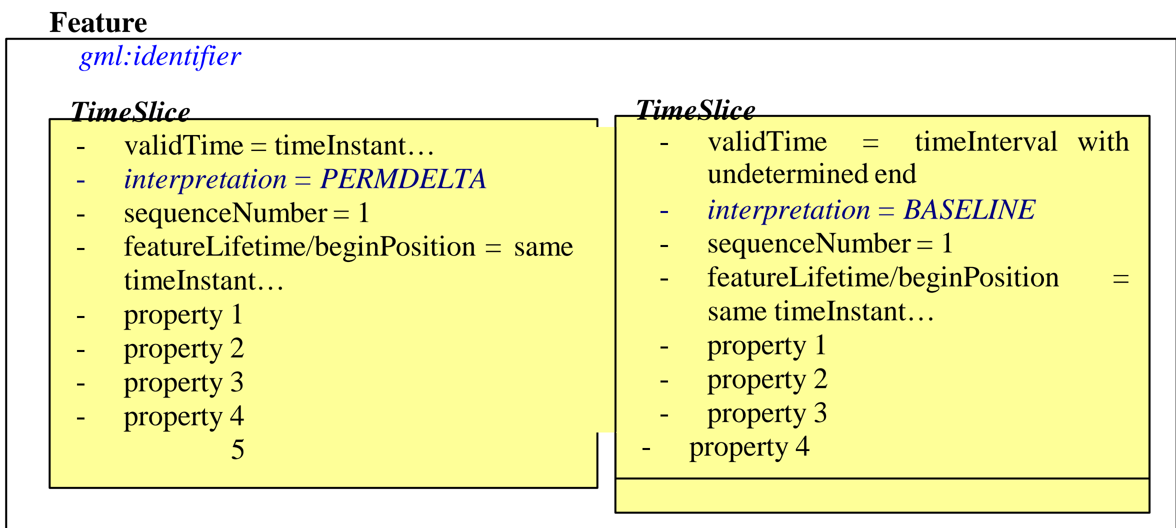
section provides some recommendations. They are relevant especially for “push” type applications, which generate and provide notifications in the form of TEMPDELTA and PERMDELTA Time Slices.

4.2 Feature creation (commissioning)

The start of life of a feature (also known as “commissioning”) is modelled as a PERMDELTA which gives an initial value to the startOfLife property and to all other feature properties that are defined. The validTime of the PERMDELTA shall be the effective date and time when the feature is commissioned.



Optionally, if this has been requested by the user, a BASELINE Time Slice, containing the same property values as the PERMDELTA (as they are the result of the PERMDELTA) may also be included. The validTime of the BASELINE shall be a timeInterval with the end “undetermined”.



4.3 Permanent change

A permanent change is modeled as a PERMDELTA Time Slice, containing all properties that change their values. The validTime of the PERMDELTA shall be the effective date and time of the change.

Optionally, a BASELINE Time Slice can be included, containing all the properties that have a value as they result after the PERMDELTA. The validTime of the new BASELINE shall be a timeInterval with the end "undetermined".

Feature

gml:identifier

TimeSlice
- validTime = timeInstant…
- *interpretation = PERMDELTA*
- sequenceNumber = 2
- property 3 (new value)
- property 5 (new value)

TimeSlice
- validTime = timeInterval with undetermined end …
- *interpretation = BASELINE*
- sequenceNumber = 2
- featureLifetime/beginPosition = timeInstant…
- property 1
- property 2
- property 3 (new value)
- property 4

4.4 Digital NOTAM

A temporary state of a feature is encoded as a TEMPDELTA Time Slice, containing all properties that temporarily change their values. The validTime of the PERMDELTA shall indicate the start and the end of effectivity for the temporary state. The end can be undetermined.

Feature

gml:identifier

TimeSlice
- validTime = timeInterval…
- *interpretation = TEMPDELTA*
- sequenceNumber = 1
- property 4 (temporary value)

Optionally, a SNAPSHOT Time Slice can be included in the data set (used as "natural key").

Feature
gml:identifier

TimeSlice
- validTime = timeInterval…
- *interpretation = TEMPDELTA*
- sequenceNumber = 1
- property 4 (temporary value)

TimeSlice
- validTime = timeInstance
- *interpretation = SNAPSHOT*
- property 1 (part of natural key)
- property 2 (part of natural key)

4.5 End of life (decommissioning)

The end of life of a feature (also known as "permanent withdrawn" or "decommissioning") is modelled as a PERMDELTA which gives a value to the featureLifetime/endPosition.

Feature
gml:identifier

TimeSlice
- validTime = timeInstant…
- *interpretation = PERMDELTA*
- sequenceNumber = 3
- featureLifetime/endPosition = same timeInstant…

Optionally, the correction of the latest BASELINE can be included (if requested by the client).

Feature
gml:identifier

TimeSlice
- validTime = timeInstant…
- *interpretation = PERMDELTA*
- sequenceNumber = 3
- featureLifetime/endPosition = same timeInstant…

TimeSlice
- validTime = timeInterval with the end as specified by the PERMDELTA
- *interpretation = BASELINE*
- sequenceNumber = 2
- correctionNumber = 1
- featureLifetime/beginPosition = timeInstant…
- featureLifetime/endPosition = timeInstant, as specified by the PERMDELTA
- property 1
- property 2
- property 3
- property 4
- property 5

4.6 Complete feature histories

The Timeslice model can be used to transmit history of a feature by transmitting the sequence of changes that occur to the feature's property. The feature history can be the past history or future history.

Figure 22 shows an example history of a fictional VOR navigation aid. The navigation aid has the following events:

- Jan 7, 2006: Commissioned
- Jan 23 – Feb 18, 2006: Temporary frequency change
- Feb 11 – Mar 9, 2006: Temporary offline
- Feb 22, 2006: Change in magnetic variation
- Mar 27, 2006: Change in frequency

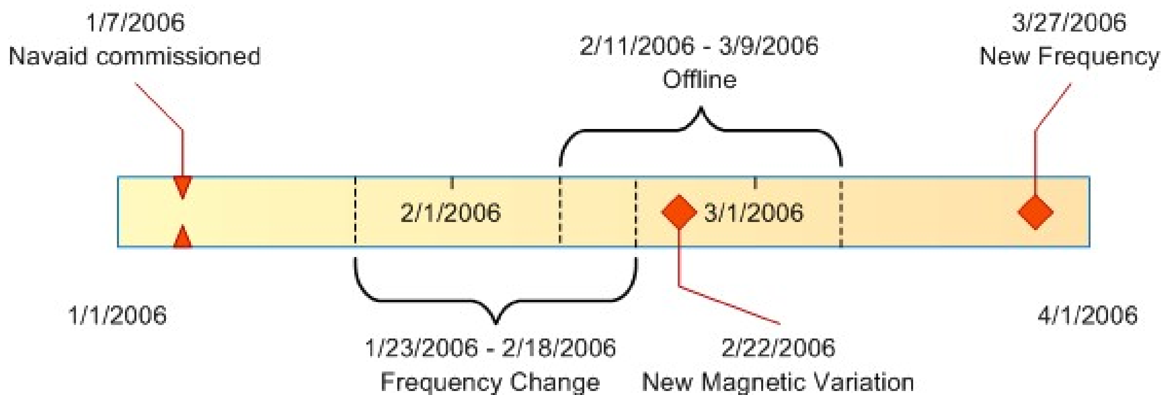


Figure 22: Fictitious example - history of a VOR navigation aid.

Using the Time Slice model we could represent the history of the VOR navigation aid as a series of five Time Slices, as shown in Figure 23. Three Time Slices are used to represent states and two are used to represent temporary events. Notice that overlapping events are encoded as separate Time Slices. The PERMDELTA Time Slices are not shown in this example.

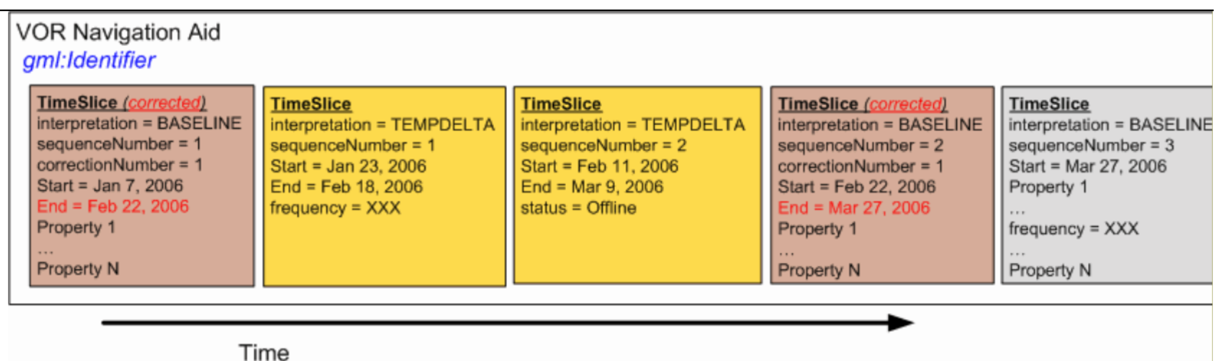


Figure 23: TimeSlices for the VOR navigation aid history

This approach to modeling history is equivalent to the recommended approach for GML 3.2 [4]. In actual implementations of the AIXM Timeslice mode, communicating histories can lead to very large messages. These large messages might be a problem for some resource constrained system. Although implementation issues are outside the scope of this design document we want to point out that the disadvantage of message size should be weighed against the value of standardization and compliance with GML. In most situations, the value of standardization may outweigh the loss of message efficiency.

References

1. Aeronautical Information Exchange Model (AIXM), Exchange Model goals, requirements and design, December 2006, www.aixm.aero
2. Aeronautical Information Conceptual Model, Edition 1.0, Ref. AIS.ET2.ST01.2000-02, 01 October 1997 (Eurocontrol Extranet, OneSky Teams)
3. "Dynamic Features" Tim Wilson and David Burggraf. September 29, 2005. Contract deliverable to FAA from Galdos Systems Inc.
4. GML: Geography Markup Language. Ron Lake, David S. Burggraf, Milan Trninic, Laurie Rae. Wiley 2004.
5. Temporal Features, James Ressler, Northrop Grumman TASC, OPENGIS PROJECT DOCUMENT #06-076
6. Geographic information - Geography Markup Language (GML), ISO 19136:2007(E) 2007-03-12
7. AIXM Primer. 4.5 draft 2 Edition. EATMP-xxxxxx-xx. Nov. 28, 2005. EUROCONTROL.
8. Annex 15 to the Convention on International Civil Aviation - Aeronautical Information Services. 12th Edition. ICAO. July 2004.

APPENDIX C

AIXM

MAPEO DEL UML AL ESQUEMA XML

AIXM

UML to XML Schema Mapping

Aeronautical Information Exchange Model (AIXM)

***Copyright**: 2010 - EUROCONTROL and Federal Aviation Administration*

All rights reserved.

This document and/or its content can be download, printed and copied in whole or in part, provided that the above copyright notice and this condition is retained for each such copy.

For all inquiries, please contact:

Brett BRUNK - brett.brunk@faa.gov

Eduard POROSNICU - eduard.porosnicu@eurocontrol.int

Edition No.	Issue Date	Author	Reason for Change
0.1	2006/06/22	Brett Brunk	First Edition
	2006/08/25	Barb Cordell	Incorporate data modelling section
	2006/09/06	Vamshi Reddy	Incorporate Data type section
	2007/01/30	Barb Cordell	Update for Release Candidate 1
0.2	2007/07/31	Scott Wilson	Update for Release Candidate 2
0.3	2007/09/20	Eddy Porosnicu	Modified model for units of measurement
0.4	2008/01/15	Eddy Porosnicu	Modified rules for associations, from now on based on role names.
1.0	2008/03/15	Eddy Porosnicu	Editorial changes for first public version.
1.1	2010/02/04	Eddy Porosnicu Hubert Lepori	Updated for AIXM 5.1

CONTENTS

1 SCOPE 1
1.1 Introduction 1
1.2 References 1
2 AIXM UML MODELLING CONVENTIONS 2
2.1 Diagram types 2
2.2 Stereotypes 2
2.3 Abstract Classes 2
2.4 Features 2
2.5 Objects 3
2.6 Choice 3
2.7 Properties 4
2.7.1 Attributes 4
2.7.1.1 DataTypes 4
2.7.2 Relationships 6
2.7.2.1 Relationships to Objects 6
2.7.2.2 Relationships to Features 7
2.7.2.3 Association Classes 7
2.8 Inheritance 7
2.9 Naming 8
3 OTHER ASPECTS OF THE MODEL 9
3.1 The Abstract Model 9
3.1.1 AIXMFeature and AIXMFeatureTimeSlice Class 9
3.1.2 Metadata 10
3.1.3 Extension 10
3.2 External packages 10
3.2.1 <<XSDschema>> XMLSchemaDatatypes 10
3.2.2 ISO 19115 Metadata 10
3.2.3 ISO 19107 Geometry 11
3.2.4 ISO 19136 11
4 MAPPING TO THE AIXM XML SCHEMA 12
4.1 AIXM - core XSD files 12
4.2 AIXM is GML 12
4.3 The GML Object-Property Model 12
4.4 Mapping Inheritance 13
4.5 Mapping Name of Classes 13

4.6 Mapping Features **13**
4.6.1 An Example Mapping 13
4.6.1.1 RunwayPropertyGroup 14
4.6.1.2 RunwayTimeSliceType 16
4.6.1.3 RunwayTimeSlice 17
4.6.1.4 RunwayTimeSlicePropertyType 18
4.6.1.5 RunwayType 19
4.6.1.6 Runway 19
4.6.1.7 RunwayExtension 20

4.7 Mapping Objects **20**
4.7.1 An Example Mapping 21
4.7.1.1 AbstractCityExtension 21
4.7.1.2 CityPropertyGroup 22
4.7.1.3 CityType 22
4.7.1.4 City 23
4.7.1.5 CityPropertyType 23

4.8 Mapping Choices **24**

4.9 Mapping Relationships to Objects **25**
4.9.1 Mapping Associations with Association Classes 26

4.10 Mapping Relationships to Features **27**

4.11 Mapping Data Types **28**
4.11.1 <<codelist>> 28
4.11.2 <<datatype>> - default case 29
4.11.3 <<datatype>> with Unit of Measurement 30
4.11.4 Particular cases 31
4.11.4.1 <<datatype>> with no BaseType 31
4.11.4.2 <<datatype>> XHTMLBaseType 32

1 Scope

1.1 Introduction

The AIXM Conceptual Model is maintained as a UML class model. The AIXM exchange format is codified as a series of XML schemas. There is a direct link between the AIXM Conceptual Model and the AIXM XML Schema.

This document describes how the AIXM Conceptual Model is converted into the AIXM XML Schema. The conversion process is illustrated using a series of examples from the AIXM 5 XML schema.

1.2 References

1. Geographic Information – Spatial Schema. ISO 19107. First Edition, 2003-05-01
2. ISO 19136:2007 - Geographic information -- Geography Markup Language (GML)
3. UML 2.0 In a Nutshell. Dan Pilone. O'Reilly Media Inc. 2005.
4. AIXM Temporality Model, www.aixm.aero (see Downloads)

2 AIXM UML Modelling Conventions

2.1 Diagram types

Two types of diagrams are used in the model:

Class diagrams – Used to represent the features, properties, relationships and inheritance between features;

Package diagrams – Used to split the model into modules and identify dependencies among sets of classes.

2.2 Stereotypes

The classes are distinguished by their stereotypes. Stereotypes are used to further define and extend standard UML concepts. The main stereotype are <<feature>>, <<object>>, <<choice>>, <<datatype>> and <<codelist>>.

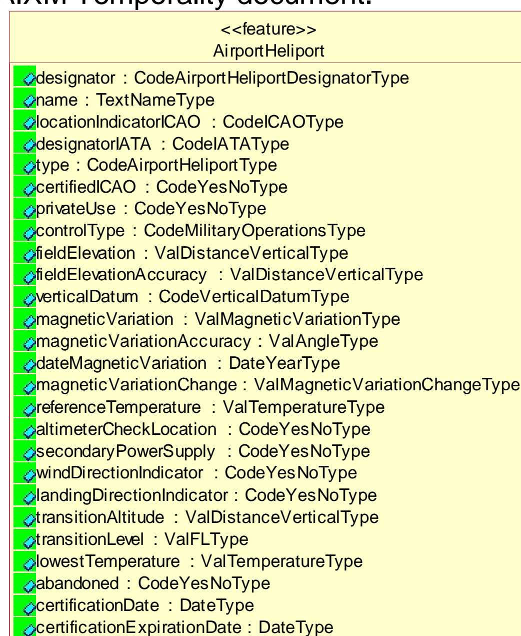
2.3 Abstract Classes

In addition, some classes are abstract. Abstract classes are designated by putting the class name in *italics*. An abstract class cannot be realised in an implementation such as an XML document. Instead, abstract classes are used as base classes in an inheritance hierarchy. For example, the AIXMFeature abstract class describes the basic properties of an AIXM Feature. Every specific AIXM Feature, such as Runway, inherits[1] from the abstract AIXMFeature class.

2.4 Features

Features describe real world entities and are fundamental in AIXM. AIXM features can be concrete and tangible, or abstract and conceptual and can change in time. Features are represented as classes with a stereotype <<feature>>. Examples include Runway and AirportHeliport.

AIXM features are dynamic features. Timeslice objects are used to describe the changes that affect the AIXM feature over time. Timeslice objects and temporality are discussed extensively in a separate AIXM Temporality document.



[1] Please see section 3.1 The Abstract Model, which explains why this inheritance is not visible in UML

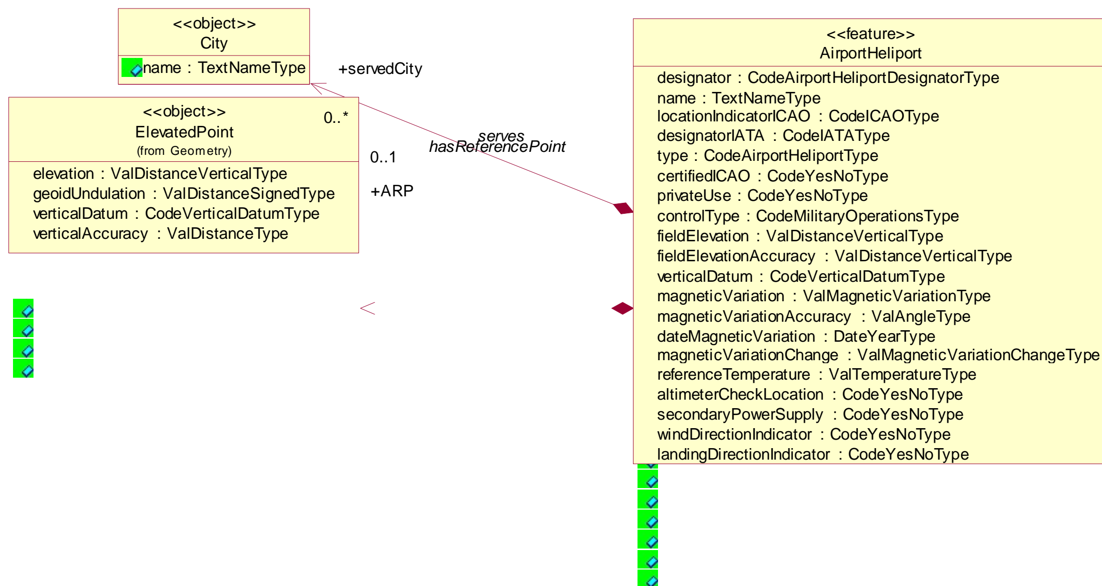
2.5 Objects

Objects are abstractions of real world entities or, more frequently, of properties of these entities, which do not exist outside of a feature. An object is created for two reasons in AIXM:

When a property has a multiplicity greater than one (such as the city served by an AirportHeliport), or

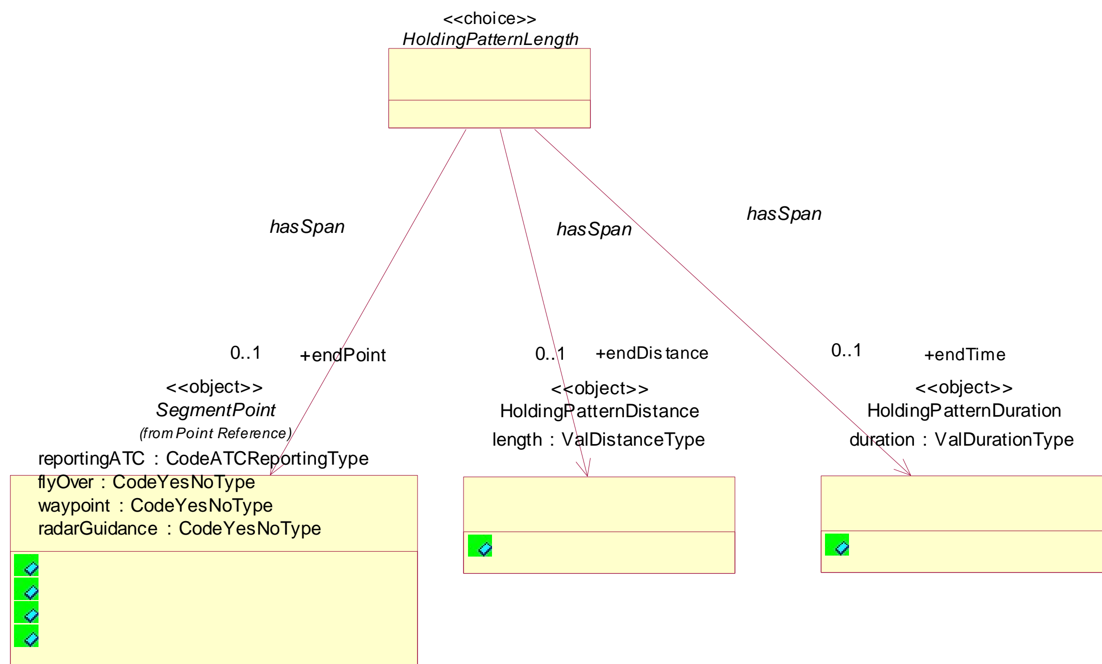
The object has its own attributes that are reused throughout the model, such as ElevatedPoint.

In both cases, the property is represented as an object with the proper UML composition relationship as shown below.



2.6 Choice

Some classes are marked as `<<choice>>`. These are used to model XOR relationships. For example, the length of a Holding Pattern can be expressed using a HoldingPatternDistance, a HoldingPatternDuration or a SegmentPoint defining the end of the outbound leg.



2.7 Properties

Properties are the attributes and relationships that characterise a feature or object. In the UML:

Attributes are used to describe simple properties of a feature or object;
Relationships are used to describe associations to features or objects. Whenever a property has a multiplicity greater than one, it is described using a UML relationship with cardinality.

2.7.1 Attributes

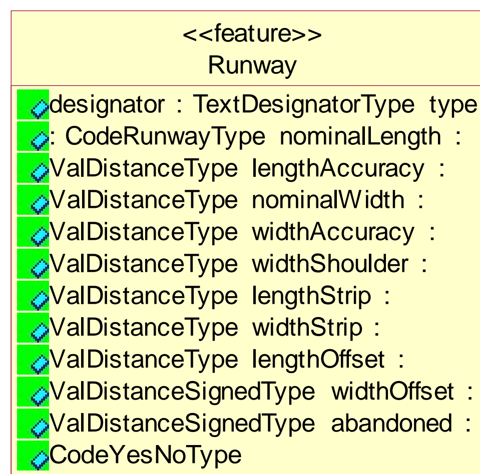
Simple properties of cardinality one are shown as attributes in the UML diagram.

An attribute has the following format:
Visibility / stereotype name : type multiplicity

For AIXM 5 the following values are used:

Visibility – Public
/ – not used
Stereotype – not used
Name – name of the property
Type – property type
Multiplicity – usually not specified; for reasons related to the AIXM Temporality model, an implementation should assume that all properties are optional, [0..1]

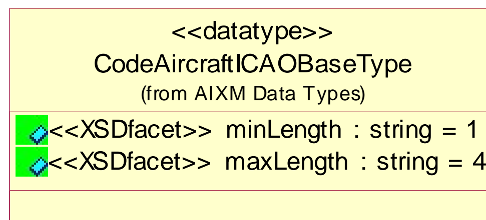
To illustrate, the Runway feature has several simple properties e.g. designator and type. These properties are assigned a datatype; for example, the designator attribute is of type TextDesignatorType.



