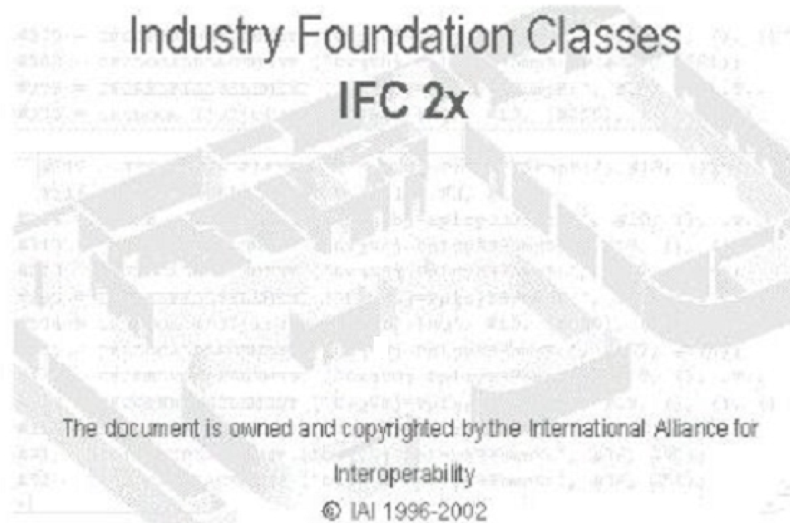
IFC 2*x*

Model Integration Guide



International Alliance for Interoperability
Model Support Group
August 2002

The scope of this document is to provide guidance to projects on approaches to integrating IFC extension models. It is issued in conjunction with IFC 2x and contains guidance relevant to integrating extension models for IFC 2x and future releases that use a platform approach. Its scope includes the integration of models developed by projects that are targeted for a release of the Industry Foundation Classes for the AEC/FM industry (IFC).

It is assumed that the reader is familiar with terminology in normal use within the AEC/FM industry.

All rights reserved. No part of the contents of this document may be reproduced or transmitted in any form or by any means without the written permission of the copyright holder (IAI).

Copyright © 1996-2002 - International Alliance of Interoperability (IAI)

Document Control

Project Reference	IFC Model Integration Guide
Document Reference	Model Integration Guide
Version	1.0
Date	26th August 2002
Status	PUBLIC
Primary Editor	Jeffrey Wix
IAI Committee Responsible	Model Support Group
Distribution	PUBLIC

Version 1.0 – 26/08/2002 - Public release of Model Integration Guide.

Contents

1 INTRODUCTION 6

2 NAMING CONVENTION INTEGRATION 7

2.1 ENSURE THAT IFC NAMING CONVENTIONS ARE FOLLOWED 7

3 CLASS INTEGRATION 9

3.1 DO NOT ALLOW OVERLAPPING CLASSES 9
3.2 ENSURE ADHERENCE TO IFC LAYERED ARCHITECTURE RINCIPLES 9

4 ATTRIBUTE INTEGRATION 11

4.1 ENSURE THAT EMPTY AGGREGATIONS ARE OPTIONAL 11

5 DATATYPE INTEGRATION 12

5.1 USE DEFINED DATATYPES 12
5.2 REUSE EXISTING DEFINED DATATYPES 12

6 SUPERTYPE/SUBTYPE INTEGRATION 13

6.1 USE EXISTING CLASSES AS SUPERTYPES 13
6.2 AVOID DEEP SUPERTYPE/SUBTYPE HIERARCHIES 13
6.3 USE SINGLE INHERITANCE 14
6.4 USE EXCLUSIVE SUBTYPES 15

7 RELATIONSHIP CLASSES INTEGRATION 17

7.1 MAKE RELATIONSHIPS BETWEEN CLASSES USING RELATIONSHIP CLASSES 17
7.2 USE EXISTING RELATIONSHIP CLASSES 18
7.3 DO NOT REDEFINE ATTRIBUTES OF RELATIONSHIP CLASSES 20
7.4 USE INVERSE ATTRIBUTES TO RELATIONSHIP CLASSES 21

8 RULE AND FUNCTION INTEGRATION 22

8.1 TEST REQUIREMENT FOR UNIQUE ATTRIBUTES 22
8.2 ARE WHERE RULES REQUIRED? 23
8.3 TEST THAT WHERE RULES PERFORM THE STATED PROPOSITIONS 23
8.4 CHECK SYNTAX OF RULES, PROCEDURES AND FUNCTIONS 23