2.7.1.1 DataTypes

The UML model lists the datatypes that are used throughout the AIXM. These are given one of the two following stereotypes:

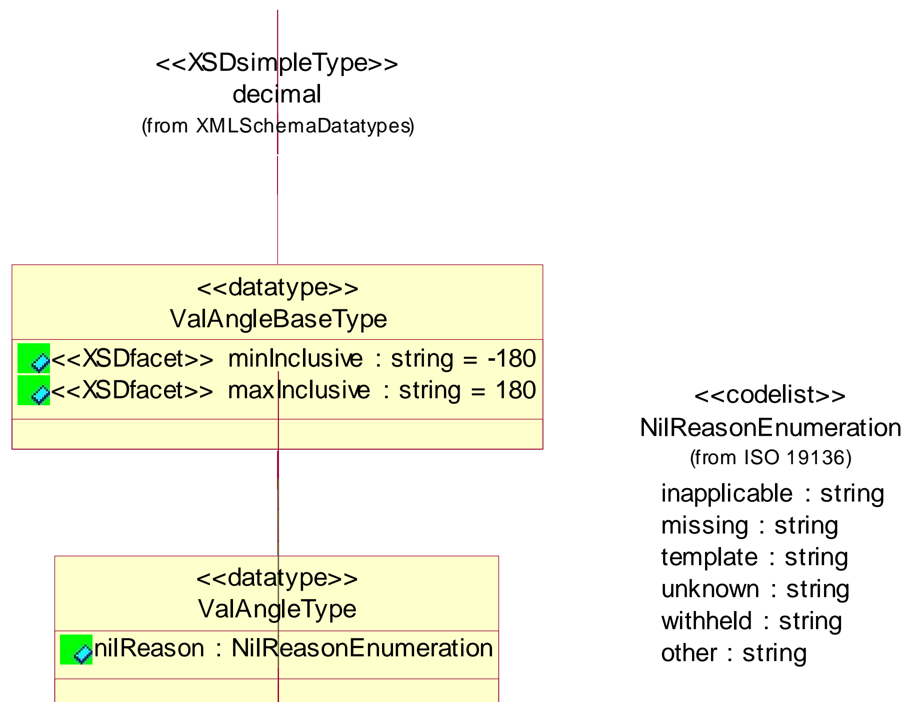
<<datatype>> - This is basic data type that specifies a pattern to use.
<<codelist>> - This is a data type which codes a predefined list of values. The <<codelist>> includes the value OTHER which can be expanded with some free text in uppercase (“OTHER:*MY_VALUE*”) to support un-supported values.



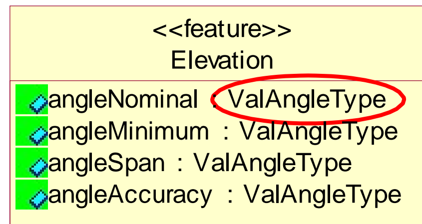
All the data types used to type AIXM simple properties define a nilReason, which is used to indicate the reason for a null value. This is realized in AIXM 5.1 by introducing

A base type, which contains the core “business” information, such as a range of value for <<datatype>>, or the list of string values for <<codelist>>

A derived data type, which explicitly declares the nilReason attribute, and which is used to type the corresponding AIXM simple properties.

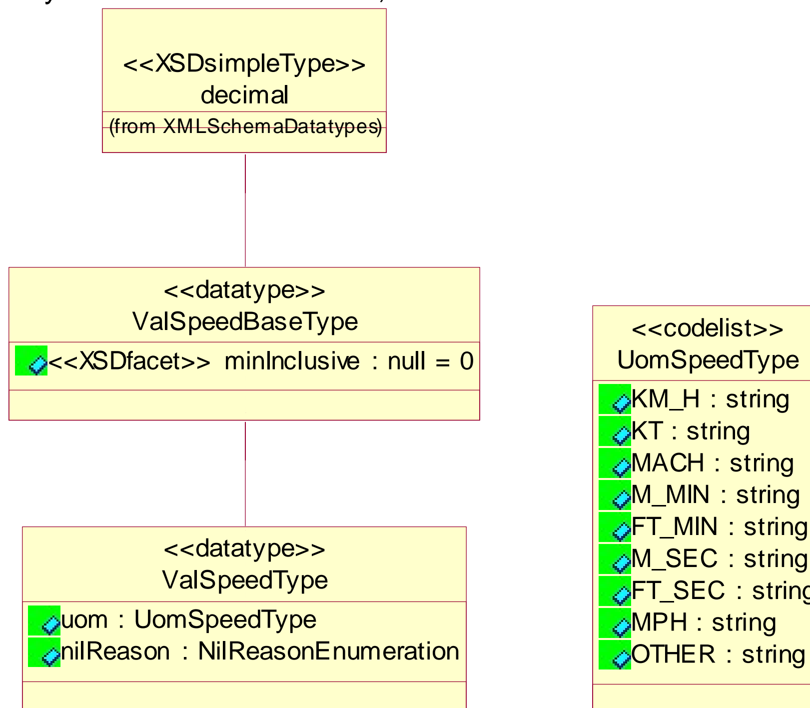


On the example above, the base type used to represent an angle is named ValAngle**BaseType**. It derives from decimal and defines the range of values allowed for an angle percentage ([-180;180]). The derived datatype ValAngle**Type** inherits from ValAngleBaseType and includes the nilReason, typed with NilReasonEnumeration. ValAngle**Type** is always used to type the percentages specified in AIXM features or AIXM objects.



A limited set of data types defined in the AIXM 5.1 UML model are not used to type directly AIXM simple properties but are basic classes from which several AIXM data types inherit. These data types are: AlphaType, AlphaNumericType, Character1, Character2, Character3. They do not require a nilReason attribute, and consequently, no corresponding BaseType types are defined in the AIXM UML model.

In addition, certain `<<datatype>>` might have an associated Unit Of Measurement. This is indicated in the model by the inclusion of a "uom" attribute at the same level as the nilReason attribute, i.e in the definition of the derived `<<datatype>>` class. The type of the uom attribute is typically a `<<codelist>>` class, as shown below:



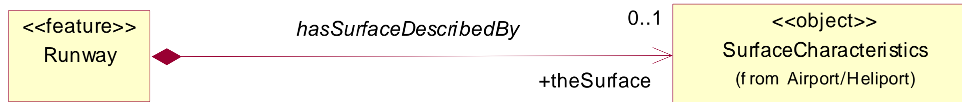
Note that the <<codelist>> types representing Units of Measurement do not require a nilReason. As a consequence, no base type is created for uom.

2.7.2 Relationships

Whenever a property has a multiplicity greater than one, it can not be described in UML with an attribute. In that case, the property is described using a UML relationship which specifies the cardinality and which is always navigable in one and only one direction. The name of the complex property is given by the name of the role played by the targeted class.

2.7.2.1 Relationships to Objects

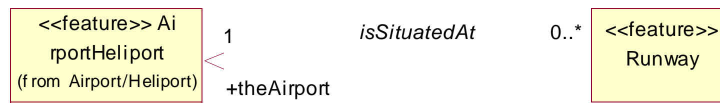
Relationships to objects are depicted by the standard UML composition (*aggregation by value)* association. Composition is a form of aggregation with strong ownership and coincident lifetime of the parts by the whole. The part is removed when the whole is removed.



The example above shows that the <<feature>> Runway has a property named *theSurface.* This property is modelled in UML using a composition association between the <<feature>> Runway and an object representing the characteristics of a geometric surface.

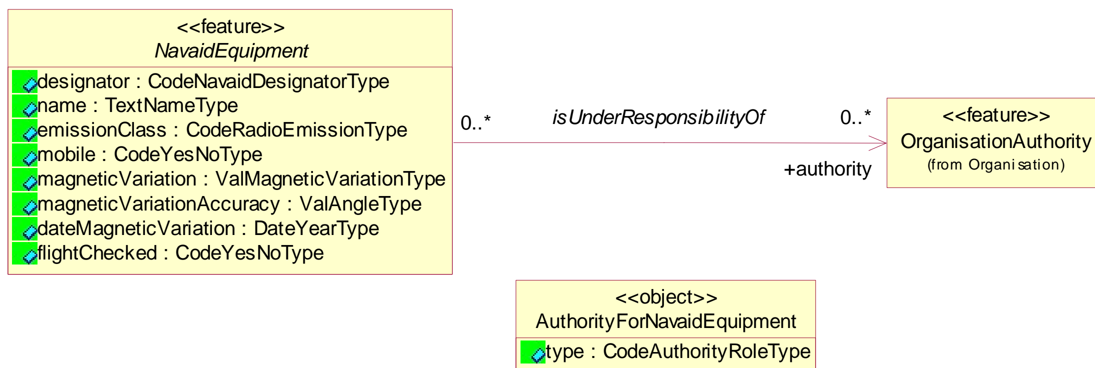
2.7.2.2 Relationships to Features

Relationships to features are described with a standard UML association. All of the associations are navigable in only one direction. This shows that the two classes are related but only one class knows that the relationship exists. In the example below the Runway feature knows about the AirportHeliport but the AirportHeliport does not know about the Runway.



2.7.2.3 Association Classes

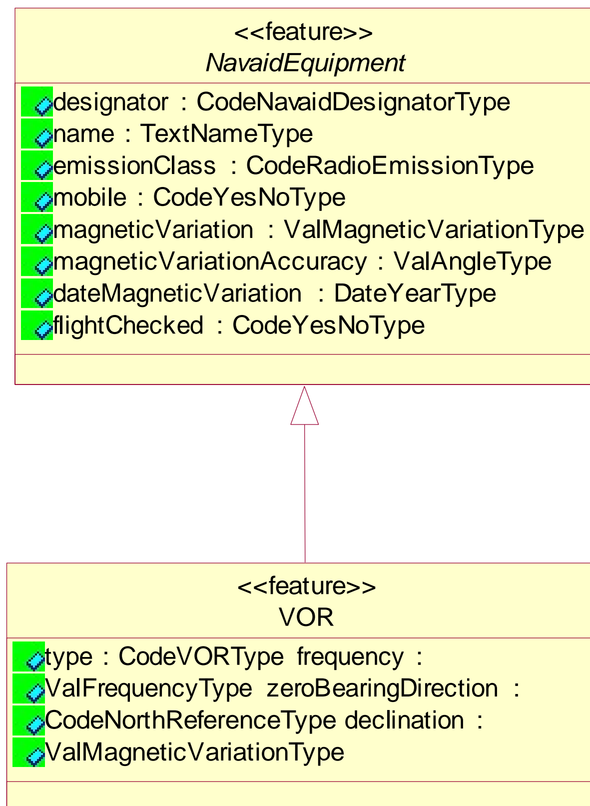
When information about a relationship is required, a UML association class is used. The association class is attached to the relationship with a dotted line.



2.8 Inheritance

Inheritance refers to the ability of one class (the specialized or child class) to inherit the properties of another class (the generalized or parent class), and then add new properties of its own. In AIXM, Features must only inherit from other Features and Objects must only inherit from other Objects. Multiple inheritance is not allowed.

In the example below the VOR is a kind of NavaidEquipment.



2.9 Naming

Feature, Object and Choice names are written in UpperCamelCase e.g. NavaidEquipment.

Simple property names (i.e. attributes) are written in lowerCamelCase e.g. widthShoulder. Relationship names are written in lowerCamelCase but as present tense verbs e.g. isSituatedAt. Relationship Role names are also written in lowerCamelCase and they are nouns that express the role played by the class in the association.

Datatype names are written in UpperCamelCase and end with ‘Type’ e.g. CodeAircraftType.

3 Other Aspects of the Model

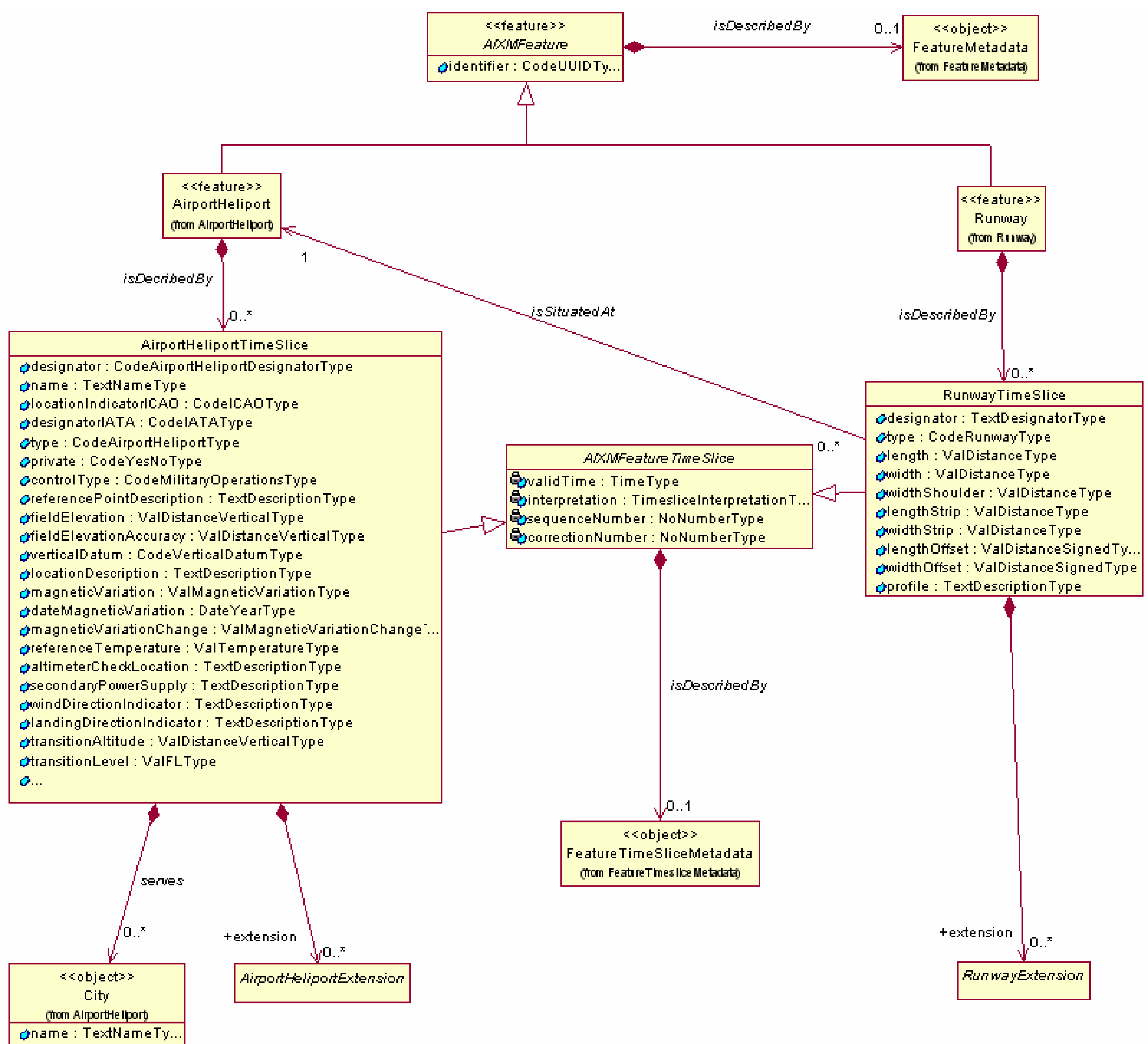
To simplify the UML model some convenience steps have been taken. Some elements are not shown on every diagram and some relationships are ‘assumed’.

3.1 The Abstract Model

The model should contain a set of abstract AIXM classes that are used as the building blocks for the AIXM XML Schema. However, for simplicity, these relationships are not shown on any diagram and do not really exist in the UML. They are just assumed to exist, when converting from the UML model to the XML Schema of AIXM.

3.1.1 AIXMFeature and AIXMFeatureTimeSlice Class

The UML below shows how each and every `<<feature>>` inherits from the abstract AIXMFeature class. The concrete features are described by TimeSlices which are composed of properties. The TimeSlice inherits from the abstract AIXMFeatureTimeSlice class.



The diagram above is quite complex. If applied to the whole set of AIXM classes, it might undermine the readability of the UML diagrams. Therefore, the Design Team has decided to

provide a simplified UML model, without visible inheritance of all features from the abstract AIXMFeature and without visible SomeFeatureTimeSlice classes.

However, all of these relationships and classes must be mapped in the AIXM XML Schema.

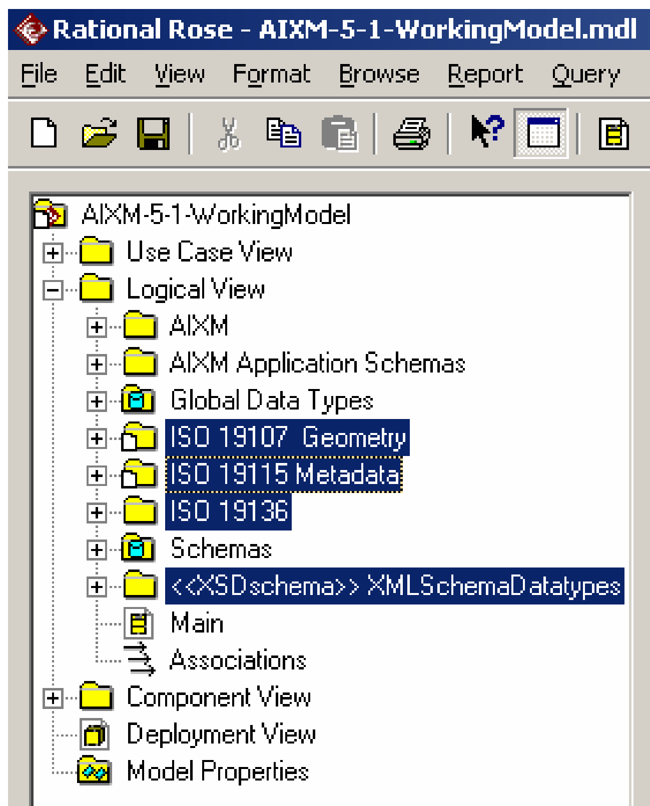
3.1.2 Metadata

The diagram also shows that each AIXM Feature and each TimeSlice is described by metadata. The AIXM XML schema incorporates the ISO 19139 metadata elements - see 3.2.2.

3.1.3 Extension

Finally, each TimeSlice may contain an Extension. The Extension mechanism allows each user of AIXM5 to define and use his own specific attributes and classes.

3.2 External packages



3.2.1 <<XSDschema>> XMLSchemaDatatypes

The XSD Schema Datatypes package declares XSD specific data types that are referenced by AIXM data types, when generating the AIXM XML (XSD) Schema. However, these XSD bindings do not mean that AIXM is "dependent" on the XML Schema specification. The pre-defined XSD simple types (such as string, decimal, unsignedInt, etc.) referenced by AIXM are sufficiently generic and mappable to the simple data types of many other data encoding standards.

3.2.2 ISO 19115 Metadata

This package contains some basic connections from the AIXM model to the ISO 19115 Metadata elements (MD_Metadata, MD_Constraints …).

3.2.3 ISO 19107 Geometry

This package contains some basic connections from the AIXM model to the ISO 19107 geometry elements (GM_Point, GM_Surface …).

3.2.4 ISO 19136

This package contains some basic connections from the AIXM model to GML specific elements, which are not part of the ISO 19107. Practically, the package contains only the data type NilReasonEnumeration, used to indicate the reason for a null value.

4 Mapping to the AIXM XML Schema

4.1 AIXM - core XSD files

The core AIXM exchange format is composed of three main files:

AIXM_AbstractGML_ObjectTypes.xsd: the file references the ISO19139 Metadata Schema and defines the base AIXM Feature/Object constructs

- o AbstractAIXMFeatureType / AbstractAIXMFeature
- o AbstractAIXMTimesliceType / AbstractAIXMTimeslice
- o AbstractAIXMObjectType
- o AbstractAIXMPropertyType, which defines the nilReason for all the AIXM complex properties

AIXM_Datatypes.xsd: this file contains the XML representation of all the data types defined in the AIXM UML model.

AIXM_Features.xsd: this file contains the XML representation of all the AIXM features with all their properties (simple and complex).

The chapters here after specify the rules that govern the mapping between the AIXM UML model and the AIXM XML Schema.

4.2 AIXM is GML

The AIXM exchange model is an XML exchange standard based on a subset of the Geography Markup Language (GML). Essentially:

AIXM Features are GML features;

AIXM Objects are GML objects;

AIXM follows the GML object-property concept.

4.3 The GML Object-Property Model

The GML object-property model explains some of the complexity of the AIXM UML to XSD mapping. It means that no GML object may appear as the immediate child of a GML object. Consequently, no element may be both a GML object and a GML property.

The object-property model prohibits the encoding of an object directly inside a feature, e.g.

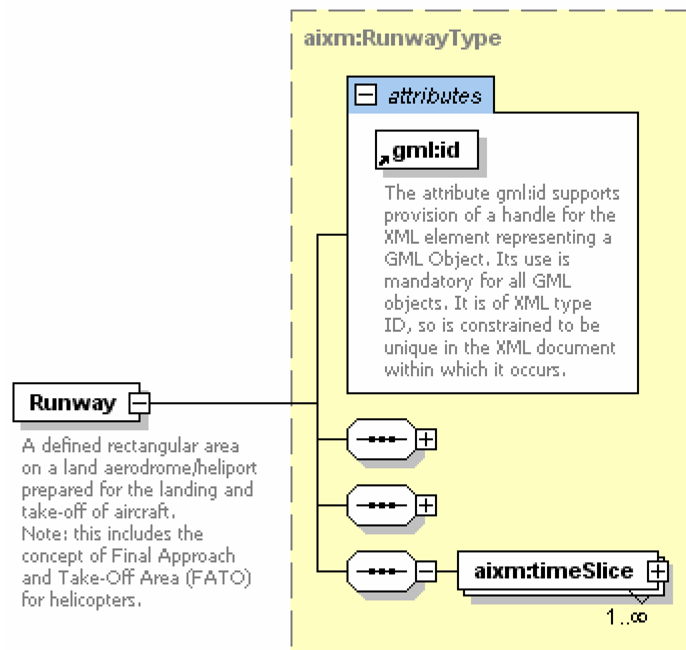
```
<AirportHeliport> <!-- feature -->
 <ElevatedPoint> <!-- object -->
```

Instead, in a compliant GML application schema, an association between two features (or a feature and an object) is implemented over a property of the feature, e.g.

```
<AirportHeliport> <!-- feature -->
 <hasReferencePoint> <!-- property -->
  <ElevatedPoint> <!-- object -->
```

The direction of the association arrow from the UML diagrams (the navigability) dictates which of the two association partners has the property that associates the other.

In the AIXM XML Schema, the object-property model is encoded by declaring a type and then assigning properties (attributes and relationships) to that type. The type defines the object.



4.4 Mapping Inheritance

Within the AIXM XML Schema, inheritance implies two characteristics:

1. Substitutability. The more general feature or object can be substituted by a specialization. In the XML schema this is supported using substitution groups.
2. Property inheritance. The specialized feature inherits all of the properties of the more general feature. In the XML schema including the properties of the general class into the specialized class supports this.

4.5 Mapping Name of Classes

The UML class name is used for the element names in the XML Schema.

4.6 Mapping Features

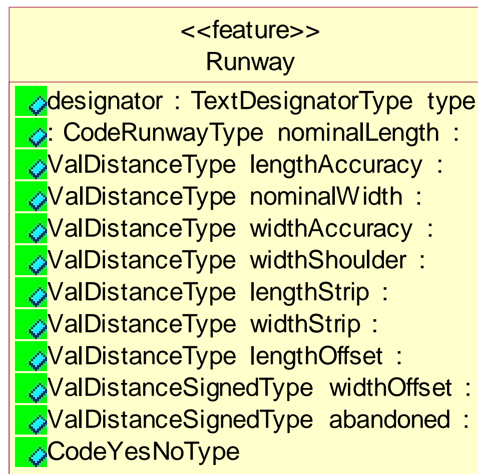
For each AIXM Feature in the UML, the following XML schema entities are created:

*Feature*PropertyType
*Feature Feature*Type
*Feature*TimeSlicePropertyType
*Feature*TimeSlice.
*Feature*TimeSliceType
*Feature*PropertyGroup
Abstract*Feature*Extension

The direction in which the different types and elements are used in the schema definition (e.g. Feature uses FeatureType)

4.6.1 An Example Mapping

The Runway feature (shown below) will be used to illustrate the mapping. The example will concentrate on the properties (shown as attributes).



4.6.1.1 RunwayPropertyGroup

An XML Schema (XSD) group is generated for each feature containing all of the properties (attributes and relationships) of the feature.

The order in which the chilled elements of the group are declared is the following:

1. (if applicable) in the case of derived classes only, the property group of the super class is inserted first;
2. then, all the elements that correspond to class attributes, in the order that they appear in the UML class diagram
3. then, all the elements that correspond to association role names, in random order;
4. last the “annotation” property – note that for derived classes this property is only defined in the super class, therefore it will appear in the property group of the super class.

Below is an example of the RunwayPropertyGroup in graphic form and as an extract from the XSD. It shows clearly how the attributes are mapped from the UML to the XSD and how the relationship ‘associatedAirportHeliport’ is created.

aixm:designator

The full textual designator of the runway, used to uniquely identify it at an aerodrome/heliport which has more than one.
E.g. 09/27, 02R/20L, RWY 1.

aixm:type

The type can be either runway for airplanes or final approach and take off area (FATO) for helicopters.

aixm:nominalLength

The declared longitudinal extent of the runway for operational (performance) calculations.

aixm:lengthAccuracy

Accuracy of the value of the physical length of the runway.

aixm:nominalWidth

The declared transversal extent of the runway for operational (performance) calculations.

aixm:widthAccuracy

Accuracy of the value of the physical width of the runway.

aixm:widthShoulder

The value of the runway shoulder width.

aixm:lengthStrip

The value of the physical length of the strip. The runway strip is a defined area including the runway and, if applicable, the stopway. It is intended (a) to reduce the risk of damage to aircraft running off the runway and (b) to protect aircraft flying over the runway during take-off or landing operations.

aixm:widthStrip

The value of the physical width of the strip.

aixm:lengthOffset

A value specifying the longitudinal offset of the strip, when it is not symmetrically extended beyond the two runway ends.

Notes: The longitudinal offset defines the distance along the centreline from the middle of the runway centreline towards the middle of the strip centreline. An offset in the direction defined from the threshold with the lower runway direction designation number towards the opposite runway threshold is indicated by a positive value. An offset in the opposite sense is indicated by a negative value.

Example: a runway oriented 09/27 has a strip that is extending 120 m before the threshold of the runway direction 09 and only 100 m before the threshold of the runway direction 27. The value of the longitudinal offset will be -10 m.

aixm:widthOffset

A value specifying the lateral offset of the strip, when it is not symmetrically extended beyond the two runway edges.

Note: The lateral offset defines the distance from the runway centreline to the strip centreline in direction perpendicular to the runway centreline. An offset to the right, based on the direction defined from the threshold with the lower runway direction designation number towards the opposite runway threshold, is indicated by a positive value. An offset to the left is indicated by a negative value.

Example: a runway oriented 09/27 has a strip that is extending 150 m to the right of the runway direction 09 and 300 m to the left of the same runway direction. The value of the lateral offset will be -75 m.

aixm:abandoned

Indicating that the surface is no longer in operational use, but it is still physically present and visible, although usually in a degraded state.

aixm:surfaceProperties

Identifies the surface characteristics of the runway.

aixm:associatedAirportHeliport

Identifies the Airport where the Runway is situated.

aixm:overallContaminant 0..∞

Identifies the contaminant of the runway.

aixm:areaContaminant 0..∞

aixm:annotation 0..∞

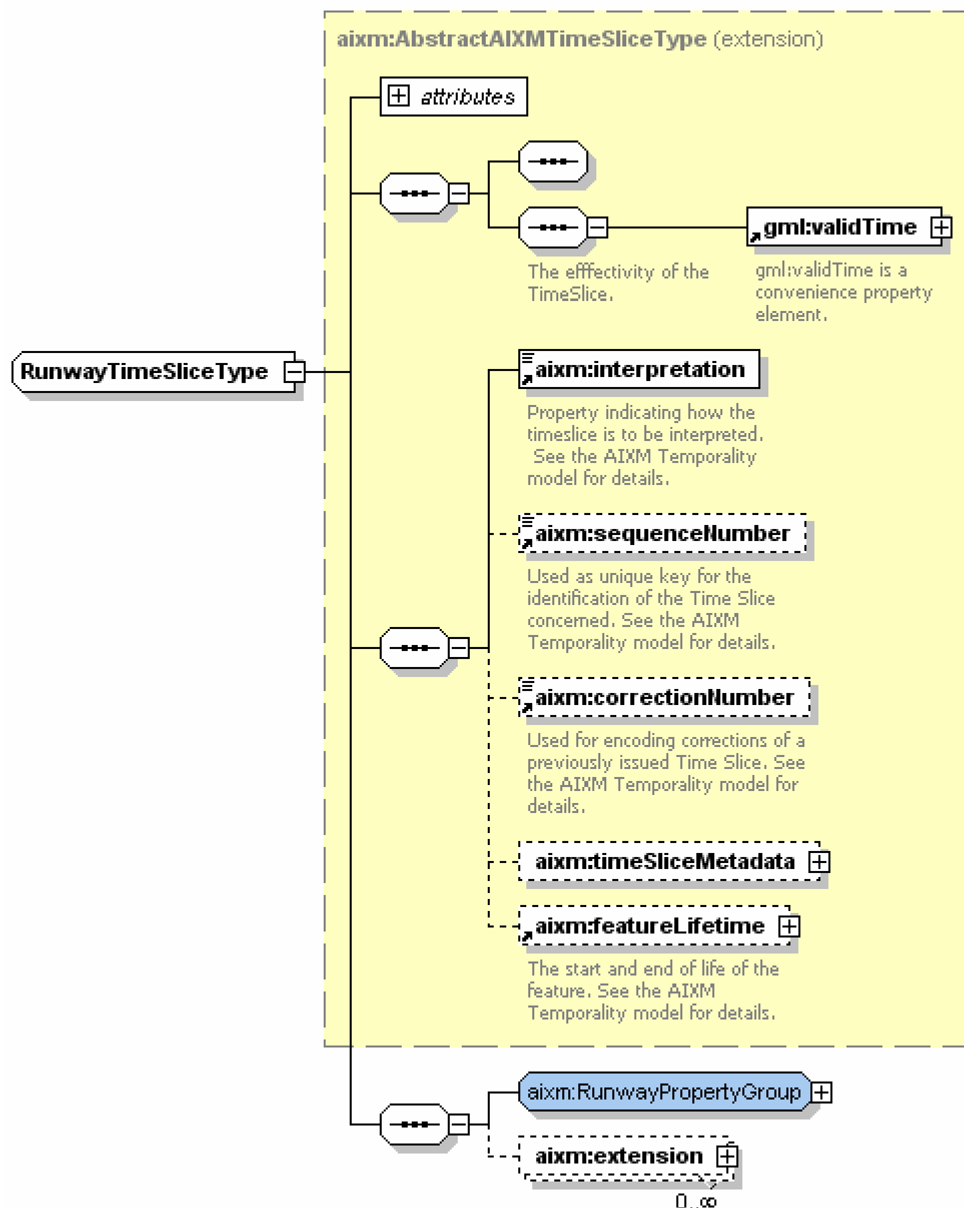
Generated by XmlSpy www.altova.com

```
<group name="RunwayPropertyGroup">
   <sequence>
     <element name="designator" type="aixm:TextDesignatorType"
nillable="true" minOccurs="0"/>
     <element name="type" type="aixm:CodeRunwayType" nillable="true"
minOccurs="0"/>
     <element name="nominalLength" type="aixm:ValDistanceType"
nillable="true" minOccurs="0"/>
     <element name="lengthAccuracy" type="aixm:ValDistanceType"
nillable="true" minOccurs="0"/>
     ...
     <element name="associatedAirportHeliport"
type="aixm:AirportHeliportPropertyType" nillable="true" minOccurs="0"/>
     ...
     <element name="areaContaminant" type="aixm:
RunwayContaminationPropertyType" nillable="true" minOccurs="0"
maxOccurs="unbounded"/>
     <element name="annotation" type="aixm:NotePropertyType" nillable="true"
minOccurs="0" maxOccurs="unbounded"/>
   </sequence>
</group>
```

4.6.1.2 RunwayTimeSliceType

The properties of a feature or the target of any feature relationship can change within the lifetime of the feature. This temporality can be expressed in GML by using dynamic features and feature collections. The TimeSlice property of a dynamic feature contains one or more Feature TimeSlices that capture the evolution of the feature over time. A gml:TimeSlice object contains the dynamic properties of the feature.

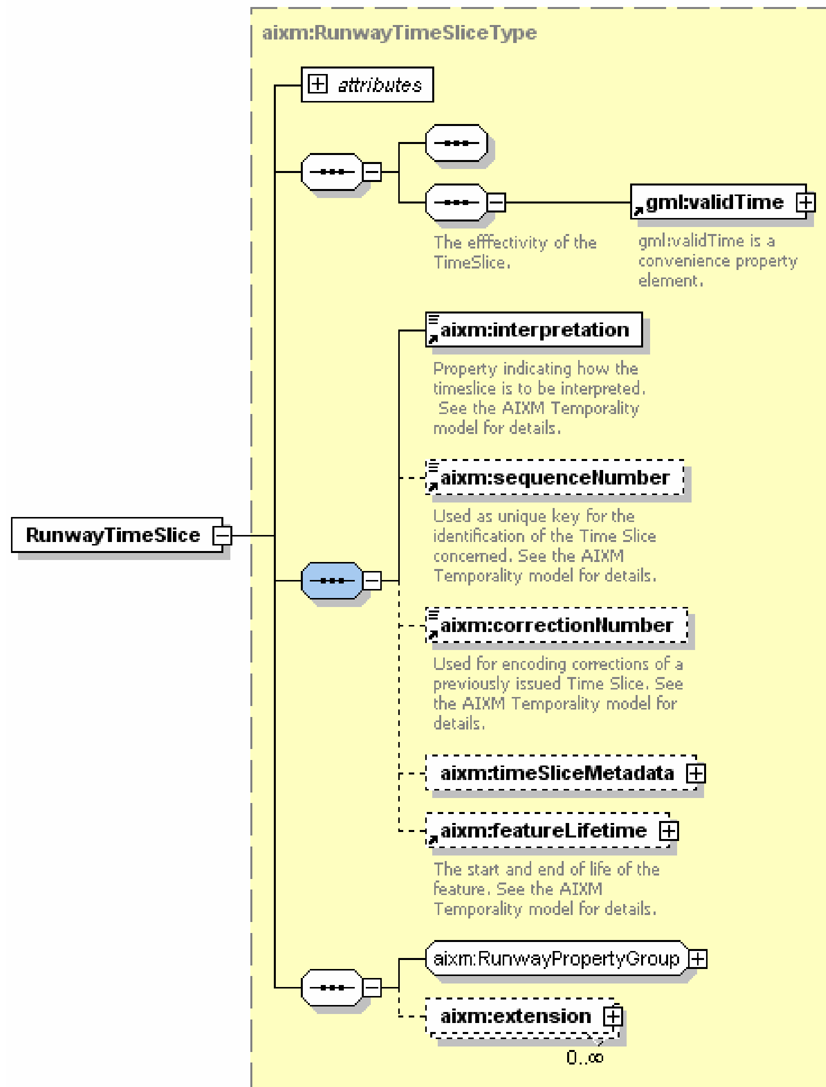
For each feature a TimeSlice property is created that contains an array of feature TimeSlice objects. This example shows the RunwayTimeSliceType encapsulating all of the Runway properties (RunwayPropertyGroup created above) that change over time.




```
<complexType name="RunwayTimeSliceType">
 <complexContent>
  <extension base="aixm:AbstractAIXMTimeSliceType">
   <sequence>
    <group ref="aixm:RunwayPropertyGroup"/>
    <element name="extension" minOccurs="0" maxOccurs="unbounded">
     <complexType>
      <sequence>
       <element ref="aixm:AbstractRunwayExtension"/>
      </sequence>
      <attributeGroup ref="gml:OwnershipAttributeGroup"/>
     </complexType>
    </element>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

4.6.1.3 RunwayTimeSlice

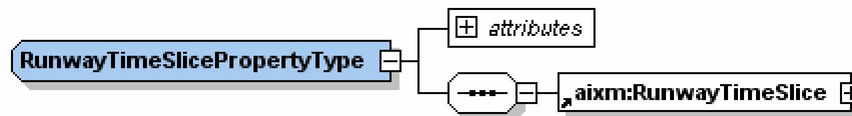
The *Feature*TimeSlice object is of type *Feature*TimeSliceType. Continuing the example, the RunwayTimeSlice element is of type RunwayTimeSliceType.




```
<element name="RunwayTimeSlice" type="aixm:RunwayTimeSliceType"
substitutionGroup="gml:AbstractTimeSlice"/>
```

4.6.1.4 RunwayTimeSlicePropertyType

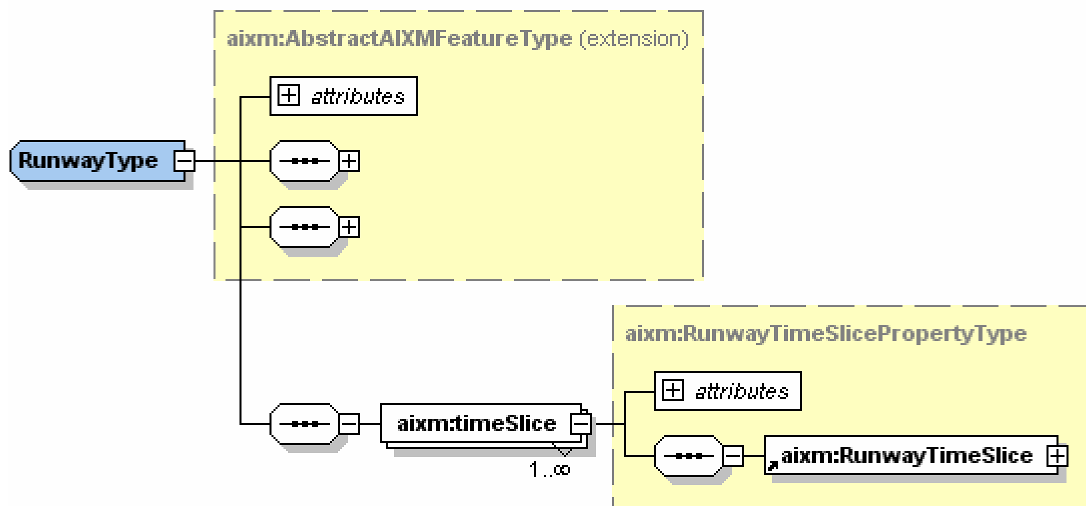
A GML property type containing a *Feature*TimeSlice objects is created.




```
<complexType name="RunwayTimeSlicePropertyType">
 <sequence>
  <element ref="aixm:RunwayTimeSlice"/>
 </sequence>
 <attributeGroup ref="gml:OwnershipAttributeGroup"/>
</complexType>
```

4.6.1.5 RunwayType

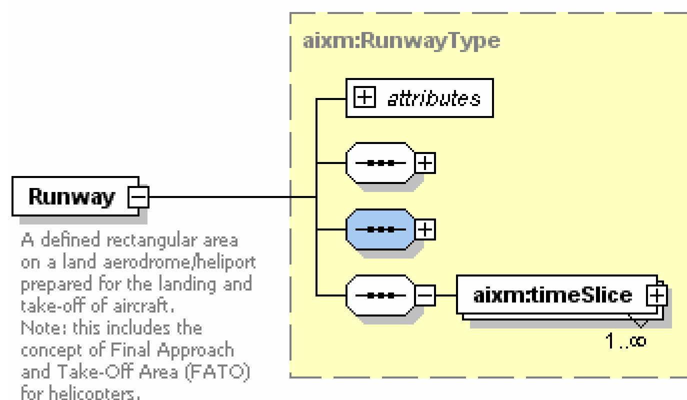
Continuing with the object-property model, the Runway feature type is created extending the AbstractAIXMFeatureType with the RunwayTimeSlice object created above.




```
<complexType name="RunwayType">
 <complexContent>
  <extension base="aixm:AbstractAIXMFeatureType">
   <sequence>
    <element name="timeSlice" type="aixm:RunwayTimeSlicePropertyType"
maxOccurs="unbounded"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

4.6.1.6 Runway

The Runway feature is then defined by the RunwayType.




```
<element name="Runway" type="aixm:RunwayType"
substitutionGroup="aixm:AbstractAIXMFeature">
 <annotation>
  <appinfo>RWY</appinfo>
  <appinfo><gml:description>A defined rectangular area on a land
aerodrome/heliport prepared for the landing and take-off of aircraft. Note:
this includes the concept of Final Approach and Take-Off Area (FATO) for
helicopters.</gml:description></appinfo>
 </annotation>
</element>
```

4.6.1.6.1 RunwayPropertyType

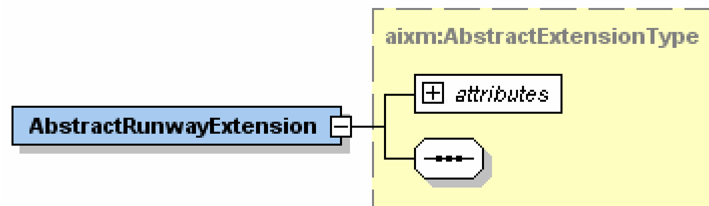
When a property of a feature is a relationship, the relationship must be associated to another feature or object. This is done through the creation of the *Feature*PropertyType, in this case, the RunwayPropertyType.

In AIXM, when the relationship or association points to another feature, the feature is referenced using the `xlink:href` attribute (it's always a "remote" encoding). When the relationship points to an object, the object is included inside the parent. (Objects cannot exist without the parent.) Since a Runway is a feature the RunwayPropertyType is created with the attribute xlink:href .

```
<complexType name="RunwayPropertyType">
 <attributeGroup ref="gml:OwnershipAttributeGroup"/>
 <attributeGroup ref="gml:AssociationAttributeGroup"/>
</complexType>
```

4.6.1.7 RunwayExtension

All Features and Objects can be extended. A relationship is created with an abstract XML element that acts as the root for all extensions. Below is an example of the extension for the Runway feature. The AbstractRunwayExtension element uses the AbstractExtensionType as shown below.




```
<element name="AbstractRunwayExtension" type="aixm:AbstractExtensionType"
abstract="true" substitutionGroup="aixm:AbstractExtension"/>
```

4.7 Mapping Objects

AIXM objects are encoded as GML objects. For the most part, the XML schema entities are created in the same way as for Features, following the object-property model. However it is important to remember that AIXM objects do not exist outside of a feature and are therefore part of the feature timeslice. TimeSlice types and elements are not created for objects.

For each AIXM Object the following XML schema entities are created:

*Object*PropertyGroup
*Object Object*Type
*Object*PropertyType
Abstract*Object*Extension

*Object*Type is complex type which extends AbstracAIXMObjectType.

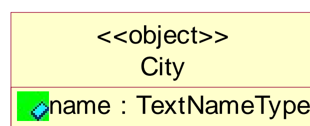
```
<complexType name="NavaidEquipmentDistanceType">
  <complexContent>
    <extension base="aixm:AbstractAIXMObjectType">
      <sequence>
        <group ref="aixm:NavaidEquipmentDistancePropertyGroup"/>
        <element name="extension" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element ref="aixm:AbstractNavaidEquipmentDistanceExtension"/>
            </sequence>
            <attributeGroup ref="gml:OwnershipAttributeGroup"/>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

*Object*PropertyType type is a complex type which extends aixm:AbstractAIXMPropertyType.

```
<complexType name="NavaidEquipmentDistancePropertyType">
  <complexContent>
    <extension base="aixm:AbstractAIXMPropertyType">
      <sequence>
        <element ref="aixm:NavaidEquipmentDistance"/>
      </sequence>
      <attributeGroup ref="gml:OwnershipAttributeGroup"/>
    </extension>
  </complexContent>
</complexType>
```

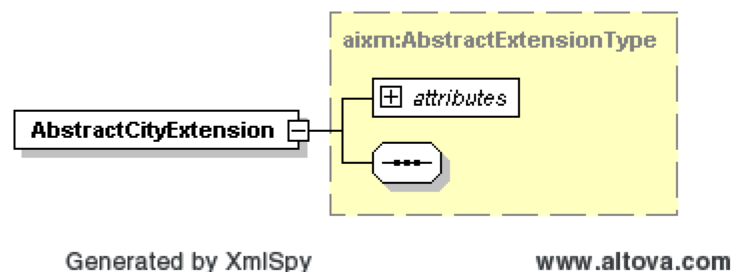
4.7.1 An Example Mapping

The example will use the City object illustrated below. The object represents the town that is served by an AirportHeliport.



4.7.1.1 AbstractCityExtension

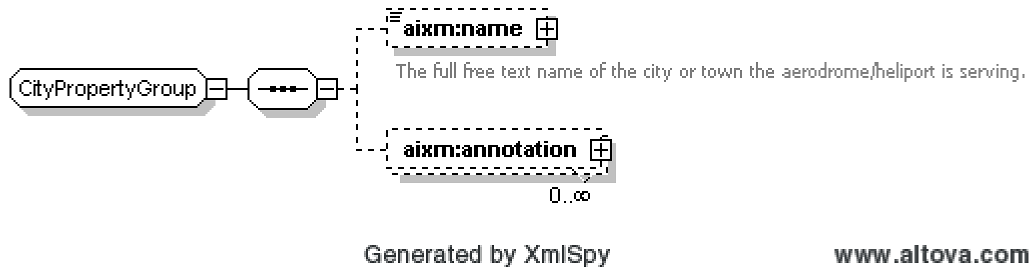
An abstract XML element acts as the root for all extension to the City object. Object extensions are defined in the same way as Feature extensions.




```
<element name="AbstractCityExtension" type="aixm:AbstractExtensionType"
abstract="true" substitutionGroup="aixm:AbstractExtension"/>
```

4.7.1.2 CityPropertyGroup

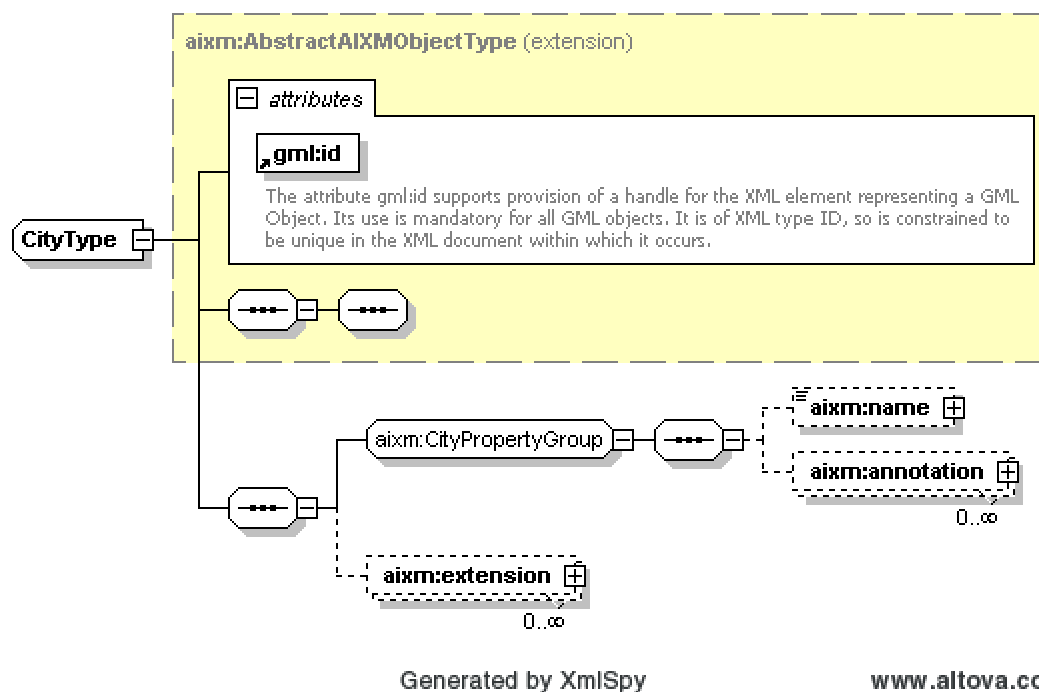
An XSD group containing the properties of the City object is created, again similar to Features.