9 PROPERTY SET INTEGRATION 24

9.1 SHOULD A PROPERTY SET EXIST WHERE NONE IS DEFINED? 24
9.2 SHOULD A PROPERTY SET THAT IS DEFINED EXIST? 24
9.3 LOOK TO REUSE EXISTING PROPERTY SET: 24
9.4 CHECK FOR SYNONYM 24
9.5 CHECK FOR HOMONYM 25
9.6 CHECK FOR DATATYPE CONSISTENCY 25
9.7 CHECK FOR LEGAL OBJECT REFERENCING 26
9.8 ENSURE EXISTENCE OF PROPERTY SET FOR EACH 'TYPE' ENUMERATION .. 26
9.9 ENSURE PROPERTY SET CONVERSION TO XML 27

10 DOCUMENTATION INTEGRATION 28

10.1 ENSURE EXISTENCE OF DOCUMENTATION 28
10.2 DOCUMENTATION SHOULD MAKE SENSE 28

10.3 PROVIDE USAGE DEFINITIONS.. 29

Figures

Figure 1: Names in a project model before integration 7
Figure 2: Names in a project model after integration... 8
Figure 3: Allowed/disallowed relationships between classes in schema 10
Figure 4: Using a defined datatype instead of a simple datatype 12
Figure 5: Reusing an existing defined datatype 12
Figure 6: Preferred use of existing subtypes 13
Figure 7: Avoiding deep supertype/subtype hierarchies 14
Figure 8: Single and multiple inheritance 14
Figure 9: Exclusive (ONEOF) constraint... 15
Figure 10: No constraint (ANDOR)... 15
Figure 11: Inclusion (AND) constraint 16
Figure 12: Principle of replacing a relationship with a relationship class 17
Figure 13: Domain specification of a 'Helpdesk' action satisfying requests 17
Figure 14: Integrated replacement of ‘request/action’ relationship by a relationship class 18
Figure 15: Principle of using existing relationship class.. 18
Figure 16: Domain assignment of project orders to a 'Helpdesk' action..................... 19
Figure 17: Integrated assignment of project orders to a 'Helpdesk' action 20
Figure 18: Principle of NOT redefining attributes 20
Figure 19: Redefinition of staggered attributes.. 21
Figure 20: Describe inverse attributes on relationship classes.................................... 21
Figure 21: Inverse attributes on IfcRelConnectsElements 21
Figure 22: Class with attributes not defined as unique 22
Figure 23 : Class with unique attribute defined (not asterisk on relationship name).. 22
Figure 24: Synonyms in property sets.. 25
Figure 25: Homonyms in property sets 25
Figure 26: Datatype consistency in property sets.. 25
Figure 27: Type enumeration for valves 27

1 Introduction

The purpose of the IFC Model Integration Guide is to provide guidance to persons who will be carrying out work to ensure the integration of an information model developed by a project with a release of the IFC information model. It is intended to serve the needs both of the 'pre-integration' process whereby the project takes responsibility for developing preliminary integration and for the final integration work undertaken by a qualified person authorized by the IAI Model Support Group.

The guide sets out a number of tasks that the integration process needs to address. These tasks are grouped according to the aspect of the model that they are intended to integrate and include applying the IFC naming convention, classes, attributes, datatypes, supertypes/subtypes, relationship classes, rules and functions, property sets and documentation.

For each integration task, there is a brief description of the purpose of integration and an indication of the steps to be followed to achieve integration. For a number of integration tasks, an example is provided of what happens during and as a result of integration.

Whilst the guide intends to be comprehensive, it is not necessarily complete. There may be additional tasks that need to be completed to ensure that a project model becomes fully integrated with the IFC model. It is the responsibility of the person(s) appointed to carry out the integration to ensure that it is properly and completely achieved.