```
<group name="CityPropertyGroup">
  <sequence>
    <element name="name" type="aixm:TextNameType" nillable="true"
minOccurs="0">
      <annotation>
        <appinfo>AIXM 4.5</appinfo>
        <appinfo><gml:description>The full free text name of the city or town
the aerodrome/heliport is serving.
        </gml:description></appinfo>
      </annotation>
    </element>
    <element name="annotation" type="aixm:NotePropertyType" nillable="true"
minOccurs="0" maxOccurs="unbounded">
    </element>
  </sequence>
</group>
```

4.7.1.3 CityType

The CityType definition uses the CityPropertyGroup and the extension. It extends AbstractAIXMObjectType

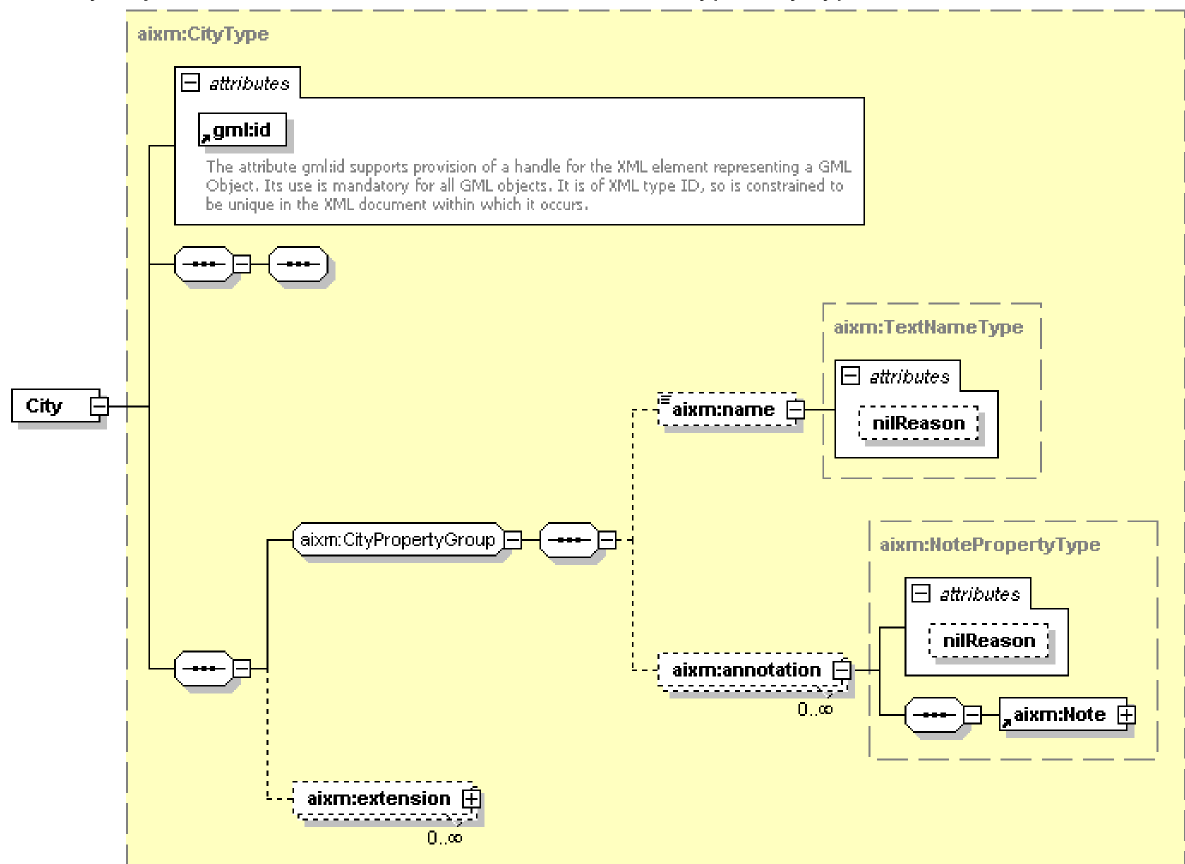



```
  <complexType name="CityType">
    <complexContent>
      <extension base="aixm:AbstractAIXMObjectType">
        <sequence>
```

```
          <group ref="aixm:CityPropertyGroup"/>
          <element name="extension" minOccurs="0" maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element ref="aixm:AbstractCityExtension"/>
              </sequence>
              <attributeGroup ref="gml:OwnershipAttributeGroup"/>
            </complexType>
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

4.7.1.4 City

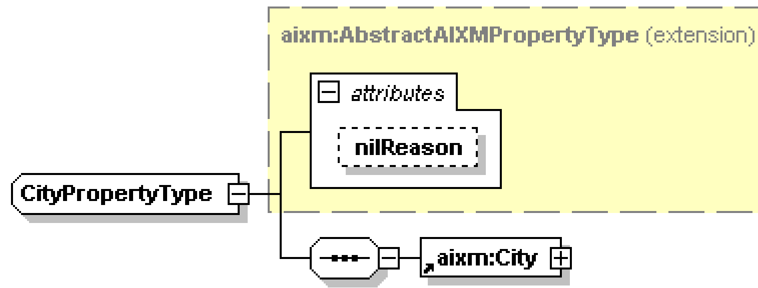
The City object is then defined as an XSD element of type CityType.




```
<element name="City" type="aixm:CityType">
  <annotation>
    <appinfo><gml:description>A city or location that may be served by an
airport/heliport.</gml:description></appinfo>
  </annotation>
</element>
```

4.7.1.5 CityPropertyType

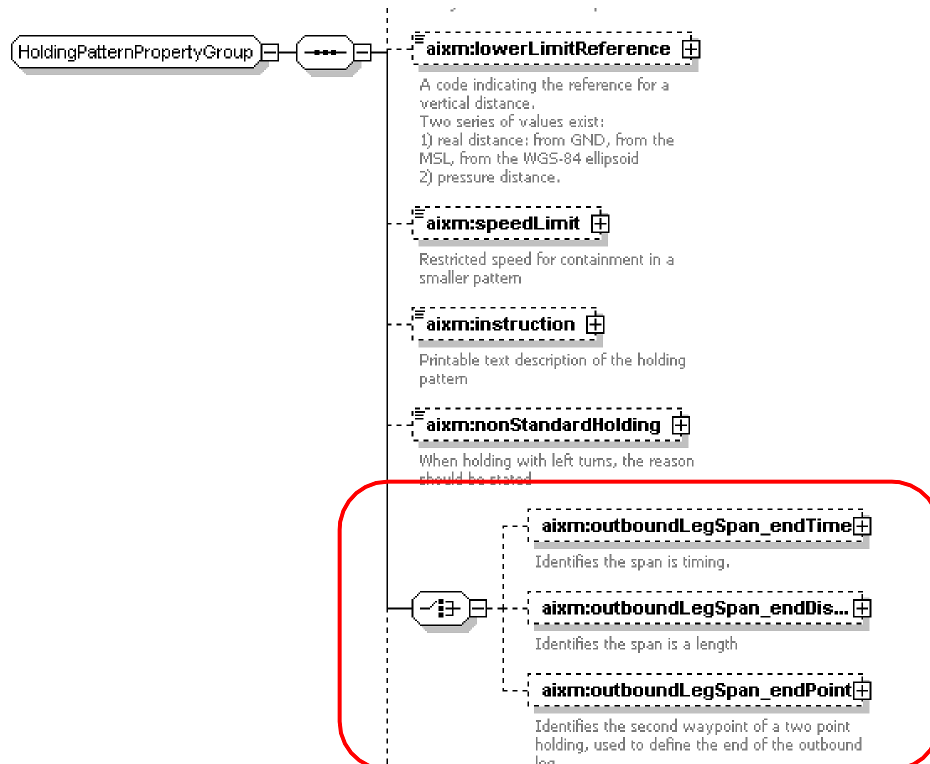
An XSD complex type representing a GML property type is created. A Feature uses this element to include the City object rather than reference it (using `xlink:href`) because object does not exist without the parent.




```
  <complexType name="CityPropertyType">
    <complexContent>
      <extension base="aixm:AbstractAIXMObjectType">
        <sequence>
          <element ref="aixm:City"/>
        </sequence>
        <attributeGroup ref="gml:OwnershipAttributeGroup"/>
      </extension>
    </complexContent>
  </complexType>
```

4.8 Mapping Choices

Classes marked with the stereotype <<choice>> do not appear in the XML Schema. Instead, the choice of elements is created.



The name of the element is the concatenation of the role of the <<choice>> class with the role of the target class of each choice branch, separated by “_”.

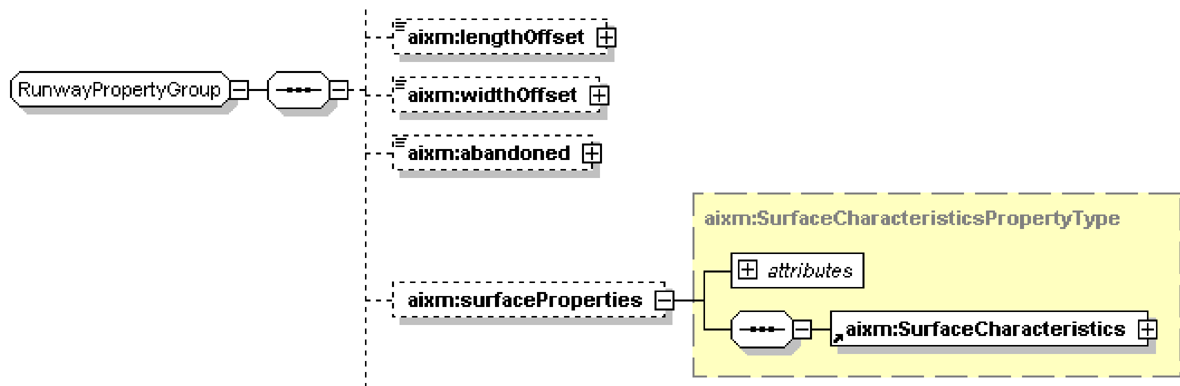
```
<group name="HoldingPatternPropertyGroup">
  <sequence>
    ............
      <element name="nonStandardHolding" type="aixm:CodeYesNoType"
nillable="true" minOccurs="0">
```

```
      <annotation>
        <appinfo>
          <gml:description>ndicates whether the HoldingPattern is non-
standard, for example because it uses left-hand turns.</gml:description>
        </appinfo>
      </annotation>
    </element>
    <choice>
      <element name="outboundLegSpan_endTime"
type="aixm:HoldingPatternDurationPropertyType" nillable="true"
minOccurs="0">
        <annotation>
          <appinfo>
            <gml:description>Span is timing</gml:description>
          </appinfo>
        </annotation>
      </element>
      <element name="outboundLegSpan_endDistance"
type="aixm:HoldingPatternDistancePropertyType" nillable="true"
minOccurs="0">
        <annotation>
          <appinfo>
            <gml:description>span is length</gml:description>
          </appinfo>
        </annotation>
      </element>
      <element name="outboundLegSpan_endPoint"
type="aixm:SegmentPointPropertyType" nillable="true" minOccurs="0">
        <annotation>
          <appinfo>
            <gml:description>The second waypoint of a two point holding, used
to define the end of the outbound leg.</gml:description>
          </appinfo>
        </annotation>
      </element>
    </choice>
   ............
  </sequence>
</group>
```

4.9 Mapping Relationships to Objects

Relationships are encoded by creating an XML element with the same name as the role name on the UML model. It is of type *Object*PropertyType.

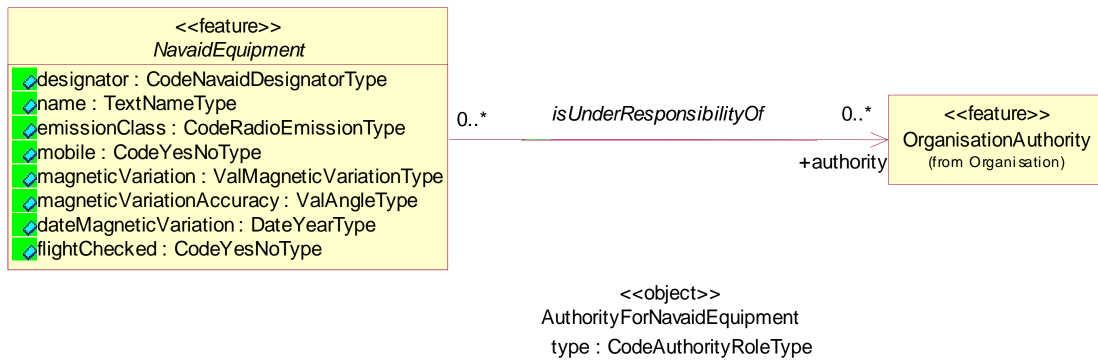
In this example, the SurfaceCharacterisitcs object is a property of the Runway. The "surfaceProperties" property of the Runway is defined as being of type SurfaceCharacteristicsPropertyType.



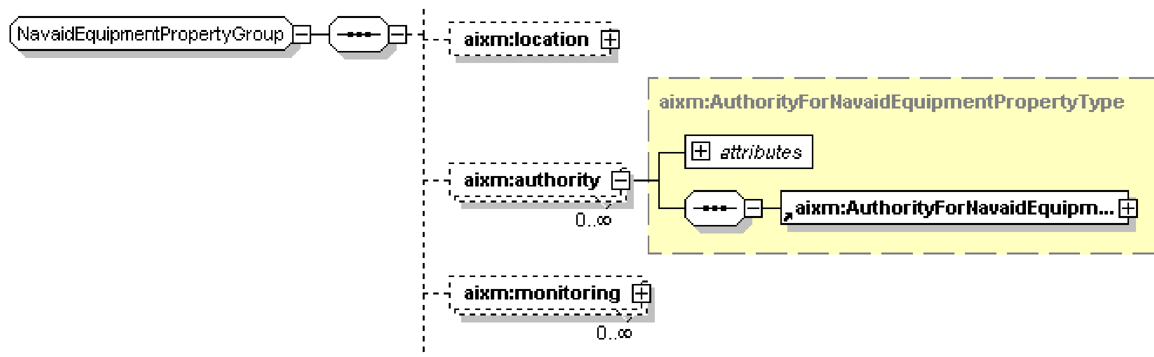

```
<element name="surfaceProperties"
type="aixm:SurfaceCharacteristicsPropertyType" minOccurs="0"/>
```

4.9.1 Mapping Associations with Association Classes

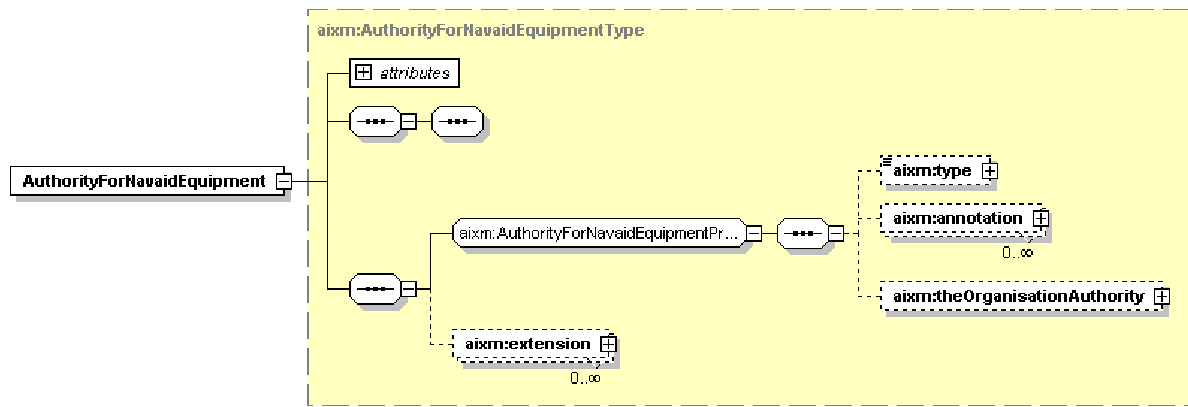
In the UML below, the NavaidEquipment feature has a relationship to the OrganisationAuthority feature. This relationship contains properties defined in the AuthorityForNavaidEquipment class.



When mapping this in XSD, an ‘authorityForNavaidEquipment’ property is created in the NavaidEquipment feature as shown below. The name of this property is automatically derived from the name of the association class, by conversion to lowerCamelCase style. The direction of the arrow is important. If the direction would have been to the NavaidEquipment, the property would have been created in the OrganisationAuthority feature.

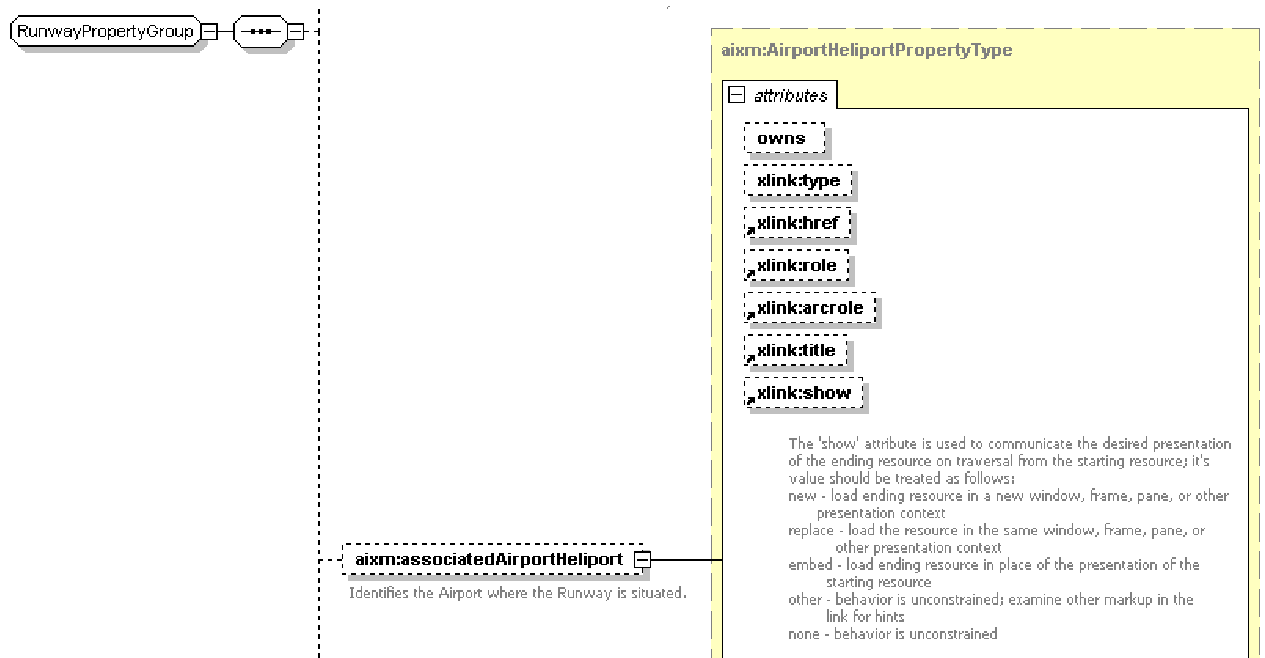


A second step is then required to complete the XSD. In this case an element named ‘theOrganisationAuthority’ is added in the definition of the AuthorityForNavaidEquipmentPropertyGroup, based on the role of the OrganisationAuthority class in this association.



4.10 Mapping Relationships to Features

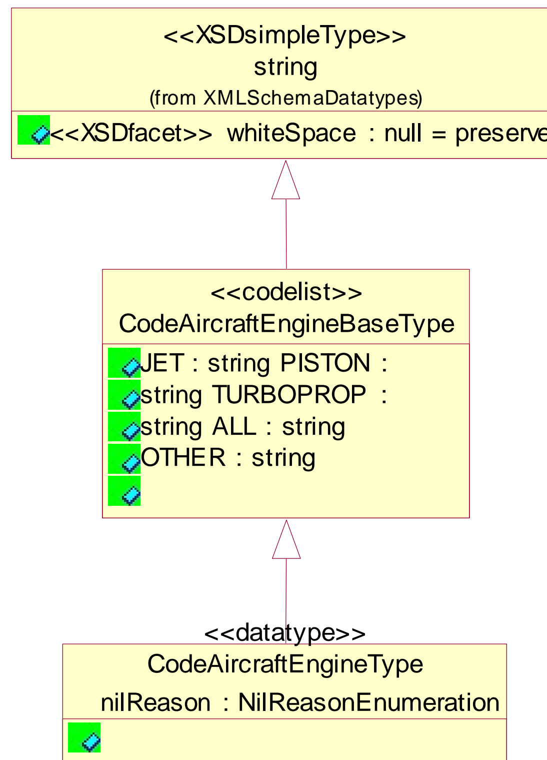
In AIXM, Relationships to features are described by reference using `xlink:href`. The UML role name is used for the XML element name and the XML element is of type *Feature*PropertyType.



4.11 Mapping Data Types

4.11.1 <<codelist>>

Codelists are given by the stereotype `<<codelist>>`. As it can be seen from the diagram elow, for each <<codelist>> type, there also is a <<datatype>> class, which defines the nilReason attribute.



First, the <<codelist>> class is converted into a simpleType in the XSD:

```
<simpleType name="CodeAircraftEngineBaseType">
   <annotation>
     <appinfo><gml:description>A code indicating the type of aircraft engine
(for example, jet, piston, turbo).</gml:description></appinfo>
   </annotation>
   <union>
     <simpleType>
       <restriction base="xsd:string">
         <enumeration value="JET">
           <annotation>
             <appinfo><gml:description>Jet Engine</gml:description></appinfo>
           </annotation>
         </enumeration>
         <enumeration value="PISTON">
           <annotation>
             <appinfo><gml:description>Piston
Engine</gml:description></appinfo>
           </annotation>
         </enumeration>
         <enumeration value="TURBOPROP">
           <annotation>
             <appinfo><gml:description>Turbo Propeller
Engine</gml:description></appinfo>
           </annotation>
         </enumeration>
```

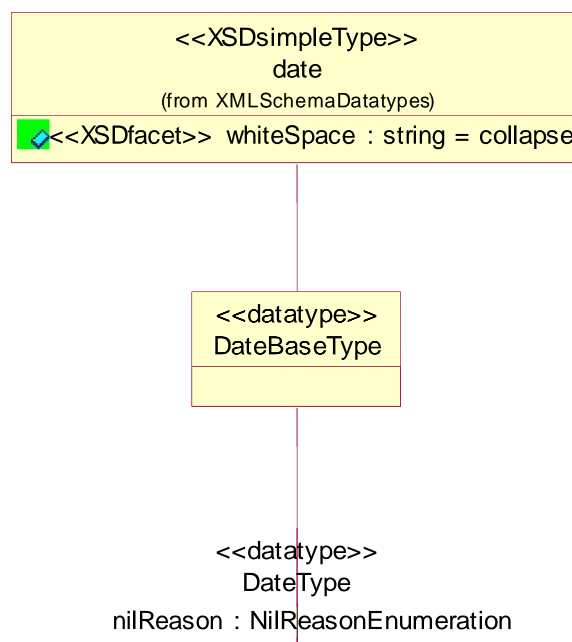
```
        <enumeration value="ALL">
          <annotation>
            <appinfo><gml:description>All aircraft engine
types.</gml:description></appinfo>
          </annotation>
        </enumeration>
      </restriction>
    </simpleType>
    <simpleType>
      <restriction base="string">
        <pattern value="OTHER(:(\w|_){1,58})?"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

Note that the simple data types is declared as a union between the enumerated values declared in the UML model (with the exception of the value "OTHER") and a string with the pattern "OTHER(:(\w|_){1,58})?". This enables <<codelist>> data types to include values that are not supported by the enumeration list. For example, an electric engine type could be encoded as "OTHER:ELECTRIC".

In addition, a complex type is defined, including the declaration of the nilReason attribute:

```
<complexType name="CodeAircraftEngineType">
  <simpleContent>
    <extension base="aixm:CodeAircraftEngineBaseType">
      <attribute name="nilReason" type="gml:NilReasonEnumeration"/>
    </extension>
  </simpleContent>
</complexType>
```

4.11.2 <<datatype>> - default case



As for <<codeList>>, the mapping of <<datatype>> used to type simple properties (see 2.7.1.1) consists of two steps.

The first step is the creation of the simpleType corresponding to the BaseType.

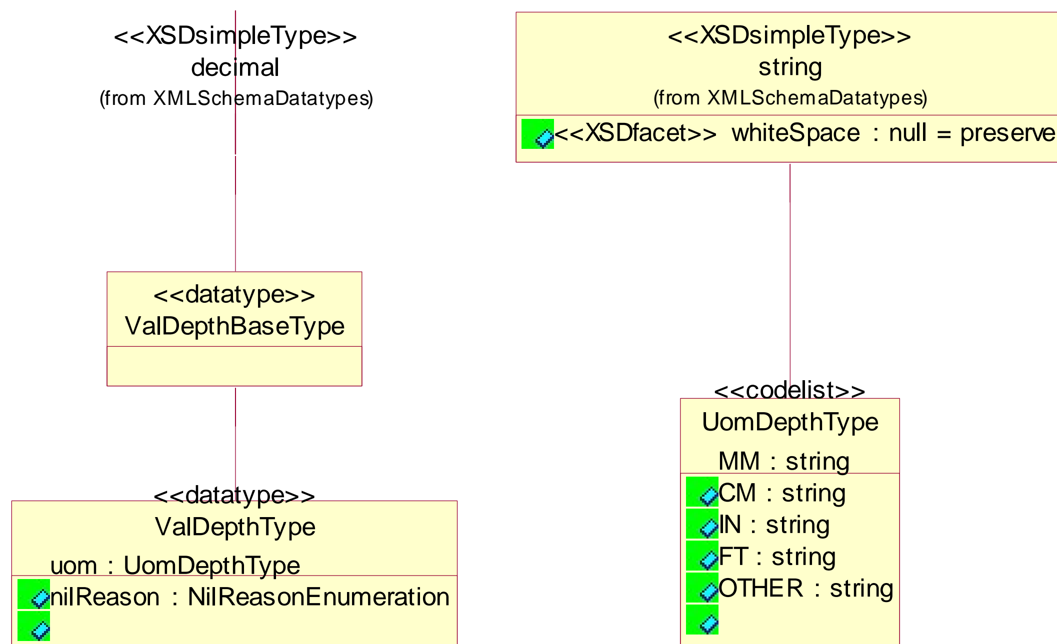
```
<simpleType name="DateBaseType">
  <restriction base="xsd:date">
  </restriction>
</simpleType>
```

The second step is the creation of the complexType which defines the attribute nilReason.

```
  <complexType name="DateType">
    <simpleContent>
      <extension base="aixm:DateBaseType">
        <attribute name="nilReason" type="gml:NilReasonEnumeration "/>
      </extension>
    </simpleContent>
  </complexType>
```

4.11.3 <<datatype>> with Unit of Measurement

A Unit of measurement (UOM) exists for many data types that take numerical values. This has been modelled as a uom attribute in the <<datatype>> class.



The XSD mapping of uom types follows the same rules as for any other <<codelist>>, except that no complex type is required with the nilReason.

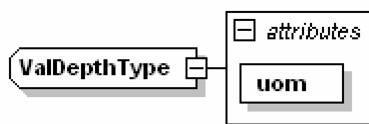
```
  <simpleType name="UomDepthType">
    <union>
      <simpleType>
        <restriction base="xsd:string">
          <enumeration value="MM">
          </enumeration>
          <enumeration value="CM">
          </enumeration>
          <enumeration value="IN">
          </enumeration>
          <enumeration value="FT">
          </enumeration>
        </restriction>
      </simpleType>
      <simpleType>
```

```
      <restriction base="string">
        <pattern value="OTHER:\w{2,58}"/>
      </restriction>
    </simpleType>
  </union>
</simpleType>
```

In a second step, the class ValDepthBaseType is generated as a simple type, as described in 4.11.2.