2 Naming Convention Integration

2.1 Ensure that IFC naming conventions are followed

Ensure that IFC conventions for naming the various components of a model are followed

The IFC model has a strict convention for the naming of schema, entities and data types. A task of the model integration process is to ensure that this convention has been followed. Checks should be made to ensure that the naming convention has been followed at the schema, class, data type, relationship, rule/function and property set levels

Integration

Check and ensure that:

1. all names used are in upper and lower case characters with words running on from each other (no spaces or other separation characters such as underscore _);
2. schema, classes, data type and function names have the 'Ifc' prefix;
3. schema are named according to level of the IFC technical architecture at which they exist e.g. IfcDrainageDomain for a schema at the domain layer of the architecture;
4. schema do not have any extraneous characters that may have been used during the project model development to identify the project e.g. IfcDrainageDomain_BS99;
5. class names are nouns or noun phrases except where it is a relationship class;
6. relationship class names for classes at the IfcKernel schema and above include the 'IfcRel' prefix follow by a verb or verb phrase;
7. relationship classes within the resource layer have the suffix 'Relationship' appended;
8. relationship names do NOT have the 'Ifc' prefix
9. enumeration and select data types have the correct suffix (paying particular attention to type enumerations that will drive property sets);
10. property sets have the 'Pset_' prefix;
11. names of property sets that are type driven conform to the names given in the type enumeration from which they are driven e.g. if the enumeration value is 'ISOLATINGVALVE', the type driven property set name should be Pset_IsolatingValve;

Example

In the HelpDesk request/action scenario, the request and action classes and the relationship name are initially in lower case characters and an underscore character connects the individual words making up the entity name[1].

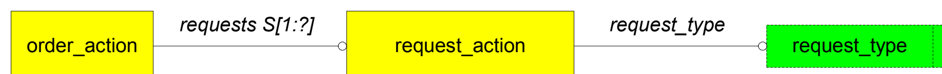


Figure 1: Names in a project model before integration

To conform to the IFC naming convention:

- amend words to commence with an upper case character (Order_Action, Request_Action and Requests),
- remove the underscore character (OrderAction, RequestAction),
- add the Ifc prefix is to any entities, data types (defined, enumeration or select) and functions,

[1] This is the naming convention used in schema developed for ISO TC184/SC4

- add the appropriate suffix is to enumeration (Enum) and select (Select) data types.

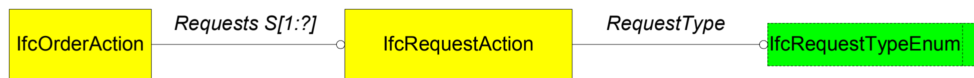


Figure 2: Names in a project model after integration

3 Class Integration

3.1 Do Not Allow Overlapping Classes

A class should not wholly or partially fulfil the same scope as a class already defined within the IFC Model.

Project models may define classes to fulfil particular business requirements either with reference to existing IFC classes or not. A task of integration is to check if the scope of any class defined specifically within the project model can be handled by the scope of classes that already exist within the IFC model. Meeting scope by existing classes may be for the whole or a part of the scope of a class within the project model.

Integration

1. For each class that is defined within a project model that is not already defined within the IFC model, check the scope of information capture for the class.
2. Check the IFC model to determine if an existing class wholly or partly meets this scope.
3. If an existing class wholly meets the scope, use it to replace the class in the integrated project model.
4. If the scope is partially met by more than one existing class, adjust the schema of the integrated project model to reflect all the existing classes and how they are related.
5. If the scope is partially met by one existing class, use the existing class in the integrated project model schema and adjust the scope of the class proposed by the project model accordingly. In this case, relationship(s) between the existing class and the proposed new class will also need to de defined during integration.
6. If the scope is not met at all by existing classes, include the proposed new class in the integrated project model.

3.2 Ensure adherence to IFC layered architecture rinciples

A class may reference or use a class at the same or lower layer within the IFC Technical Architecture but may not reference or use a class from a higher layer.

The IFC Model is structured according to a well-defined architecture that is layered. Starting from the lowest, the layers are:

1. Resource layer
2. Core layer
3. Interoperability layer
4. Domain layer

A class at a given layer may only have a relationship with a class that is at the same or a lower layer of the architecture. It is not allowed to create a relationship to a class at a higher layer of the architecture.

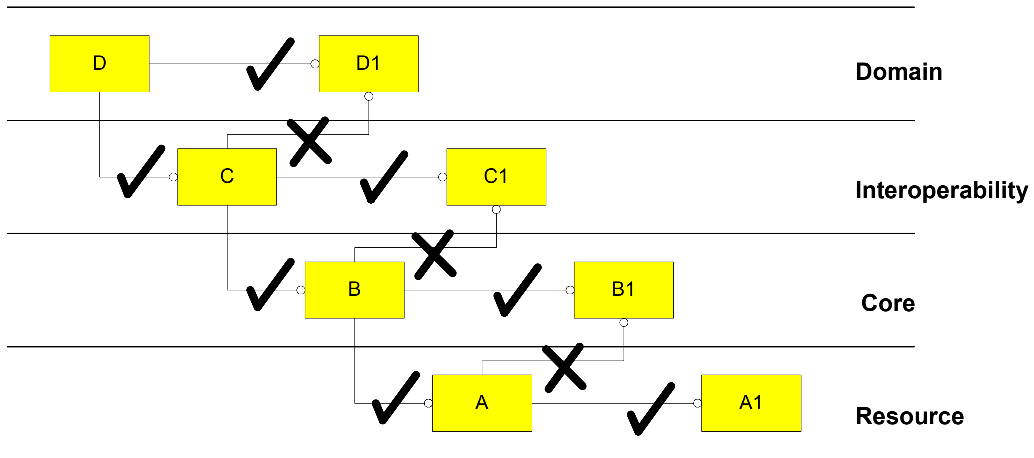


Figure 3: Allowed/disallowed relationships between classes in schema

Integration

Integration needs to check each relationship between classes (including relationship classes) to ensure that the architectural rules are properly applied. Where a breach of the rules is observed, a class making a relationship to a class in a higher level schema, then either:

- the relationship needs to be adjusted so that it is made with a class at the same or a lower layer,
- the class to which the relationship is made needs to be moved a schema at a lower layer in the architecture.

4 Attribute Integration

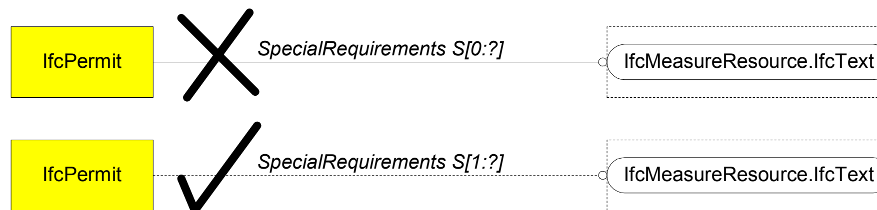
4.1 Ensure That Empty Aggregations Are Optional

Aggregation relationships (LIST, SET, BAG) that may be specified as empty are defined as an optional relationship with a cardinality of one to many.

Aggregations (ARRAY, BAG, LIST, SET) can be defined such that an empty aggregation is acceptable. There are two ways in which this can be done:

1. make assertion of the attribute mandatory and declare the cardinality of the aggregation to be zero to many,
2. make assertion of the attribute optional and declare the cardinality of the aggregation to be one to many,

Whilst within the EXPRESS model, the two constructs above may be seen as achieving the same result, they do have different effects on the data exchange file that results from applying the model[2]. In case 1, the exchange file will show the aggregation as an empty list with no members thus ( ) whilst case 2 will show the optional placeholder character $.



For model development within IFC, aggregations that may be empty should be shown as optional aggregations with a cardinality of one to many. This is shown in the 'ticked' correct diagram above.

Integration

1. Search through the model for instances where an aggregation may be marked as mandatory with zero to many [0:?] cardinality.
2. Change the attribute to optional and the cardinality to being one to many [1:?].

[2] There is also not a difference for software implementations that export or import the data exchange file. Since the aggregation may contain values, the allowance must be made for it. Therefore, from the software perspective, the concept of an optional aggregation does not exist; it is a mandatory aggregation that has no members.

5 Datatype Integration

5.1 Use Defined Datatypes

All attributes that resolve to a simple data type (INTEGER, REAL, STRING, LOGICAL) shall be specified as defined data types.

In general, attributes that resolve to simple data types within the IFC model should be defined either as properties within property sets (q.v.) or as defined datatypes. The use of defined datatypes in particular provides improved semantics within the model in comparison to a simple datatype.

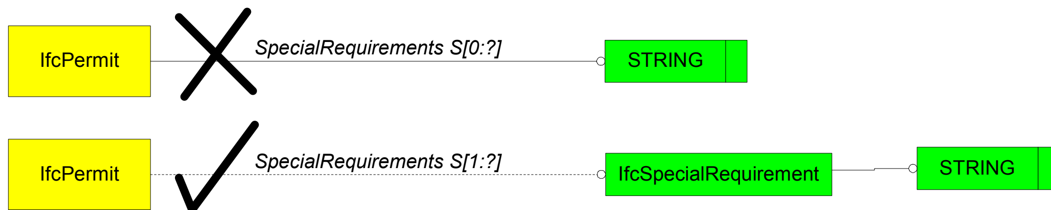


Figure 4: Using a defined datatype instead of a simple datatype

Integration

1. Check through the model and look for the occurrence of attributes that resolve to simple datatypes
2. If a class has several attributes that resolve to simple datatypes then it is most probable that these should be migrated top a property set (q.v.)
3. If there are one or two such attributes then consider changing them to defined datatypes that themselves resolve to the indicated simple datatype.