```
<simpleType name="ValDepthBaseType">
  <restriction base="xsd:decimal"/>
</simpleType>
```

Then, the uom attribute is added to the complexType ValDepthType, after the definition of nilReason attribute.



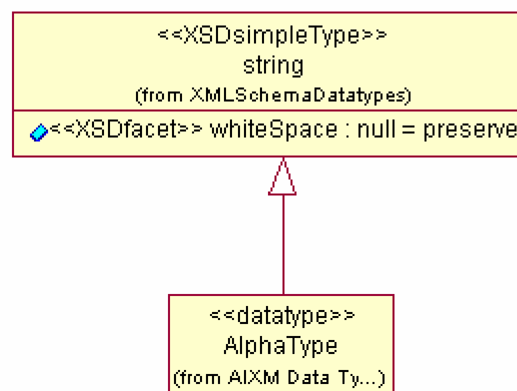

```
<complexType name="ValDepthType">
  <simpleContent>
    <extension base="aixm:ValDepthBaseType">
      <attribute name="nilReason" type="gml:NilReasonEnumeration"/>
      <attribute name="uom" type="aixm:UomDepthType" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

4.11.4 Particular cases

4.11.4.1 <<datatype>> with no BaseType

The 5 data types listed in 2.7.1.1 map directly to the built-in datatypes defined by the XML schema specification. The default datatypes are `string`, `float`, `double`, etc, which are considered `simpleTypes`.

The AlphaType acts as a convenient example.




```
<simpleType name="AlphaType">
 <restriction base="xsd:string">
  <pattern value="[A-Z]*"/>
 </restriction>
</simpleType>
```

4.11.4.2 <<datatype>> XHTMLBaseType

<<datatype>> XHTMLBaseType represents a structured XHTML document compliant with http://www.w3.org/1999/xhtml. It should be mapped as follows in XML:

```
<complexType name="XHTMLBaseType">
  <sequence>
    <any namespace="http://www.w3.org/1999/xhtml" minOccurs="1"
maxOccurs="unbounded" processContents="skip"/>
  </sequence>
</complexType>
```

APPENDIX D

AIXM

AIXM APPLICATION SCHEMA GENERATION

AIXM

AIXM Application Schema Generation

Aeronautical Information Exchange Model (AIXM)

***Copyright**: 2010 - EUROCONTROL and Federal Aviation Administration*

All rights reserved.

This document and/or its content can be download, printed and copied in whole or in part, provided that the above copyright notice and this condition is retained for each such copy.

For all inquiries, please contact:

Brett BRUNK - brett.brunk@faa.gov

Eduard POROSNICU - eduard.porosnicu@eurocontrol.int

Edition No.	Issue Date	Author	Reason for Change
0.1	2007/12/20	Barb Cordell / Paul Heffley	First Edition
0.2	2008/01/08	Barb Cordell / Paul Heffley	Updated version
1.0	2008/03/10	Eddy Porosnicu	First public version Editorial modifications
1.1	2010/02/04	Hubert Lepori	Updated version - AIXM 5.1

CONTENTS

1 SCOPE 1

1.1 Introduction 1

1.2 Background 1

1.3 Objective 3

1.4 References 3

2 EXTENDING AIXM FEATURES / OBJECTS 4

2.1 UML Package for Extensions 4
2.1.1 Package Structure 4
2.1.2 Package Specifications and Namespaces 4

2.2 UML Extension Package 5
2.2.1 Overview 5
2.2.2 XML Schema Generation 6
2.2.2.1 Imported and Included Schemas 7
2.2.2.2 Executing the Script 8
2.2.2.3 XML Schema Output 9

2.3 Data Type Extension Package 11
2.3.1 Overview 11
2.3.2 XML Schema Generation 13
2.3.2.1 Imported and Included Schemas 13
2.3.2.2 Executing the Script 14

2.4 UML Message Package 15
2.4.1 Overview 15
2.4.2 XML Schema Generation 16
2.4.2.1 Imported and Included Schemas 16
2.4.2.2 Executing the Script 16
2.4.2.3 XML Schema Output 16

1 Scope

1.1 Introduction

The purpose of this document is to describe how the AIXM UML model can be extended to support the needs of a particular Community of Interest (COI):

- Define messages that are necessary and eventually restrict the content of these messages to a sub-set of the AIXM features;
- Extend existing AIXM features with new attributes or associations or define new features, which are only relevant for that community.

The UML to XML Schema conversion for extensions is illustrated using a series of examples from the AIXM 5 Application Schema extensions.

1.2 Background

The AIXM conceptual model and data standard are maintained as a UML model. AIXM was developed to be extendable allowing greater flexibility for international use. Each feature and codelist may be extended to meet individual needs of the AIXM Community of Interest (COI).

If you are not familiar with the AIXM UML to AIXM XSD mapping document please review before reading this document. A good understanding of the mapping document will help to understand AIXM extensions.

Features describe real world entities and are fundamental in AIXM. AIXM features can be concrete and tangible, or abstract and conceptual and can change in time [7]. Features are represented as classes with a stereotype `<<feature>>`. Examples include Runway and AirportHeliport.

AIXM features are dynamic features. Timeslice objects are used to describe the changes that affect the AIXM feature over time. Timeslice objects and temporality are discussed extensively in a separate AIXM Temporality document.

Objects are abstractions of real world entities or, more frequently, of properties of these entities, which do not exist outside of a feature. An object is created for two reasons in AIXM:

- When a property has a multiplicity greater than one (such as the city served by an AirportHeliport), or
- The object has its own attributes that are reused throughout the model, such as ElevatedPoint.

Some classes are marked as `<<choice>>`. These are used to model XOR relationships. For example, an AirspaceVolume's horizontal projection can be a Surface, an AirspaceCorridor or the same shape as for another Airspace.

Properties are the attributes and relationships that characterise a feature or object. In the UML:

- Attributes are used to describe simple properties of a feature or object;
- Relationships are used to describe associations to features or objects. Whenever a property has a multiplicity greater than one, it is described using a UML relationship with cardinality.

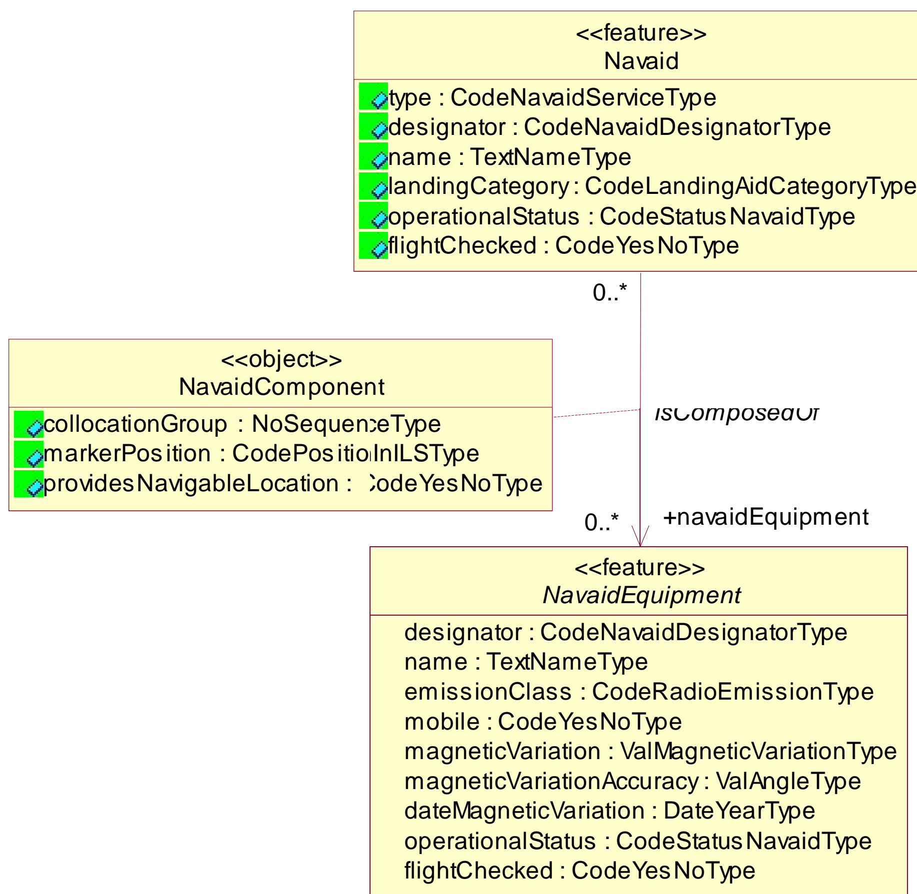
- Relationships to objects are depicted by the standard UML composition (*aggregation by value)* association. Composition is a form of aggregation with strong ownership and coincident lifetime of the parts by the whole. The part is removed when the whole is removed
- Relationships to features are described with a standard UML association. All of the associations are navigable in only one direction. This shows that the two classes are related but only one class knows that the relationship exists

The UML model lists the datatypes that are used throughout the AIXM. These are given one of the two following stereotypes:

<<datatype>> - This is basic data type that specifies a pattern to use.

<<codelist>> - This is a data type which codes a predefined list of values. The <<codelist>> includes the value OTHER which can be expanded with some free text in uppercase (“OTHER:*MY_VALUE*”).to support un-supported values.

When information about a relationship is required, a UML association class is used. The association class is attached to the relationship with an Association Class line.



Inheritance refers to the ability of one class (the specialized or child class) to inherit the properties of another class (the generalized or parent class), and then add new properties of its own. In AIXM, Features must only inherit from other Features and Objects must only inherit from other Objects. Multiple inheritance is not allowed.

Important Note**: Inheritance is supported only from “Abstract” classes**. The UML to XSD scripts do not support the inheritance from non-Abstract classes. The “extension”

mechanism, explained in this document, gives the possibility to extend an existing AIXM class with new properties, with the advantage that the extended class remains valid against the core AIXM schema.

1.3 Objective

The core AIXM model provides the definition of standardised aeronautical information features. In order to use AIXM for a specific application, a Community of Interest (COI) will have to agree upon how instances of AIXM features are to be exchanged and communicated in the community. This can be accomplished by either using a pre-defined Web Service (such as WFS), which enables the direct provision of individual AIXM features or collections of AIXM features or by defining custom AIXM messages with specific properties and encompassing a selected list of AIXM features.

In the definition of the AIXM Application Schema, the COI might also want to extend the core AIXM with additional properties and features. Some principles that regulate such extensions include:

- An extension of an existing AIXM feature should remain valid against the definition of the core AIXM XSD element with the same name (for that purpose, the AbstractSomeFeatureExtension element is provided in the core AIXM XSD). A consequence is that it is not possible to extend <<datatype>> classes. Only <<codelist>> may be extended.
- An additional feature and objects shall follow the core AIXM modelling conventions (stereotypes, naming, data types, etc.)

Important Note: It is under the responsibility of the COI to ensure that the extensions do not duplicate features and properties that already exist in the core model. When such extensions are defined, the COI might want to share it with the global AIXM community. For this purpose, the application schema can be made available through www.aixm.aero.

1.4 References

1. Geographic Information – Spatial Schema. ISO 19107. First Edition, 2003-05-01
2. ISO 19136:2007 - Geographic information -- Geography Markup Language (GML)
3. UML 2.0 In a Nutshell. Dan Pilone. O’Reilly Media Inc. 2005.
4. AIXM UML to XML Schema Mapping, www.aixm.aero (see Downloads)
5. AIXM Temporality Model, www.aixm.aero (see Downloads)

2 Extending AIXM Features / Objects

2.1 UML Package for Extensions

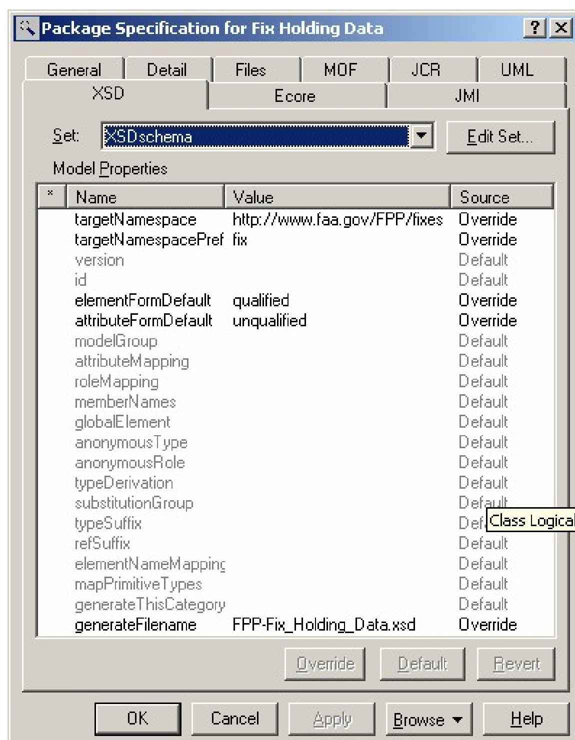
To extend AIXM, a new package must be created under the AIXM Application Schemas package. This package will contain all the information you need for your extension.

2.1.1 Package Structure

Different types of sub-packages are used to control the generation of appropriate XML schemas (XSD). The Extension sub-package contains the extensions to AIXM Core features and objects. If the extension requires new data types, then a second sub-package, the Extension Data Types, is created containing any new data types and codelists needed. The final sub-packages that are needed are the message packages. Multiple packages may be required based on the number of different message schemas needed. Most Application Schema Packages will have at least one request package and one response package.

2.1.2 Package Specifications and Namespaces

The extension package must have the appropriate XSD tool attributes set so the script can generate the namespaces correctly. Below is an example of how these attributes are set for the Fix Holding Data sub-package.



There are five properties that are needed for each new Application Schema package being used to generate XML Schemas. These properties are highlighted below with the Source as 'Override'.

To modify the value of these properties, open the Package Specification and navigate to the XSD tab. The targetNamespace and targetNamespacePrefix property values are determined by the COI and used in accordance with other related schemas and will determine if an external import is included or imported.

Additionally, denote the generateFilename property as applicable so the schema is named consistently each time it is generated with the UML to XSD scripts.

The following was generated for the Fix Holding Data Application Schema package.

<schema xmlns:fix="**http://www.faa.gov/FPP/fixes**"
xmlns="**http://www.w3.org/2001/XMLSchema**"
xmlns:xsd="**http://www.w3.org/2001/XMLSchema**"
xmlns:gml="**http://www.opengis.net/gml/3.2**"
xmlns:xlink="**http://www.w3.org/1999/xlink**"
xmlns:aixm="**http://www.aixm.aero/schema/5.1**"
xmlns:dpshare="**http://www.faa.gov/avnis/shared**"
targetNamespace="**http://www.faa.gov/avnis/fixes**"
elementFormDefault="**qualified**"
attributeFormDefault="**unqualified**">

2.2 UML Extension Package

2.2.1 Overview

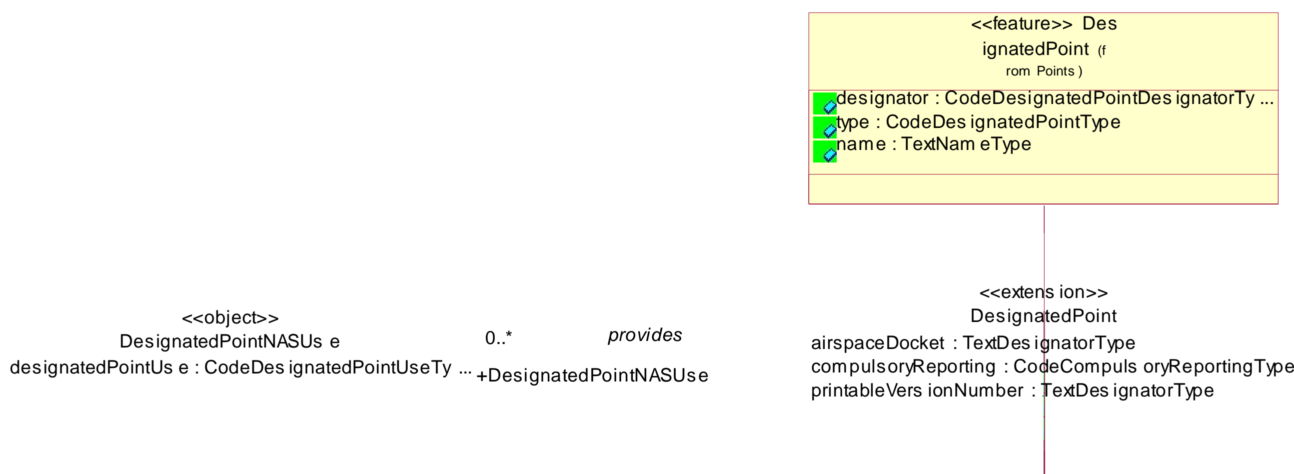
A feature or object may be extended by creating a class with the same name as the core AIXM feature and giving it a stereotype <<extension>>. This new class can contain:

New attributes

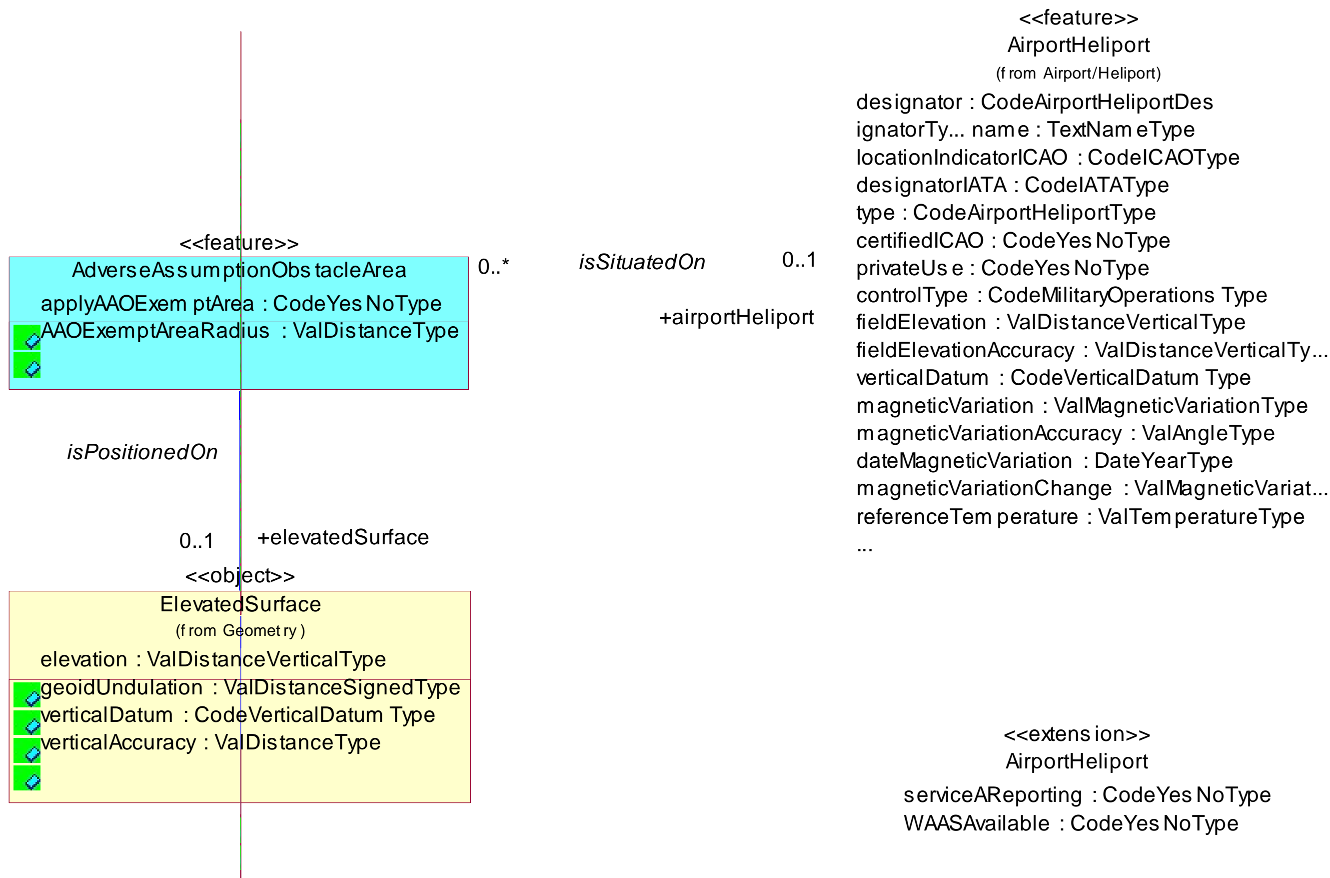
New associations. ***Important Note:*** if a core AIXM class is involved, the navigability of the association should always be from the <<extension>> class towards the core AIXM class.

In addition, it is possible in extensions to create declare totally new classes (features and objects), that do not extend existing AIXM Core classes. The only rule is to follow the AIXM UML modelling conventions described at the beginning of the document. This will enable the script AIXM-FeatureGenerator.ebs to correctly generate the XML elements for these new classes. This situation is not described in detail in this document because it does not require any special action. Just follow the UML modeling conventions.

The example below shows the modelling convention used to extend the DesignatedPoint feature. The example adds a new attribute to DesignatedPoint and a new relationship to a new object called DesignatedPointNASUse.



Associations can also be created between new features or objects and AIXM Core features or objects as depicted below in the association between AdverseAssumptionObstacle and AirportHeliport. The new association should be created in the Application Schema package and towards the AIXM Core Feature rather than an extension (if present). This action ensures the relationship is represented properly in the XML Schema



Use the approved rules for AIXM Core elements to produce new features or objects. Some rules that apply to new *extension* classes are:

The extension class stereotype must be <<extension>>.

The extension class name for the extension must match the class you are extending. (When using Rational Rose, it is possible to do that if you create a new class in the navigation menu at the left and change the name of that class; only afterwards drag and drop the class on the diagram.)

The extension class must be a specialized class extending the matching base class.

The extension class attributes are added to the extension class the same as they would a regular AIXM class (Data Types are discussed later in this document).

2.2.2 XML Schema Generation

Use the AIXM-FeatureGenerator.ebs script to generate the extension XML Schema in which the script triggers the generation of an extension element by recognizing the <<extension>> stereotype. Generation of the extension follows the AIXM generation rules.

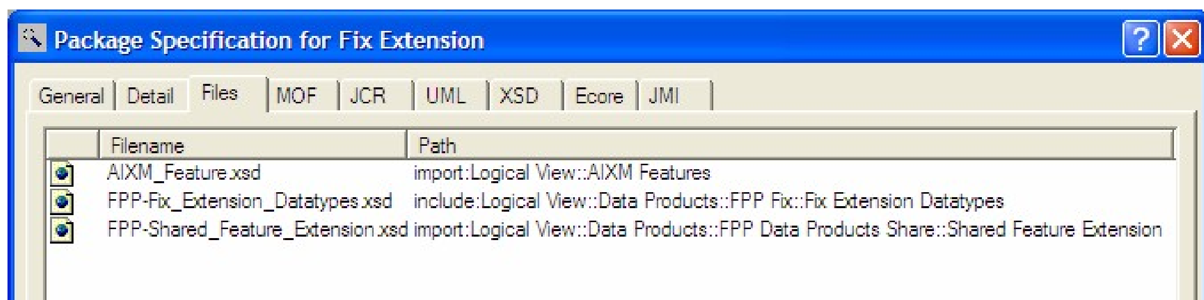
If new data types or codelists are introduced the script, AIXM-DataTypeGenerator.ebs, must be executed first on the associated Data Type Package.

2.2.2.1 Imported and Included Schemas

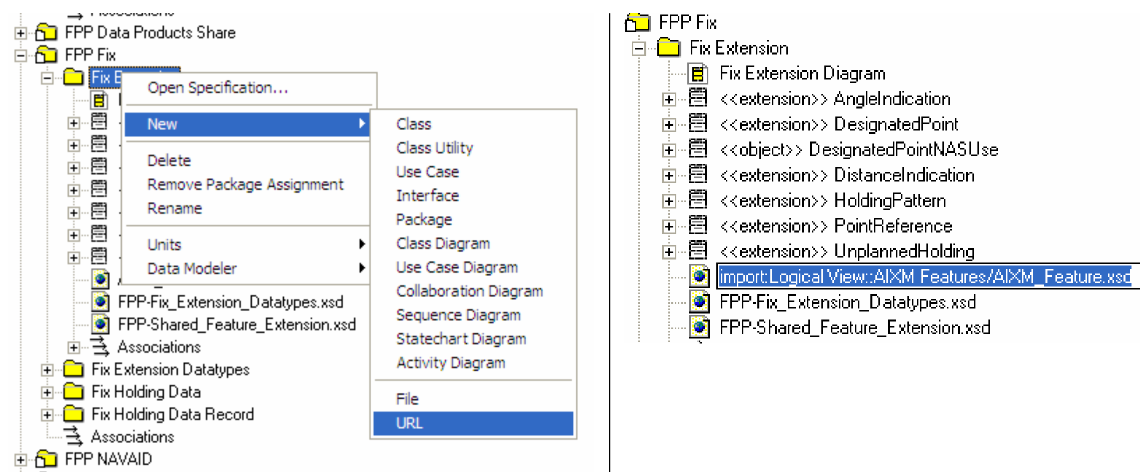
Every application schema sub-package must incorporate imported XML Schemas from the AIXM Core Schemas. Additionally, if new data types or codelists are introduced, the schema from that sub-package must be included. It is sometimes required to utilize common objects from a 'shared' package to increase object re-use. These elements should be incorporated as well by importing the schema.