5.2 Reuse Existing Defined Datatypes

Make use of existing datatypes wherever possible rather than define new ones.

For attributes within a schema that resolve to defined datatypes, the objective should be to reuse datatypes that are already defined rather than defining new ones. This limits the number of datatypes that exist within the model and make implementation easier.

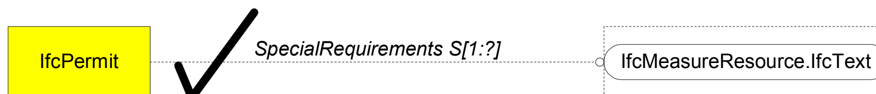


Figure 5: Reusing an existing defined datatype

Integration

1. Check through the model and look for the occurrence of attributes that resolve to defined datatypes
2. For each defined datatype, establish whether an existing datatype can be used instead (most of the reusable defined datatypes are in the IfcMeasureResource schema; but not all of them).
3. Where an existing defined datatype can be used, replace that specified in the project model with a reference to that existing (usually by reference from another schema)

6 Supertype/Subtype Integration

6.1 Use Existing Classes as Supertypes

A subtype should use an existing class as a supertype by preference.

A project model may describe a supertype/subtype hierarchy in terms of classes that it defines. On integration, it may be considered appropriate to change the supertype of a class from one defined by the project model to one that is already defined by the IFC Model. This may be because the functionality of the supertype class proposed by the project model is already covered by an existing class (see also 'Do Not Allow Overlapping Classes') or some other reason.

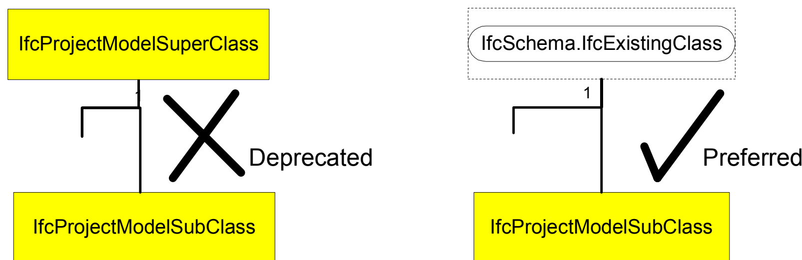


Figure 6: Preferred use of existing subtypes

Where integration does cause a change in the supertype of a class, care should be taken to ensure that the new supertype does in fact fulfill the functional requirements that were expected from the project model supertype.

Integration

1. Check if a subtype class can use an existing class in the IFC model as a supertype
2. If so, change the supertype

6.2 Avoid Deep Supertype/Subtype Hierarchies

The depth of a supertype/subtype hierarchy should be kept as small as possible.

In developing the IFC model, careful atgtention has been given to keeping the depth of supertype/subtype hierarchies as shallow as possible. That is, there should be as few as possible supertype classes between the IfcRoot class and a class that fulfils a functional requirement. In particular, classes that are simply used for structuring then taxonomy of the model are deprecated (unless the resulting list of subtypes becomes huge in which case good modelling practice suggests that a structuring layer of classes should be introduced.

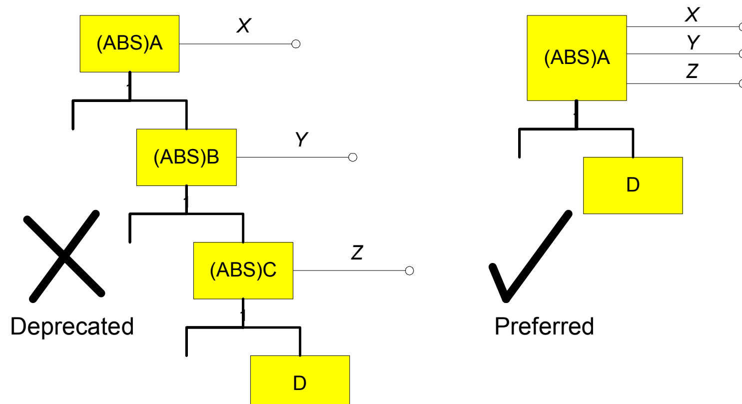


Figure 7: Avoiding deep supertype/subtype hierarchies

When integrating a project model, the depth of supertype/subtype hierarchies should be examined. If there are a number of abstract supertypes in the hierarchy (classes that are not directly instantiated) then integration should seek to reduce the depth of the hierarchy by deleting intermediate classes. However, it must be remembered that any attributes that exist on a class that is to be deleted may need to be migrated to a class that is to remain.