It is not required for these schemas to be generated for the script to run in Rational Rose, but if they are not created and in the folder structure when the schema is opened it will have errors. The script, AIXM-DataTypeGenerator.ebs, is used to create the schema on the associated Data Type sub-package.

These linked schemas, essentially URL's, can be incorporated by entering them on the Files tab of the Package Specification.



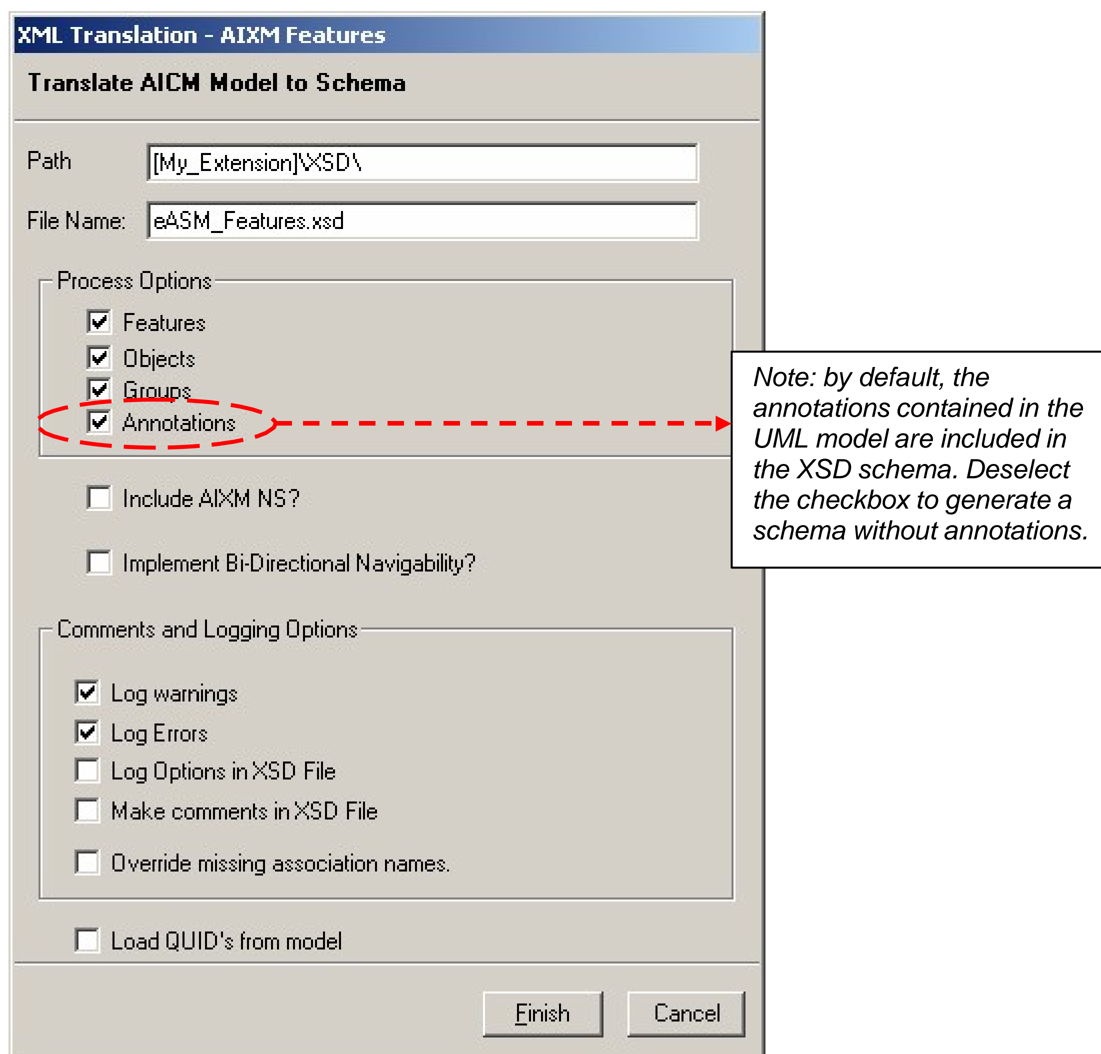
They can be added also be added in the navigation window by entering the full path of the schema location within the model.



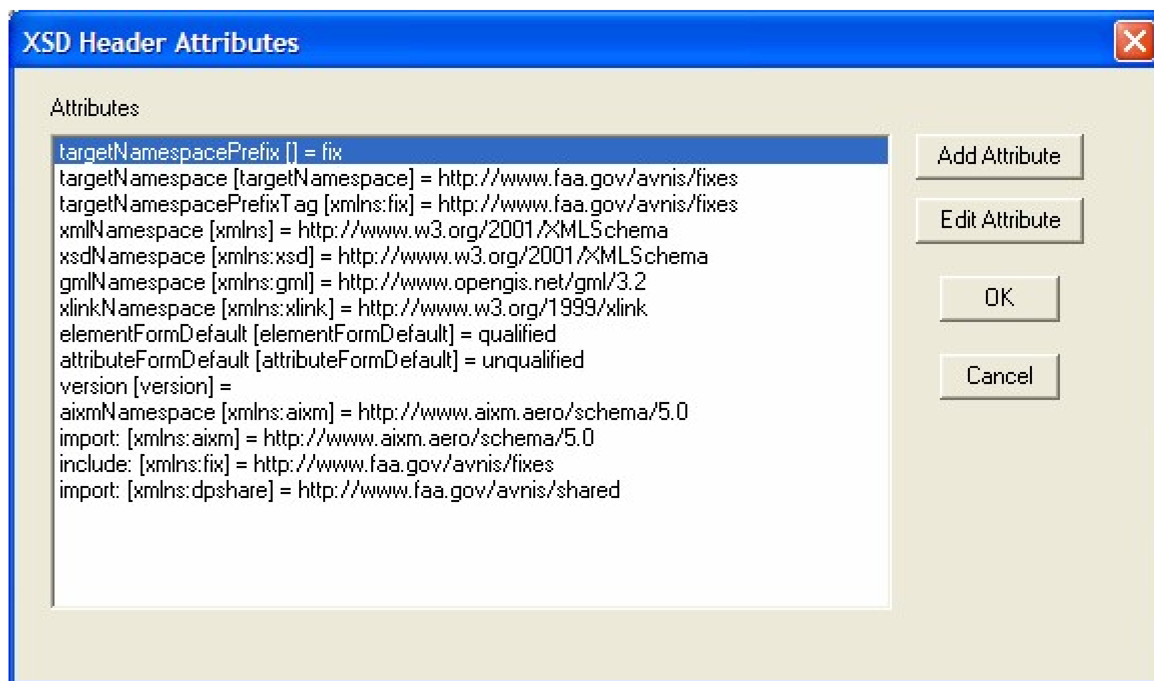
2.2.2.2 Executing the Script

There are some specific options that need to be set when executing the script in Rational Rose, but most of the options will be defaulted. In the image below, notice the File Name is pulled from the generateFilename property set in the package specification earlier.

The schema generation scripts are used for both AIXM Core and Application Schemas. The checkbox for 'Include AIXM NS' is selected when executing scripts for packages that are not part of the AIXM Core set since the AIXM Namespace is automatically included for those schemas. Furthermore, select the checkbox for 'Load QUID's from model' when new classes have been added to the model since the script was last run, which ensures the element identifiers are correctly recognized.



After selecting 'Finish' and regenerating the QUID's (if needed), a dialogue will open to allow the addition or editing of the XSD Header attributes. It should not be necessary to make any changes, but notice the targetNamespace attributes generated from the package specifications set earlier. After selecting 'OK', the XML schema file will be generated and can be found in the directory in which the script was run.

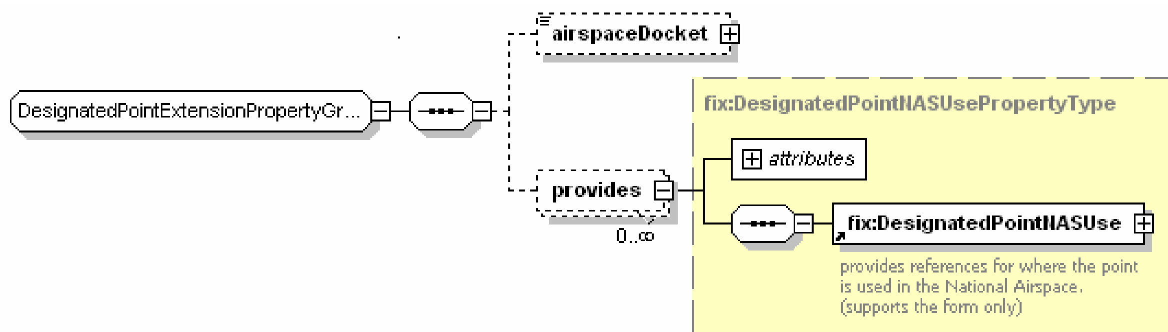


2.2.2.3 XML Schema Output

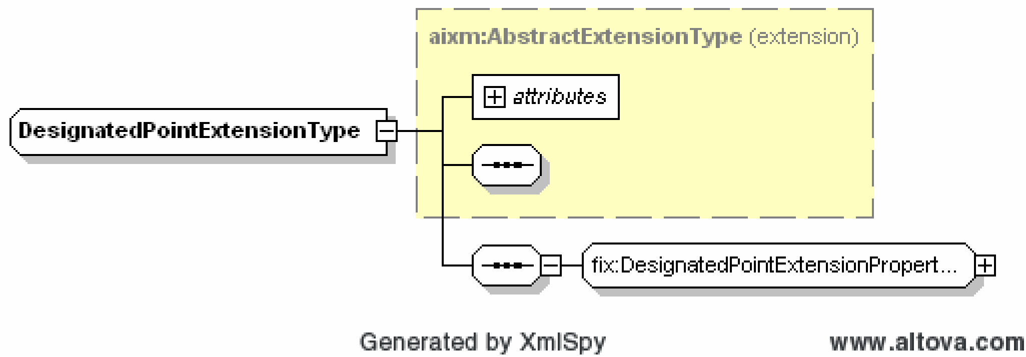
Classes with the stereotype of <<extension>> generate three related elements for that class.

<classname>ExtensionPropertyGroup
<classname>ExtensionType
<classname>Extension

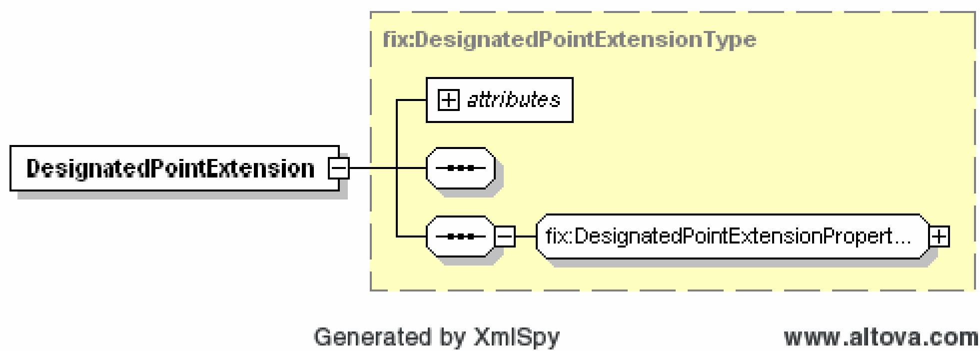
The <classname>ExtensionProperyGroup contains the properties (elements and relationships) of the Extension.



The <classname>ExtensionType element is generated as a XMLSchema <complexType.> and extends base type aixm:AbstractExtensionType.



The <classname>Extension element is generated as a XMLSchema <element>. The Extension element cannot stand alone, it may only exist as an extension to an AIXM base element. The Extension element does not have a timeslice. The Extension element attribute substitutionGroup is the substitutionGroup of the base type extension. Extension elements are not extensible.

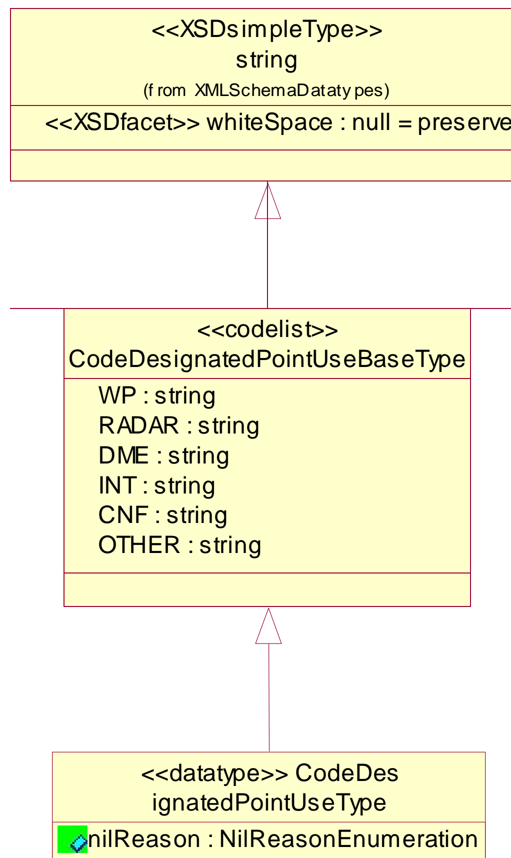


2.3 Data Type Extension Package

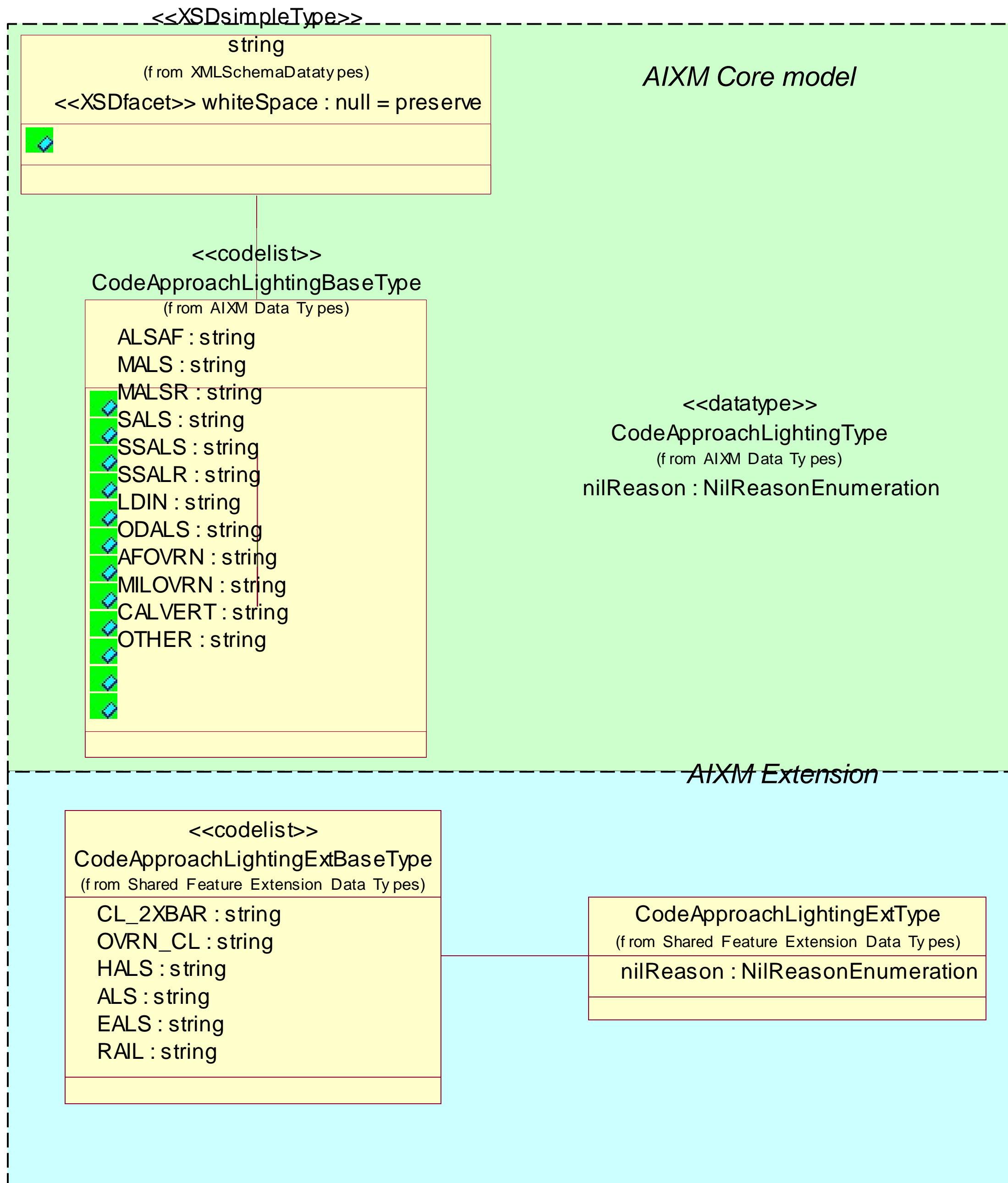
2.3.1 Overview

An extension, new feature or object class may require additional data types or codelists to capture the valid values for new attributes added to the class. To add new data types or codelists create a Data Type sub-package containing any new data types needed.

In the example below, an <<codelist>> is defined in an extension package. It is called CodeDesignatedPointUseBaseType, it has a generalization to the 'string' class and inherits the basic attributes of an XSD string variable. The <<datatype>> CodeDesignatedPointBaseType is created with the nilReason property, as specified in [4].This is the most common configuration for codelists.



AIXM Core <<codelists>> can also be extended in the Data Type sub-package. Extend a codelist by creating a class with the same name as the codelist and giving it a stereotype <<codelist>>.



Careful analysis must be done to ensure that the extended list of values remains normalised. It shall not duplicate values that already exist in the core <<codelist>> (including the OTHER value), but with other names/abbreviations.

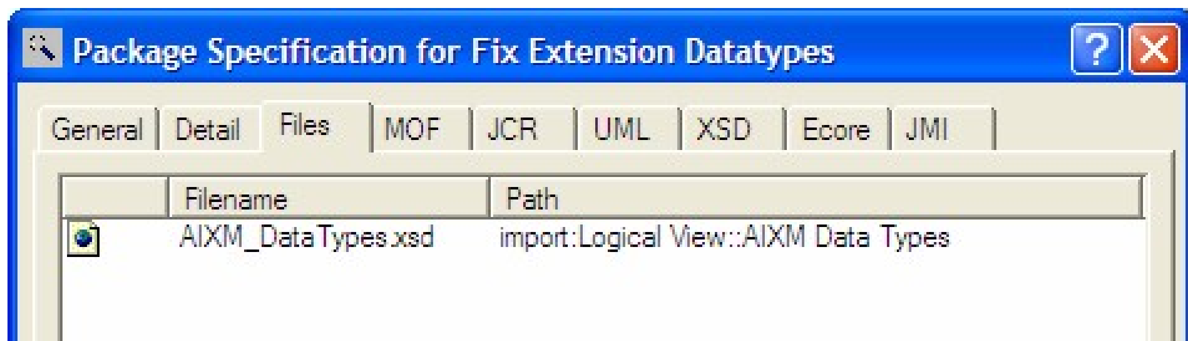
If additional values are required for an AIXM Core codelist for international data exchange, then AIXM Core model will need to be updated.

2.3.2 XML Schema Generation

Use the AIXM-DataTypeGenerator.ebs script to generate the data type extension XML Schema. Data Types are generated as a XMLSchema <simpleType> with the appropriate facets, Patterns and/or codelists defined.

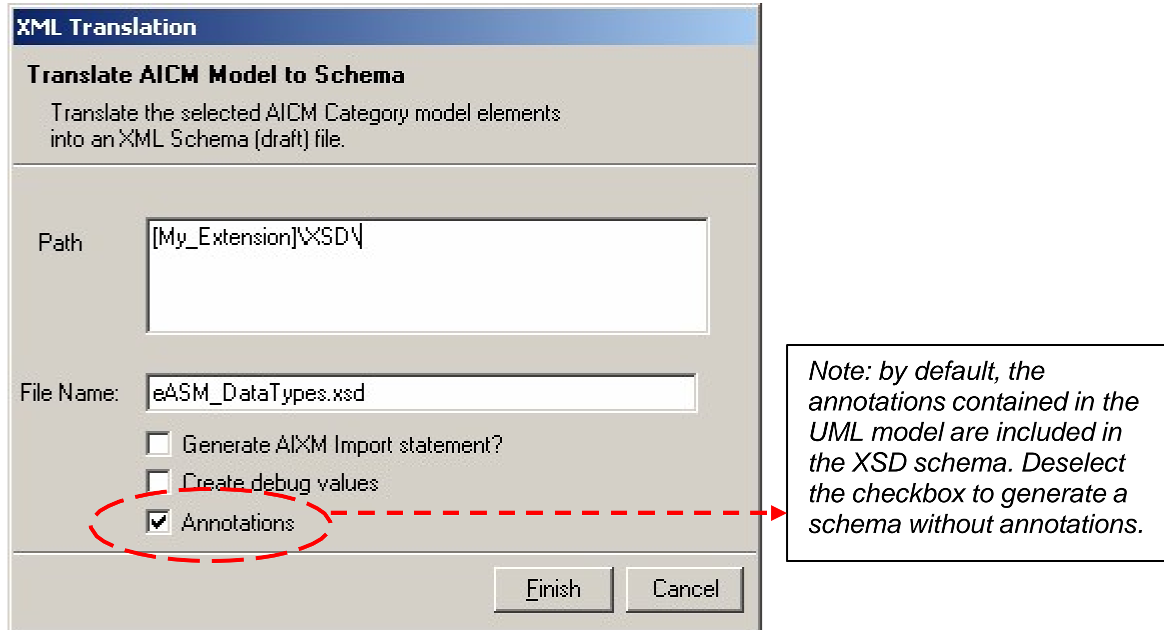
2.3.2.1 Imported and Included Schemas

Each data type sub-package must incorporate the AIXM Core Data Type schema. It is not required for this schema to be generated for the script to run in Rational Rose, but if the AIXM Core Data Type schema is not created and in the folder structure when the schema is opened it will have errors.



2.3.2.2 Executing the Script

In the image below, notice the File Name is pulled from the generateFilename property set in the package specification earlier. However the Path denotes the location of the UML model file, which should be changed appropriately. The checkbox for 'Include AIXM NS' is selected when executing scripts for packages that are not part of the AIXM Core set since the AIXM Namespace is automatically included for those schemas.

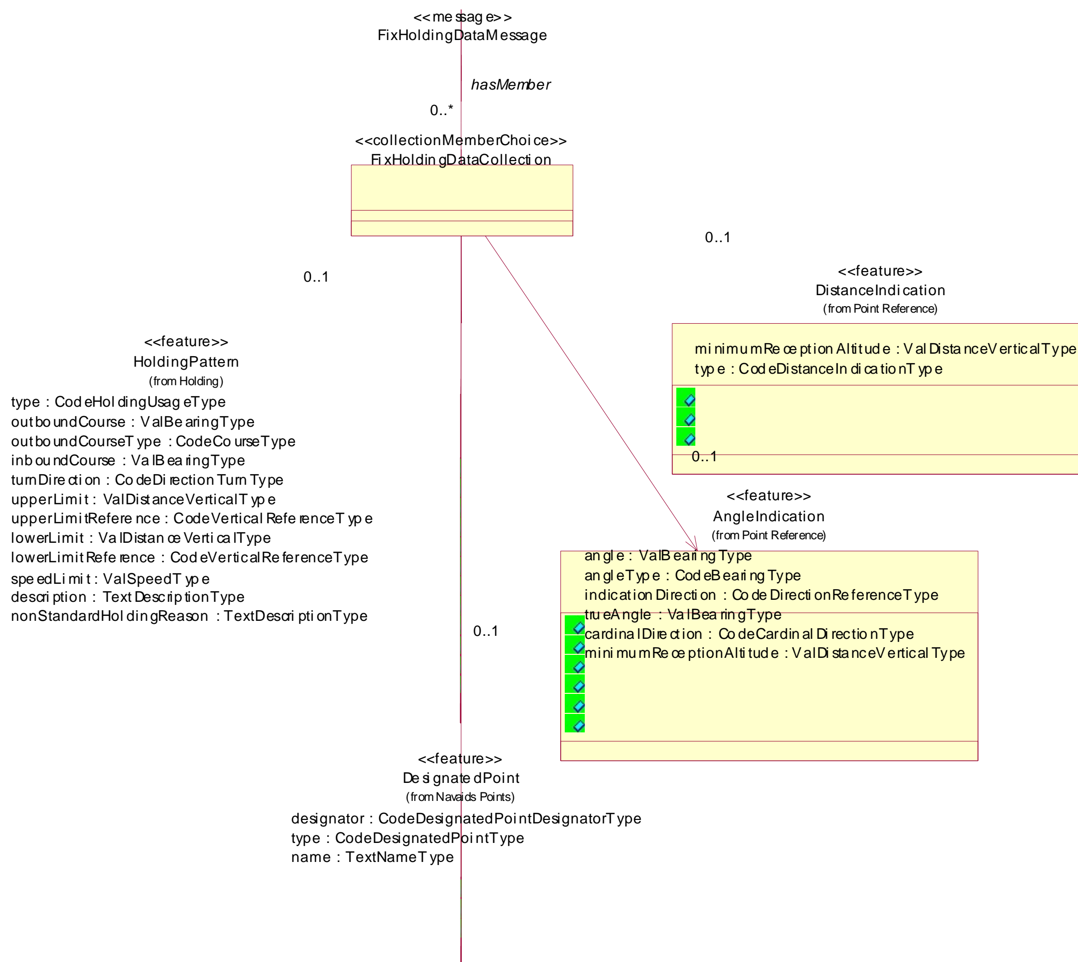


2.4 UML Message Package

2.4.1 Overview

The message package is used to generate an XML Schema for request and response messages. Below is an example of the FixHoldingData Response Message. This message includes the extensions described previously even though they do not appear in the diagram.

A Message is modelled in UML using the class object with a stereotype <<message>>. In this example the message is a limited collection of AIXM features with extensions. This is modelled by relating the collection of features; <<collectionMemberChoice>> to the message through the relationship “hasMember”.

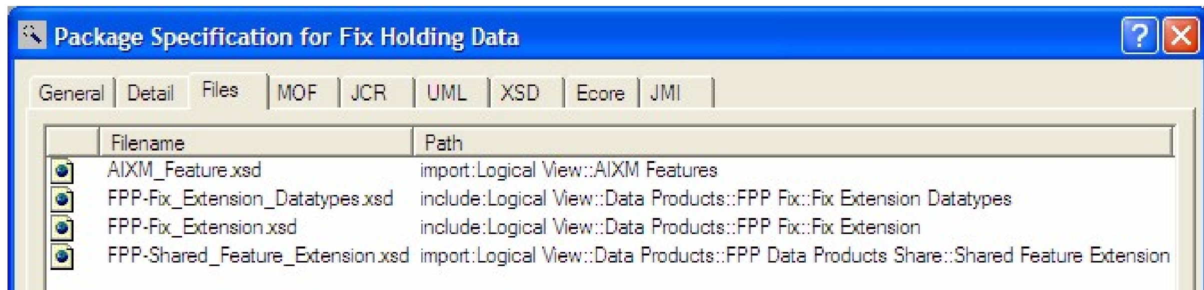


2.4.2 XML Schema Generation

Use the AIXM-ApplicationSchemaGenerator.ebs script to generate the message XSD. The script triggers the generation of the message element by recognizing the <<message>> stereotype.

2.4.2.1 Imported and Included Schemas

The UML Message sub-package brings together all of the related elements created in earlier processes such as extensions and data types. As before, the AIXM Core schema must be included as well as any other referenced schemas (i.e. shared or common objects that are to used in multiple application schemas).



The accumulation of the imported and included XML Schemas is displayed below.

```
<import namespace="http://www.opengis.net/gml/3.2"
  schemaLocation="./ISO_19136_Schemas/gml.xsd"/>
<import namespace="http://www.aixm.aero/schema/5.1"
  schemaLocation="./AIXM_Feature.xsd"/>
<import namespace="http://www.w3.org/1999/xlink" schemaLocation="./xlink/xlinks.xsd"/>
<import namespace="http://www.faa.gov/avnis/shared" schemaLocation="./FPP-
  Shared_Feature_Extension.xsd"/>
<include schemaLocation="FPP-Fix_Extension_Data_Types.xsd"/>
<include schemaLocation="FPP-Fix_Extension.xsd"/>
```

2.4.2.2 Executing the Script

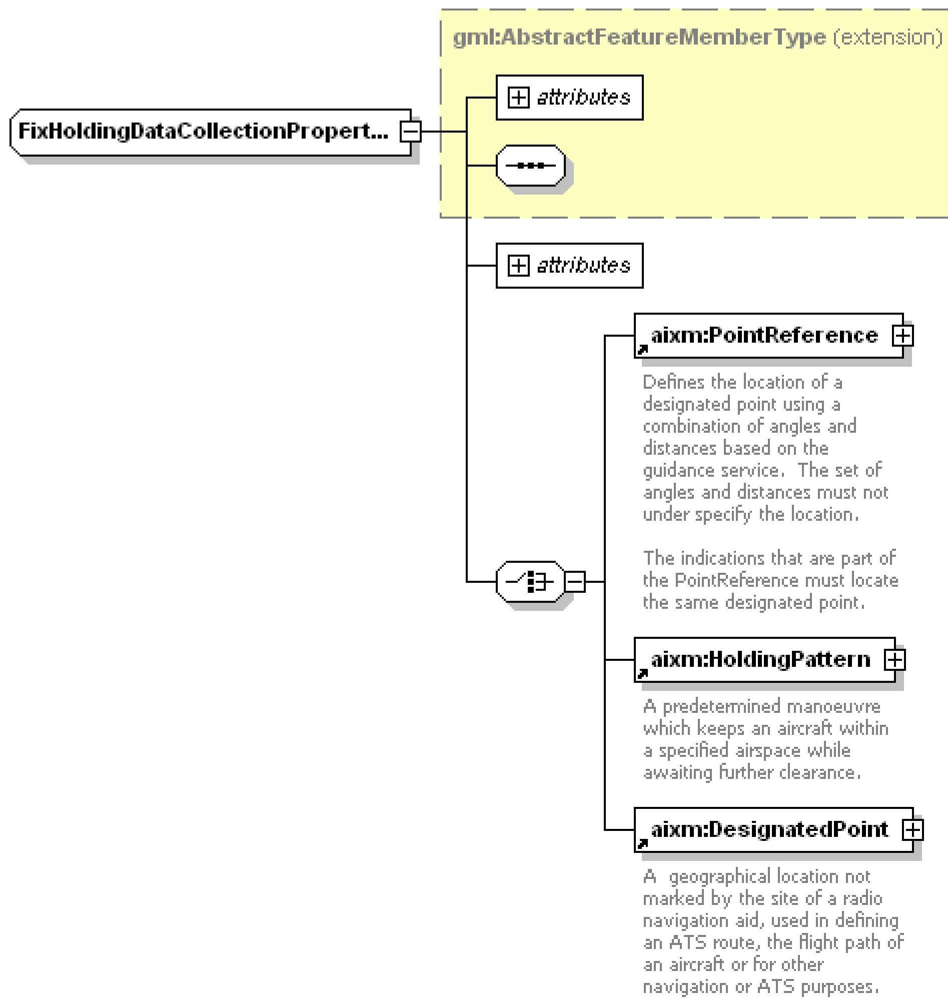
Follow the procedures outlined in section 2.2.2.2.

2.4.2.3 XML Schema Output

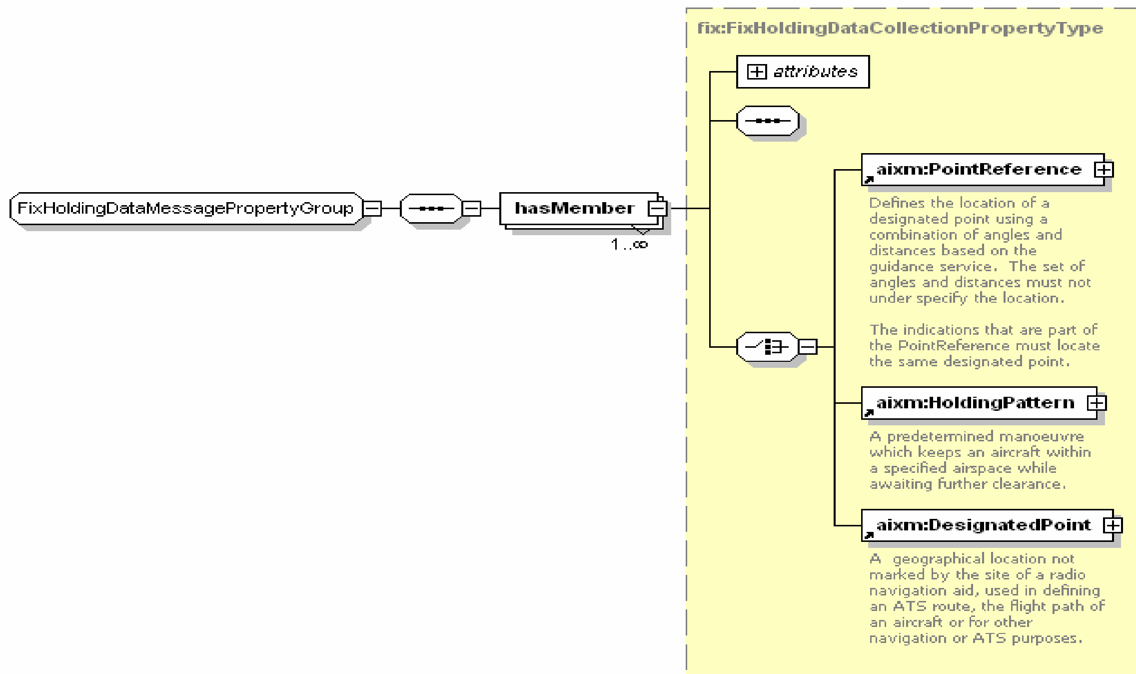
Classes with the stereotype of <<message>> following the AIXM feature collection response generates four related elements for that class.

<classname>CollectionPropertyGroup
<classname>MessagePropertyGroup
<classname>MessageType
<classname>Message

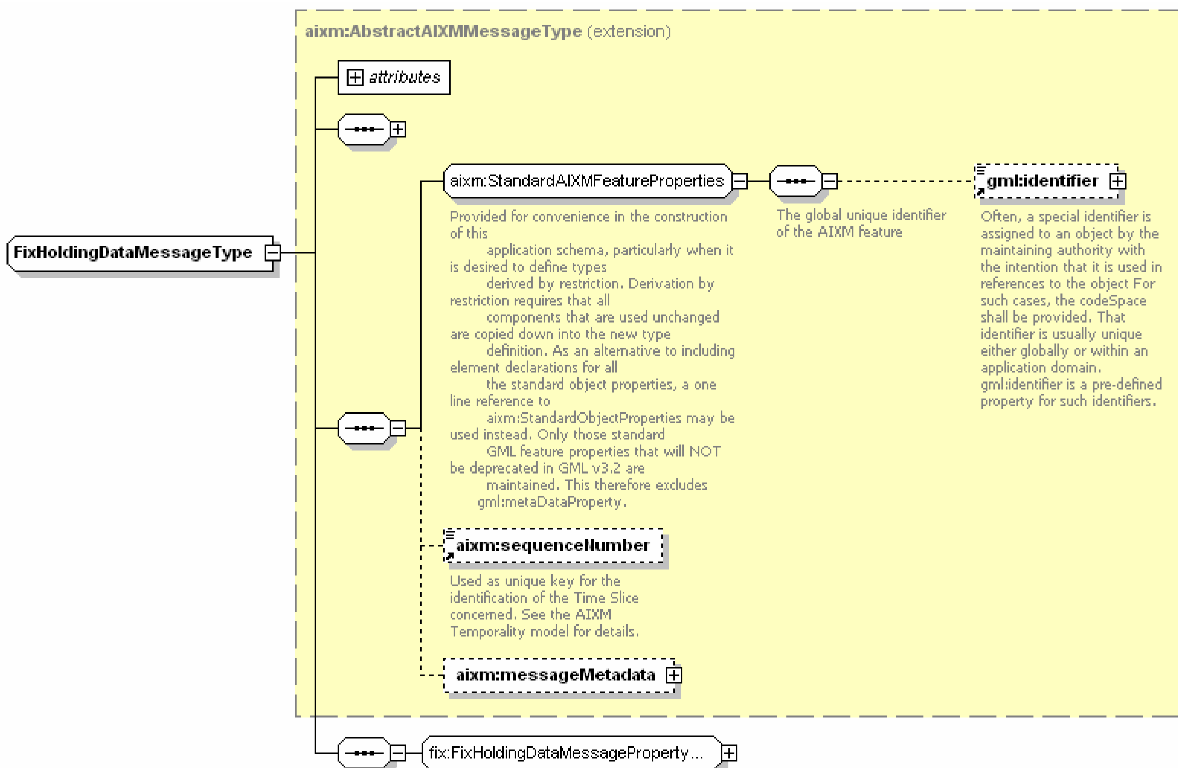
The <classname>CollectionPropertyGroup is generated as a XMLSchema <complexType>, which extends gml:AbstractFeatureMemberType, and includes a <choice> between the all the features it is pointing to.



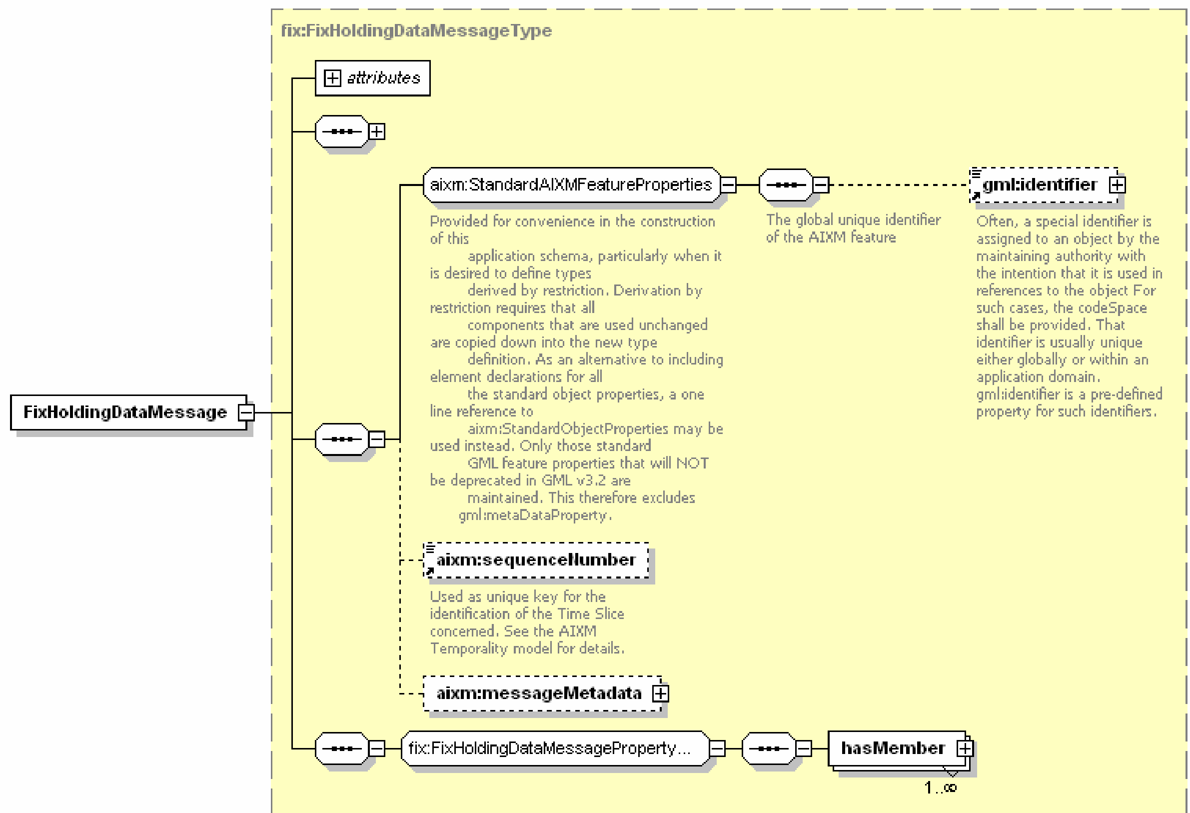
The <classname>MessagePropertyGroup is generated as a XMLSchema <group>, which contains the properties (elements and relationships) of the Message.



The <classname>MessageType element is generated as a XMLSchema <complexType.> and extends base type aixm:AbstractAIXMMessageType.



The <classname>Message element is generated as a XMLSchema <element>. The associations are treated as objects. They are included in the schema.



APPENDIX E

AIXM 5

FEATURE IDENTIFICATION AND REFERENCE

Use of xlink:href and UUID

AIXM 5

Feature Identification and Reference

- use of xlink:href and UUID -

Aeronautical Information Exchange Model (AIXM)

***Copyright**: 2011 - EUROCONTROL and Federal Aviation Administration*

All rights reserved.

This document and/or its content can be download, printed and copied in whole or in part, provided that the above copyright notice and this condition is retained for each such copy.

For all inquiries, please contact:

Deborah COWELL - deborah.cowell@faa.gov

Eduard POROSNICU - eduard.porosnicu@eurocontrol.int

<table>
<tr><th colspan="2">Edition No.</th><th>Edition Issue Date</th><th>Author</th><th>Reason for Change</th></tr>
<tr><td>0.1</td><td>First</td><td>2009</td><td>Design Team</td><td>First Edition</td></tr>
<tr><td>0.2</td><td>Proposed</td><td>2010</td><td>Design Team</td><td>Update</td></tr>
<tr><td>0.3</td><td>Proposed</td><td>06 Oct 2010</td><td>Design Team</td><td>Update Abstract references</td></tr>
<tr><td>0.4</td><td>Updated proposal</td><td>22 Nov 2010</td><td rowspan="4">Eurocontrol-FAA AIXM Design Team

with the contribution of AIXM Forum members</td><td>Included guidelines for UUID generation. Updated based on comments and recommendations received from the AIXM Forum members and during the AIXM-XML Seminars of 2010.</td></tr>
<tr><td>0.5</td><td>Updated Proposal</td><td>21 Dec 2010</td><td>Updated following latest comments on the AIXM Forum. Use of “uuid” URN. Use of URN only for abstract references.</td></tr>
<tr><td>0.6</td><td>Updated</td><td>21 FEB 2011</td><td>Updated following discussions off-line with the most active (on this subject) AIXM Forum members</td></tr>
<tr><td>1.0</td><td>Released</td><td>29 APR 2011</td><td>Final release. Added a sentence in section 2.2 stressing the importance for UUID to be allocated by the real authoritative source for a feature’s data.</td></tr>
</table>

Table of Contents

1 Scope 4
1.1 Introduction 4
1.2 References 4
1.3 Assumptions and Dependencies 4
2 Feature identification (UUID) 5
2.1 The gml:identifier property 5
2.2 Use of UUID 5
2.3 UUID version and codeSpace 6
2.4 The gml:id property 6
3 Feature Reference (xlink:href) 8
3.1 Introduction 8
3.2 Concrete local references within a message 8
3.3 Concrete external references 9
3.4 Abstract references 9
3.5 Use of xlink:title 11
A.1. UUID algorithms 13

1 Scope

1.1 Introduction

The Aeronautical Information Exchange Model (AIXM) is a GML 3.2 application schema meant to allow for the machine-to-machine exchange of aeronautical information in a structured format. As services that disseminate information in AIXM 5.1 to consumers are developed, the ability to manage the linkages between aeronautical features is key. This encompasses the concepts of feature identification and feature reference.

The AIXM 5.1 schema uses the XLink schema bundled with GML 3.2 for representing a reference between two features. In concert with the XLink standard, the XPointer standard may be used to address individual XML elements of messages. This document defines a number of standard cases for how XLinks should be used within an AIXM 5.1 message and how those XLinks should be resolved by applications.

The AIXM 5.1 schema also relies on the use of universal unique identifiers (UUID) as artificial identifiers for AIXM features. In fact, they do not identify the feature itself, but the data that represents that feature in digital aeronautical information management systems. This document provides guidance with regard to the algorithms that can be used for the generation of these UUID values.

1.2 References

[XLINK]	XLink v1.0 Specification http://www.w3.org/TR/xlink
[XPTR]	XPointer Specification http://www.w3.org/TR/xptr
[XPTH]	XPath Specification http://www.w3.org/TR/xpath
[UUID]	Universally unique identifier (theory) http://en.wikipedia.org/wiki/Universally_unique_identifier
[UUID-AIXM]	Universally Unique Identifier (UUID) Analysis, by Robert DeBlanc, MITRE, in support to FAA
[UUID-ISO]	ISO/IEC 9834-8, which is also ITU-T Rec x.667 http://www.itu.int/ITU-T/studygroups/com17/oid.html

1.3 Assumptions and Dependencies

This document assumes that the systems in question are communicating using messages or data sets that comply with the AIXM 5.0, 5.1 or later schema version.

2 Feature identification (UUID)

2.1 The gml:identifier property

Each AIXM Feature is identified through the use of the ***identifier*** property, which is inherited from the abstract AIXMFeature. In the AIXM XML Schema, this is mapped to the gml:identifier property, which all AIXM features inherit from the gml:DynamicFeatureType.

According to the AIXM Temporality Concept, the *identifier* property is the only time-invariant property; therefore it is situated outside the TimeSlice complex object, which encapsulates all the feature properties that can change in time. The gml:identifier property can be transmitted along with any Feature TimeSlice, allowing to identify the feature to which the TimeSlice belongs.

There are two essential requirements for the identifier property:

1. ***to be unique*** – there should exist a reasonable confidence that an identifier will never be unintentionally used by anyone for anything else;
2. ***to be universal*** – the same identifier should be used in all systems to identify a given AIXM Feature.

2.2 Use of UUID

The first requirement can be satisfied through the use of Universal Unique Identifiers (UUID). UUID generation algorithms can guarantee that the risk for the same UUID value to be generated by another system, for another feature, is extremely low. Information about such algorithms is provided in Appendix 1 of this document.

Concerning the second requirement, it is important to note that the identifier does not identify a feature. It identifies the data that someone has about a feature! In order to get maximum benefit from UUID, they should be generated by the primary originator (authoritative source) for that feature data.

Ideally, all stakeholders should have the same data about a given feature. However, as multiple “pseudo-primary” information sources may exist for the same data item or because the digital data transmission chain may be broken or duplicated, this cannot be guaranteed, at least on short term. Ensuring that the same gml:identifier is used in all systems for a given AIXM feature is a requirement for the information management process; therefore, it needs to be taken care through the process rules. From this point of view, UUIDs can indicate the continuity and the coherence of the data chain. If two systems use the same UUID for a feature, this indicates that either:

- they have the data from the same source (could be one of the two system, or a third one), or
- there are processes in place that ensure the consistency of the data between the two systems.

It is therefore possible that two or more information sets (list of TimeSlices) exists for the same AIXM feature, in two different systems, with different gml:identifier values. When data from different sources is merged in a single system, the owner of that system might be confronted with the need to identify and merge duplicate feature data, based on actual properties of the feature, not on the gml:identifier.

Overall, the most important advantage of using UUID as gml:identifier in AIXM is for software development. It is much simpler and less error prone to write code that relies on

UUID for feature identification and reference, as compared the use any kind of "natural key" combination.

2.3 UUID version and codeSpace

On the basis of the analysis presented in Appendix 1, ***the use of version 4 UUID based on random number generation, is recommended for AIXM***. An example of a gml:identifier using a UUID value is provided below.

```
<gml:identifier
     codeSpace="urn:uuid:">a82b3fc9-4aa4-4e67-8def-
aaea1ac595j</gml:identifier>
```

Information about UUID generation capabilities in common software is provided in Appendix 1, section A.1.9.

Note that a Uniform Resource Name (URN) is used as codeSpace for the gml:identifier. The ISO/IEC 9834-8 or RFC 4122 (http://www.ietf.org/rfc/rfc4122.txt) provide the UUID codeSpace value "urn:uuid:".

2.4 The gml:id property

Every GML object is required to have a gml:id value, which is intended as a local unique identifier, within the XML data set. AIXM features also being GML objects, they must have a gml:id value as well. In addition, all other GML objects inside the feature (TimeSlice, gml:TimePeriod, gml:Point, aixm:SurfaceCharacteristics, aixm:AirspaceVolume, etc.) are also required to have a gml:id values, as shown in the example below:

```
<aixm:Airspace gml:id="…">
  <gml:identifier
     codeSpace="urn:uuid:">a82b3fc9-4aa4-4e67-8def-
aaea1ac595j</gml:identifier>
  <aixm:timeSlice>
    <aixm:AirspaceTimeSlice gml:id="…">
      <gml:validTime>
        <gml:TimePeriod gml:id="…">
          <gml:beginPosition>2010-06-29T17:31:00</gml:beginPosition>
          <gml:endPosition>2010-06-29T19:00:00</gml:endPosition>
        </gml:TimePeriod>
      </gml:validTime>
      <aixm:interpretation>BASELINE</aixm:interpretation>
      <aixm:sequenceNumber>1</aixm:sequenceNumber>
      <aixm:type>D</aixm:type>
      <aixm:geometryComponent>
        <aixm:AirspaceGeometryComponent gml:id="…">
          <aixm:theAirspaceVolume>
            <aixm:AirspaceVolume gml:id="…">
              <aixm:upperLimit uom="FT">500</aixm:upperLimit>
              <aixm:upperLimitReference>MSL</aixm:upperLimitReference>
              <aixm:lowerLimit uom="FT">GND</aixm:lowerLimit>
              <aixm:lowerLimitReference>MSL</aixm:lowerLimitReference>
              <aixm:horizontalProjection>
              <aixm:Surface gml:id="…">
                <gml:patches>
                  <gml:PolygonPatch>
                    <gml:exterior>
              ...
```

```
</aixm:Airspace>
```

The gml:id value has to comply with the same rules as any other XML ID attribute: to be unique within the XML file, to start with a letter, etc.