Note that restriction of the supertype/subtype hierarchy depth also promotes easier software implementation.

Integration

1. Check through the model and look for incidences of a deep hierarchy (here defined as at least two layers of abstract supertype before a class that is directly instantiated)
2. Determine if one or more layers in the hierarchy can be deleted.
3. Migrate attributes from classes that are to be deleted to classes in the hierarch that are to remain.
4. Delete classes to be removed.

6.3 Use Single Inheritance

A subtype is a specialization of exactly one supertype.

Within the IFC model, all supertype/subtype relationships use the concept of single inheritance. That is, a class may be a subtype of only one supertype class.

Multiple inheritance, which is where a class may be a subtype of more than one supertype class is not allowed.

The easiest way to test for multiple inheritance is to use the EXPRESS language form of the model. However, it may also be seen from a graphical form of the model as well.

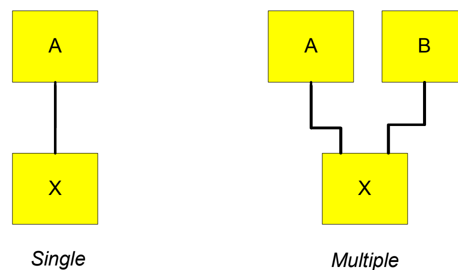


Figure 8: Single and multiple inheritance

In a graphical model, single inheritance is seen when there is a single broad subtype line taken to the subtype (child) class from a subtype (parent) class. In EXPRESS, it can be more easily seen from the statement that says '*entity X is a subtype of only one entity which is A*'.

```
ENTITY X
      SUBTYPE OF (A);
END_ENTITY
```

In a graphical model, multiple inheritance is seen when there are two (or more) broad subtype lines taken to the subtype (child) class. In EXPRESS, it can be more easily seen from the stament that says 'entity Y is a subtype of two entities which are B and C.

```
ENTITY Y
      SUBTYPE OF (B, C);
END_ENTITY
```

Integration

Where multiple inheritance exists in a project model it needs to be removed. This will involve reviewing the model and looking for alternative ways to achieve the objectives required by the inclusion of multiple inheritance in the first place. This could mean extensive restructuring of the model and the development of new classes and attributes. Care needs to be exercised in doing this so that the functionality of the model is not changed or compromised.

Because of the potential need for restructuring, the discovery of multiple inheritance should cause the integration process to be suspended. The model should be returned to the project. It should be the responsibility of the project to carry out the necessary restructuring to remove the incidence of multiple inheritance whilst at the same time ensuring that the original objectives and scope of the project continue to be met.

6.4 Use Exclusive Subtypes

A supertype is constrained so that it can be instantiated exclusively by one of its subtypes.

Within the specification of the EXPRESS language, three types of subtype constraint can be applied. These are ONE OF, ANDOR and AND: Within the IFC model, only the ONEOF or exclusively constrained subtype mechanism should be used.

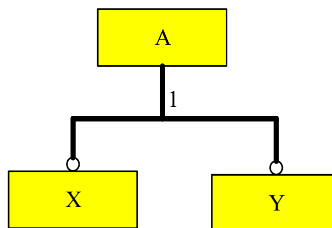


Figure 9: Exclusive (ONEOF) constraint

1. ONEOF constraint in which a supertype (class A) must be instantiated exclusively by one of its subtypes (class X OR class Y). In EXPRESS-G, a numeric ‘1’ character at the branch of the supertype/subtype relationship line shows this. In EXPRESS, the code is of the form:

```
ENTITY A
   SUPERTYPE OF (ONE OF (X, Y));
END_ENTITY
```

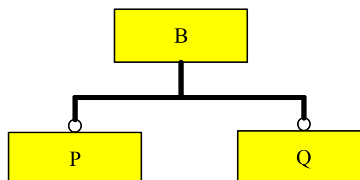


Figure 10: No constraint (ANDOR)

2. ANDOR constraint in which a supertype (class B) may be instantiated either by a combination of its supertypes (class P AND class Q) or by one of its subtypes alone (class P OR class Q). In EXPRESS, the code is of the form:

```
ENTITY B
END_ENTITY
```

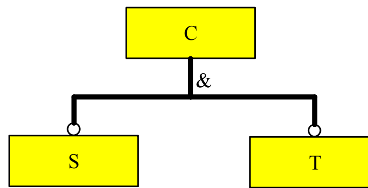


Figure 11: Inclusion (AND) constraint

3. AND constraint in which a supertype (class C) must be instantiated by a combination of its subtypes (class S AND class T). This is seen within the EXPRESS-G notation by an ampersand '&' character at the branch of the supertype/subtype relationship line. In EXPRESS, the code is of the form:

```
ENTITY C
    SUPERTYPE OF (S AND T);
END_ENTITY
```

Integration

Ensure that all supertypes that have multiple subtypes use the ONEOF constraint.

Note that where a supertype has only one subtype, the constraint will not show in either EXPRESS-G or EXPRESS.

Where an ANDOR constraint is used in the project model, this may be changed to a ONEOF constraint directly. However, the project should be contacted to ensure that there are no circumstances where this may create problems (because of an intention to use the AND part of the constraint).

Where an AND constraint is used, an alternative approach needs to be determined in conjunction with the project. One possibility is to use a 'role' attribute and to allow the object to play more than one role.

7 Relationship Classes Integration

7.1 Make Relationships Between Classes Using Relationship Classes

Relationships between classes that are subtypes of IfcObject (at any level) shall be defined through the use of a relationship class by preference.

Within the IFC model, it is preferred that relationships made between classes are defined through a relationship class rather than directly between the classes themselves. Although this appears to make the relationship more complex, it does provide more overall flexibility within the model.

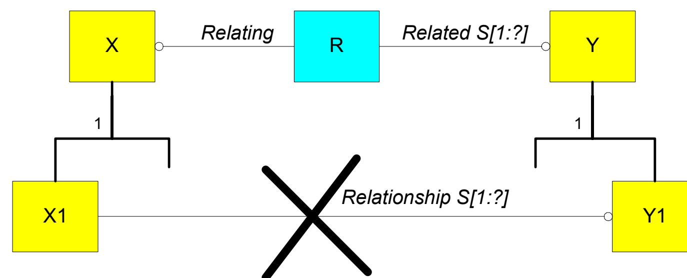


Figure 12: Principle of replacing a relationship with a relationship class

As far as possible, it is also preferred that existing relationship classes should be used rather than defining new classes.

Consider the case where a domain model defines a relationship between two classes, X1 and Y1 such that an instance of X1 has a relationship with one or more instances of Y1.

- The first aspect of model integration in this case is to consider the addition of a relationship class between X1 and Y1 such that there is one relating
- Class X1 is a subtype of class X. Class Y1 is a subtype of class Y. Classes X and Y are already specified in the IFC model.
- An class R exists already within the IFC model to define a relationship between one relating class X and one or more related classes Y.
- Therefore R1 is a repetition (and potentially a redefinition) of R and consequently is redundant. It should be deleted on integration.

Example

This example is from a Facilities Management model and captures the idea of one or more requests made to a Helpdesk for an action to take place and the action that satisfies these requests.

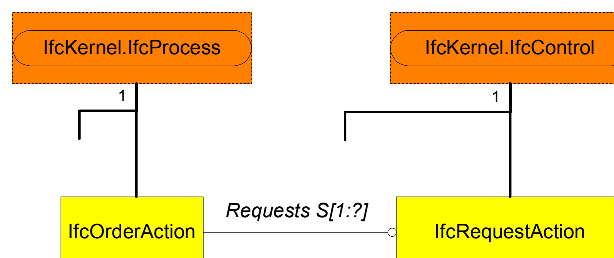


Figure 13: Domain specification of a 'Helpdesk' action satisfying requests

The domain model defines a relationship whereby an action satisfies the requirements of one or more requests. The inverse, that each request may be met by zero or one actions, is not specified in the domain model.

Integration

The clues to look for in this example are:

- Check for the existence of an appropriate relationship class existing between the supertypes of the classes in the domain model.

Within the IFC kernel schema, there exists an IfcRelAssignsToProcess class. This handles the assignment of one or more objects (instances of IfcObject or a subtype) that the process (instance of IfcProcess or a subtype) operates on[3]. Because RequestAction is ultimately a subtype of IfcObject, it can participate in is such a relationship.