It is recommended that the gml:id of the AIXM features (such as aixm:Airspace, aixm:Runway, etc.) also make use of the UUID value, prefixed with “uuid.”. Since UUID are globally unique, they are also locally unique and thus perfect candidates for gml:id. Attention, this recommendation concerns only the AIXM Feature level, not the lower levels, such as aixm:AirspaceTimeSlice, etc. The use of the feature UUID as gml:id will facilitate the implementation of concrete xlink:href references using the direct ‘#ID’ syntax, as explained in 3.1.

Applied to the previous example, these recommendations give the following gml:id values:

```
<aixm:Airspace gml:id="uuid.a82b3fc9-4aa4-4e67-8def-aaea1ac595j">
  <gml:identifier
     codeSpace="urn:uuid:">a82b3fc9-4aa4-4e67-8def-
aaea1ac595j</gml:identifier>
  <aixm:timeSlice>
    <aixm:AirspaceTimeSlice gml:id="ID00001">
      <gml:validTime>
        <gml:TimePeriod gml:id="ID00002">
          <gml:beginPosition>2010-06-29T17:31:00</gml:beginPosition>
          <gml:endPosition>2010-06-29T19:00:00</gml:endPosition>
        </gml:TimePeriod>
      </gml:validTime>
      <aixm:interpretation>BASELINE</aixm:interpretation>
      <aixm:sequenceNumber>1</aixm:sequenceNumber>
      <aixm:type>D</aixm:type>
      <aixm:geometryComponent>
        <aixm:AirspaceGeometryComponent gml:id="ID00003">
          <aixm:theAirspaceVolume>
            <aixm:AirspaceVolume gml:id="ID00004">
              <aixm:upperLimit uom="FT">500</aixm:upperLimit>
              <aixm:upperLimitReference>MSL</aixm:upperLimitReference>
              <aixm:lowerLimit uom="FT">GND</aixm:lowerLimit>
              <aixm:lowerLimitReference>MSL</aixm:lowerLimitReference>
              <aixm:horizontalProjection>
              <aixm:Surface gml:id="ID00005">
                <gml:patches>
                  <gml:PolygonPatch>
                    <gml:exterior>
              ...

</aixm:Airspace>
```

3 Feature Reference (xlink:href)

3.1 Introduction

Associations between AIXM Features are implemented in the AIXM XML Schema through the use of XLinks [XLINK].

The general recommendation is to use the gml:identifier (UUID) of the referenced AIXM Feature. This supports many commercial solutions out-of-the-box, as it relies entirely on the XLink, XPointer and XPath standards. It is critical that every XLink value points to the correct feature of interest. For example, that an xlink value identifying an Airspace holds the identifier of an Airspace feature, not the one of a Runway or another feature, message, etc. However, this cannot be ensured by the XML Schema and it needs to be dealt with as part of the data validation rules.

If the UUID is not available, a natural key search can be used. In general, the verbosity of such a request may increase as finding the key will require querying into a TimeSlice of the AIXM Feature.

From the point of view of the Xlink target, there exist three cases of interest:

1. Concrete local references within a data set;
2. Concrete external references resolved via web services;
3. Abstract references to be resolved by the consuming application.

These are described in more detail in the following sub-sections.

3.2 Concrete local references within a message

In some cases, services producing AIXM 5.1 data will provide data sets in which all the referenced features are included. Instead of a reference by gml:identifier, a simpler local reference to the gml:id attribute could be used in this case. The gml:id is, by definition, unique within an XML file. Therefore, when used for local references, it is unambiguous. The gml:id attributes are defined as identifiers in the schema and can be indexed during parsing.

An example is provided below. Note that the recommendation made in 2.4 is applied here, which means the gml:id of the Airspace feature is actually based on the UUID value of the gml:Identifier.

```
<aixm:Airspace gml:id="uuid.:a82b3fc9-4aa4-4e67-8def-aaea1ac595j">
  <gml:identifier
     codeSpace="urn:uuid:">a82b3fc9-4aa4-4e67-8def-
aaea1ac595j</gml:identifier>

    ...

</aixm:Airspace>
...
<aixm:AirTrafficControlService gml:id="uuid.d4d33081-54ad-4c1a-9519-
b5b67de561ae">
  <aixm:timeSlice>
    <aixm:AirTrafficControlServiceTimeSlice
gml:id="AirTrafficControlService01_TS1">
      <gml:validTime>
        <gml:TimePeriod gml:id="AirTrafficControlService01_TS1_TP1">
          <gml:beginPosition>2008-01-01T00:00:00</gml:beginPosition>
          <gml:endPosition indeterminatePosition="unknown"/>
```

```
          </gml:TimePeriod>
        </gml:validTime>
        <aixm:interpretation>BASELINE</aixm:interpretation>
        <aixm:type>ACS</aixm:type>
        <aixm:clientAirspace xlink:href="#uuid.a82b3fc9-4aa4-4e67-8def-
aaea1ac595j"/>
      </aixm:AirTrafficControlServiceTimeSlice>
    </aixm:timeSlice>
</aixm:AirTrafficControlService>
```

3.3 Concrete external references

When information about features is exposed via web services, a method by which XLinks can be resolved is via a Universal Resource Locator (URL).

In this case, the expectation is that the consumer of the message can follow the URL provided in the XLink directly and the hashtag will identify the feature within the resultant resource to which this reference resolves. If the recommendation made in 2.4 was applied and the target feature has a UUID based gml:id, then the concrete reference may be encoded using simply the '#ID' reference syntax, as in the example below:

```
        <aixm:clientAirspace
xlink:href="http://aim.faa.gov/services/AirspaceService#uuid.a82b3fc9-
4aa4-4e67-8def-aaea1ac595j"/>
```

A more general, but also much more complex solution is to use xpointer. Again, the combination of the URL and the XPointer should result in the tag which resolves to the feature which is referenced.

```
        <aixm:clientAirspace
xlink:href="http://aim.faa.gov/services/AirspaceService?get=a82b3fc9-4aa4-
4e67-8def-
aaea1ac595j#xmlns(ns1=http://www.opengis.net/gml/3.2)xmlns(ns2=http://www.
aixm.aero/schema/5.1)xpointer(//ns2:Airspace[ns1:identifier='a82b3fc9-
4aa4-4e67-8def-aaea1ac595j'])"/>
```

Note that the above URL is independent of implementation. It may identify a (Web Feature Server) WFS server, but it could also be an implementation of a simple web service which returns a particular set of features based on the user's query. As long as the resolution of the URL produces an XML document which contains the AIXM Feature, the above is a valid reference.

The approach for the first two cases (concrete local or external references) is based purely on the XLink and XPointer standard: there is no application-specific logic required to resolve the reference. However, implementers should take care to take advantage of caching strategies to avoid continually resolving features for which they already have definitions.

3.4 Abstract references

Xlink assumes a "resource centered approach", in which the XML document or fragment is a resource which can be referenced from anywhere. It assumes that the resource is available on the web in a single, definitive copy. In the aeronautical information domain, the feature is an entity which can be globally referenced but there are many representations of that feature in circulation. Many of those representations are in AIXM messages; others are in databases or applications. This is why the use of concrete references is rare and the use of abstract references needs to be considered as well.

Abstract references fit very well to the GML paradigm since they provide a reference to the abstract idea of the feature, rather than one of its representations. Resolving the abstract

reference to a physical one is the problem of the application and it can deal with issues of multiple copies in all the messages, databases etc. that it knows about.

Leaving the resolution with the application also allows the application to adapt to its context. For example, if the application is off-line it could choose to use a locally cached copy of the feature. An on-line application could go to the definitive web-service for the feature.

In those cases, the xlink:href should use a universal resource name (URN) rather than a URL; note that URNs, unlike URLs, cannot be directly used to find a resource. In the case where a URN is presented to the consumer, it is the responsibility of the consuming application to resolve the reference.

Nothing in the use of a URN implies the availability of the referenced feature or its location; it may be that the feature is defined locally within the message, accessible remotely by a web service, or directly through a database access.

In general, the following three steps will be followed:

1. The recipient of the data will use the identifier of the referenced feature to search its local database.
2. If no such feature exists in its local data set, the incoming data set would be searched for the referenced feature.
3. If the above does not find the feature, the system would search known data sources to resolve the reference.

This type of reference is limiting in its openness, as it requires application logic to be resolved. As such, its use is expected to be reduced over time, as the aeronautical information domain moves towards Web service solutions. By using the concrete standards mentioned above, any standards-complete system can resolve AIXM references without additional application logic.

3.4.1 Using UUID

It is recommended that the URN is based on the UUID of the referenced feature, as in the following example

```
    <aixm:clientAirspace xlink:href="urn:uuid:a82b3fc9-4aa4-4e67-8def-
aaea1ac595j"/>
```

In this case, the application will consume the URN-based locater and internally discover the definition of the referenced airspace through data registries, manual coding, or other methods specific to the systems involved. The beginning of the URN should match the codeSpace used for gml:identifier.

3.4.2 Using natural keys

When the UUIDs are not available, an AIXM specific URN composed with natural keys could be used, as in the example below:

```
<aixm:clientAirspace xlink:href="urn:aixm:Airspace(gml:timePosition=2010-
04-07T09:00;aixm:type=D;aixm:designator=EBD25A)"/>
```

Apart from being much more complex than those based on UUID, the disadvantage of the URN based on natural keys is that they requires specific code for decoding and identifying the target feature. Therefore, URN with natural keys should be used only during transition periods and on a limited scale. For example, it might be a solution to use natural key based URN between systems that store the data in legacy formats and which do not have the possibility to work with UUID values..

The following rule shall be applied in the composition of the "aixm" URN:

urn:aixm:***Feature***(timePosition=***time_value***;***property*_name**=***property_value***;…)

where:

- ***Feature*** is the name of an AIXM Feature as defined in the AIXM XML Schema; for example: AirportHeliport, Airspace, Runway, etc.
- ***time_value*** is a UTC date and time value in the format yyyy-mm-ddThh:mm and it indicates the moment in time at which the BASELINE TimeSlice of the AIXM Feature referred had the ***property_value*(s)** that appear in the URN composition;
- ***property_name*** is the name of property of an AIXM Feature that composes a natural key for that feature; the property name shall be spelled according to the AIXM XML Schema, exactly as it appears as child element of the AIXM Feature TimeSlice.

  It is assumed that all natural key properties are either direct child elements of the Feature TimeSlice or that they exist only once in the XML tree (such as the gml:pos of a Navaid).
- ***property_value*** is the value of the of the property identified by ***property_name*** as defined in the BASELINE TimeSlice of that feature that is valid at the date and time specific by the ***time_value***;

  In some situations, this includes properties that do not directly have a value, but have an xlink:href attribute that points to another Feature. In this case, the URN of the referenced feature shall be used as property_value. A typical example is the Runway feature, for which the natural key includes the AirportHeliport where it is located. The URN will look like in the example below:

```
urn:aixm:Runway(gml:timePosition=2010-12-
20T16:32;aixm:designator=02%2F20;aixm:associatedAirportHelipor
t=urn:aixm:AirportHeliport(gml:timePosition=         2010-12-
20T16:32;aixm:designator=EBBR))
```

  Note that the value of an AIXM feature natural key property could contain characters that are not allowed in the composition of the URN, as explained in the URN Syntax (RFC 2141) and have to be replaced with their hex code, prefixed by the character "%". This is the case for the "/" character, which is typically used in runway designators. Therefore, in the example above, the runway designator "02/20" was encoded as "02%2F20", where "%2F" is the hex representation of "/".

In order to be valid URN in terms of the RFC 2141 (URN Syntax), the "aixm" URN would have to be registered with the Internet Assigned Numbers Authority. Until then, it shall be considered as a non-standard (experimental) URN.

3.5 Use of xlink:title

The value of the attribute xlink:title on a xlink is a human-readable description of the referenced value. In this case, it would be a human-friendly description of the referenced aeronautical feature.

It is suggested that the xlink:title be used, especially in cases in which the referenced feature is defined remotely. The title should be a human-friendly name of the feature which can be used internally by applications for display purposes. It is discouraged to use the xlink:title for automatic feature identification.

```
      <aixm:clientAirspace xlink:href="urn:uuid:a82b3fc9-4aa4-4e67-8def-aaea1ac595j" xlink:title="Gabbs North MOA"/>
```

Appendix 1

A.1. UUID algorithms

A.1.1 Introduction

This Appendix examines the four main versions of the UUID; how they are generated, their efficiency and their computational overhead. On the basis of this comparison it is found that the version 4 UUIDI, based on random number generation, is the most efficient. The tendency in the industry is to move toward the use of version 4, with the notable exception of Oracle that remains on version 1. A survey of the UUID/Globally Unique Identifier (GUID) versions supported by the main software products and libraries is summarized in Figure 7.

A.1.2 Definition and Uniqueness

A UUID is a 128-bit number that is encoded either with a random number, the output of a cryptographic hash function or a combination of a random number and the time of generation. UUIDs are conventionally written or displayed in their 'canonical' form, which is a sequence of 32 hexadecimal digits grouped into a sequence of 8, 4, 4, 4 and 12 digits; an example is given below.

```
550e8400-e29b-41d4-a716-446655440000
```

The left-most digit of this string represents the most significant four bits of the UUID.

The purpose of the UUID is best described by Wikipedia http://en.wikipedia.org/wiki/UUID as: "*The intent of UUIDs is to enable distributed systems to uniquely identify information without significant central coordination. ....Anyone can create a UUID and use it to identify something with reasonable confidence that the identifier will never be unintentionally used by anyone for anything else.*" The UUID is defined in three compatible standards: ISO/IEC 11578:1996, ITU-T Rec. X.667 | ISO/IEC 9834-8:2005 and IETF RFC 4122.

The reason for the use of a number field as large as [$1-2^{122}$] is that the UUID is ***universal***, i.e. the probability of a duplicate UUID being encountered within the IT universe for the foreseeable future must be a very low.

A.1.3 Format and Versions

The UUID format is depicted in Figure 1. Six of the 128 bits are used to specify the type of UUID leaving 122 to carry a random number, the output of a cryptographic hash function or a combination of a Network Interface Card (NIC) MAC address and the time of generation, depending on the Version of the UUID. The six control bits are very awkwardly placed, and are not even contiguous, making the generation and interpretation of UUIDs more complicated than it need have been. The digits are numbered from the left-most digit of the canonical (string) form. The type of UUID is defined first by the value of the Variant field (YY) which occupies the most significant two bits of the 17th hex digit.

Note: *RFC 4122, paragraph 4.1.1 shows the variant field as having three bits. For all standard UUIDs the least significant bit of these three is irrelevant; it is used as the top bit of the clock sequence in version 1 UUIDs, or is part of the random or hash fields in version 3,4 or 5 UUIDs.*

A value of 2 in the variant field indicates that the UUID is one of the standard versions. A zero value of the most significant bit of the variant field indicates that the UUID was generated by a workstation in the Apollo Network Computing System. A value 3 of the

variant field indicates a UUID that was used by .COM in versions of Microsoft Windows before Windows 2000. The V*ersion* field (VVVV) is coded into the 13th digit allowing up to 15 UUID versions. Five versions, numbered 1 – 5 currently exist.

Hex Digits	1	2	3	4	5	6	7	8
1 to 8	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX
9 to 16	XXXX	XXXX	XXXX	XXXX	**VVVV**	XXXX	XXXX	XXXX
17 to 24	**YY**XX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX
25 to 31	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX

Figure 1 - General Format of UUID

The UUID standards do not require that the UUIDs generated by a given system must all be of the same version. A closed (i.e. not universal) distributed system, could exploit this by allowing any of the four main versions to be generated. The degree of uniqueness of the UUIDs would then be quadrupled, from a field of [$1-2^{122}$] an effective field of [$1-2^{124}$].

A.1.4 Version 1 UUID

The version 1 format is the oldest and most complicated, but is still widely used; the encoding in shown in Figure 2. The 60 bits labeled 't' are the time at which the UUID was created, in increments of 100 nano-seconds ($1x10^{-7}$) seconds. The 48 bits labeled 'm' are the (universally unique) MAC address of one of the Network Interface Cards (NIC) on the system; when this is not available a 48-bit random or pseudo-random number is generated. The 14 clock-sequence bits labeled 'c' are used to reduce the possibility of a duplicate UUID value being generated following the clock being set backwards (after a power outage) or the NIC card being changed. The field allows for 2^{14} or >16,000 resets in the lifetime of the system. In the unlikely event that the clock sequence just before the event is known, it can be used on recovery and just incremented; otherwise the clock sequence is re-initialized with a random number.

Hex Digits	1	2	3	4	5	6	7	8
1 to 8	tttt	tttt	tttt	tttt	tttt	tttt	tttt	tttt
9 to 16	tttt	tttt	tttt	tttt	**0001**	tttt	tttt	tttt
17 to 24	**10**cc	cccc	cccc	cccc	mmmm	mmmm	mmmm	mmmm
25 to 31	mmmm	mmmm	mmmm	mmmm	mmmm	mmmm	mmmm	mmmm

Figure 2 - Format of Version 1 UUID – Time and Address

The time field in a version 1 UUID provides the elapsed time, in 100 nanosecond increments, since the start of the Gregorian calendar in October 1582. The maximum

(unsigned) value of 2^{60} nanoseconds is equal to about 3663 years. Currently the highest order bit set in a version 1 UUID time field is the 57th bit. The 58th and higher bits will not be used until after 2444.

A.1.5 Version 2 UUID

Version 2 UUID is used in the IEEE Portable Operating System Interface POSIX Distributed Computing Environment. The format is very similar to version 1.

A.1.6 Version 4 UUID

The encoding of a version 4 UUID is shown in Figure 3. The 122 bits labeled 'r' comprise a pseudo-random number, or preferably a cryptographic quality random number.

Hex Digits	1	2	3	4	5	6	7	8
1 to 8	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr
9 to 16	rrrr	rrrr	rrrr	rrrr	**0100**	rrrr	rrrr	rrrr
17 to 24	**10**rr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr
25 to 31	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr	rrrr

Figure 3 - Format of Version 4 UUID – Random Number

After a distributed system starts generating version 4 UUIDs, if a source of cryptographic quality random numbers is used, the probability that it will generate a duplicate UUID of one already generated p(n;d) is given by the expression $1 - e^{-nxn/2d}$ where the number already generated is 2^n and d is the number of bits occupied by the random number. The probability of a collision will be higher if pseudo-random numbers generated by the system are used.

As an example, after 243 (≈8.8 trillion) UUIDs have been generated, the probability of the next UUID being a duplicate of one already generated is ≈ 7.276×10^{-12}. Figure 4 gives p(n;d) for values of n: 36, 41, 43 and 46 and values of d: 90, 106 and 122. {calculated with R Statistical Language}. The fourth column of the table shows that taking 4 octets of the random number field for another use would introduce a significant probability of collisions occurring.

	d = 122	d = 106	d = 90
n = 36	4.44×10^{-16}	2.91×10^{-11}	1.19×10^{-6}
n = 41	4.55×10^{-13}	2.98×10^{-8}	1.95×10^{-3}
n = 43	7.28×10^{-12}	4.77×10^{-7}	3.08×10^{-2}
n = 46	4.66×10^{-10}	3.05×10^{-5}	8.65×10^{-1}

Figure 4 - Probability of Collision for Version 4 (Random Number) UUIDs

A.1.7 Versions 3 and 5 UUID

The encoding of a version 3 or version 5 UUID is shown in Figure 5. The bits labeled 'h' are the output of an MD5 cryptographic hash function in version 3 UUIDs and are the output of a SHA-1 cryptographic hash function in version 5 UUIDs. The hash function values of 128 bits (MD5) and 160 bits (SHA-1) are truncated to the 122 bits available in the UUID.

Hex Digits	1	2	3	4	5	6	7	8
1 to 8	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh
9 to 16	hhhh	hhhh	hhhh	hhhh	**0011/0101**	hhhh	hhhh	hhhh
17 to 24	**10**rr	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh
25 to 31	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh	hhhh

Figure 5 - Format of Version 3 or Version 5 UUID

Information about the MD5 algorithm can be found at http://www.w3.org/TR/1998/REC-DSig-label/MD5-1_0, and about the SHA-1 algorithm at http://www.itl.nist.gov/fipspubs/fip180-1.htm. Both of these algorithms have been 'broken' by systematic attacks; in the case of MD5 within 2^{43} hash operations and in the case of SHA-1 within 2^{63} hash operations. These are well below the number of operation that can be guaranteed to produce a collision, 2^{64} and 2^{80} respectively. However this has no implications for the use of hash numbers in UUIDs, where they are used more for convenience than for security, other than to suggest that the uniqueness of a UUID formed with either of these algorithms will be less than that of a version 4 UUID formed with a random number of cryptographic quality.

A.1.8 Comparisons of UUID Versions

Efficiency of bit usage

The time algorithm used in version 1 UUIDs is inefficient in its granularity of 100 nanoseconds. The 56th bit of the 60-bit time field was not used until some time in 1986, and the 57th bit will not be used until 2444. This means that the top 4 bits are effectively not being used. Also, the 14 bit clock sequence field in version 1 UUIDs is only used each time the system is powered up, allowing for 2^{14} or >16,000 resets in the lifetime of the system. A lower number of bits would provide for a sufficient number of resets, given that a recycling of this field would not produce UUIDs with an ambiguous time of generation.

In contrast to version 1 UUIDs, in versions 3, 4 and 5 UUIDs all 122 available bits are used to hold either a random number or the output of a hash function.

Computational overhead

An April 2007 test to compare the times taken to generate version 1, version 3 and version 4 UUIDs { http://johannburkard.de/blog/programming/java/Java-UUID-generators-compared.html } showed the following average times to generate 1 million UUIDs with standard Java software:

Version 1 Time based	5,432 milliseconds
Version 3 MD5 based	40, 788 milliseconds
Version 4 Random number based	48, 900 milliseconds

Figure 6 - Average times for UUID generation

These results show that versions 3 and 4 UUIDs incur substantially higher processing overhead than version 1 UUIDs: by a factor of about 7.5 for version 4 and a factor of about 9 for version 3.

The SHA-1 algorithm generates a 160 bit hash compared to the 128 bits generated by the MD5 algorithm. When used in a UUID, both hashes are truncated to 122 bits, so any advantage in security or randomness that the SHA-1 might have over MD5 is obviated when it is used in a UUID.

Uniqueness

The timestamp field in the version 1 UUID is not a random number. Each bit changes with half the frequency of the preceding, less significant bit; the 48th and higher bits change less than once a year. More significant is the devotion of 14 bits to the clock sequence field, which is only incremented each time a system is switched back on. The Version 1 UUIDs are therefore substantially less unique than either Version 3 or Version 4 UUIDs.

The uniqueness of version 3 or 5 UUIDs can not be greater than the uniqueness of version 4 UUIDs and may be less, given the fact that they can both be compromised after a number of uses that is well below the number at which duplication could be expected ("brute force attack"): 2^{43} instead of 2^{64} in the case of MD5 and 2^{63} instead of 2^{80} in the case of SHA-1.

A.1.9 Support by Common Software

It is sometimes difficult to establish what versions of UUID are supported by a particular software application, as many users and even product announcements do not seem to be aware that five distinct versions exist. Typically the output of a UUID generator is only described as several groups of hexadecimal digits. Oracle is notable in this respect, simply defining its UUID as a 16 byte raw value. In the case of Microsoft, it was difficult to untangle because UUIDs are used extensively within Microsoft packages; only one of many interfaces in the Microsoft Component Object Model (COM), and now in the .NET framework, use the type of UUIDs included in this study. Microsoft switched from using version 1 to version 4 with the introduction of .NET and Windows 2000.

Although a particular application package may not support a given version of a UUID, in many cases an extension to support that version can be added from libraries such as the UNIX-based Open Source Software Project (OSSP).

	Version 1	Version 3	Version 4	Version 5	Variant
Microsoft .COM					3
Microsoft Windows 2000			X		
Oracle	X				
Java (JUG)	X	X	X	X	
Java J2SE5		X	X		

JavaScript **uuid.js**	**X**				
Linux (ossp)	**X**	**X**	**X**	**X**	
MySQL **Version 1**	**X**				
MySQL **current**			**X**		
PostgreSQL			**X**		
Apache **(Jakarta project)**	**X**	**X**	**X**	**X**	
OpenPKG	**X**	**X**	**X**		
Python	**X**	**X**	**X**	**X**	
Ruby	**X**	**X**	**X**	**X**	
C^{++} (ossp)	**X**	**X**	**X**	**X**	

Figure 7 - UUID Versions Supported by Common Software