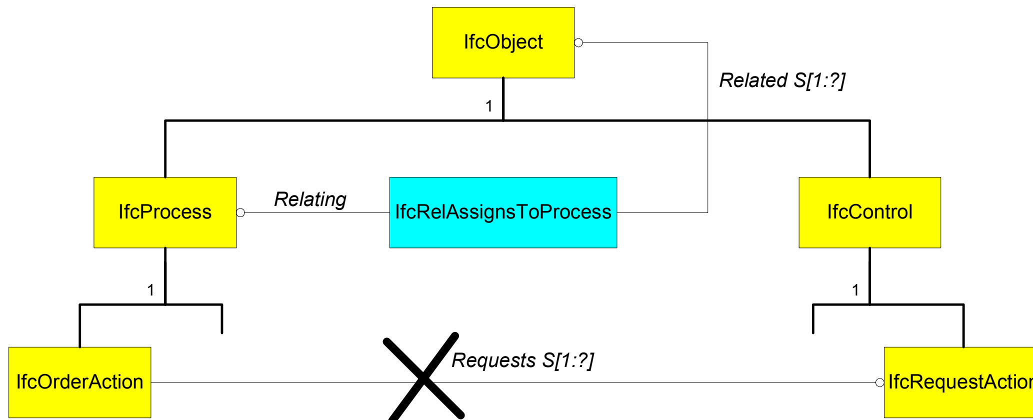


Figure 14: Integrated replacement of 'request/action' relationship by a relationship class

Therefore, the IfcOrderAction.Requests attribute can be deleted during integration since the IfcRelAssignsToProcess class that is already within the IFC model already captures the required information.

7.2 Use Existing Relationship Classes

Make use of existing relationship classes wherever possible rather than define new ones.

The IFC model already contains a number of general-purpose relationships. These cover many of the cases where there is a need to define an objectified relationship between classes. By reusing an existing relationship, the overall size of the IFC model is minimized.

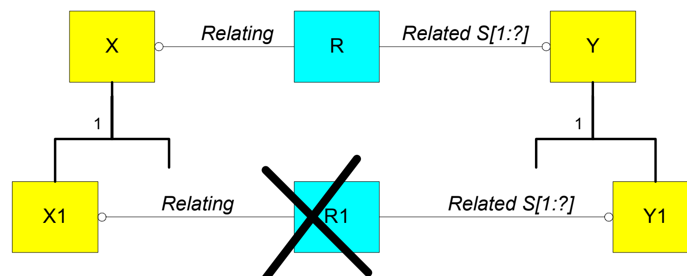


Figure 15: Principle of using existing relationship class

Consider the case where a domain model defines a relationship R1 between one relating class X1 and one or more related classes Y1.

- Class X1 is a subtype of class X. Class Y1 is a subtype of class Y. Classes X and Y are already specified in the IFC model.

[3] In this case, the optional quantity attribute is not asserted

- An class R exists already within the IFC model to define a relationship between one relating class X and one or more related classes Y.
- Therefore R1 is a repetition (and potentially a redefinition) of R and consequently is redundant. It should be deleted on integration.

Example

This example is from a Facilities Management model and captures the idea that a Helpdesk may define an action that results in the generation of several work orders. The idea of the work order is captured in this case by its supertype IfcProjectOrder since an action may result in the generation of any type of order.

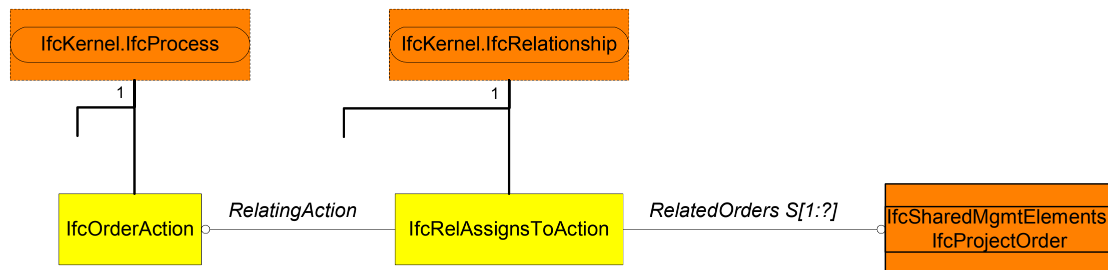


Figure 16: Domain assignment of project orders to a 'Helpdesk' action

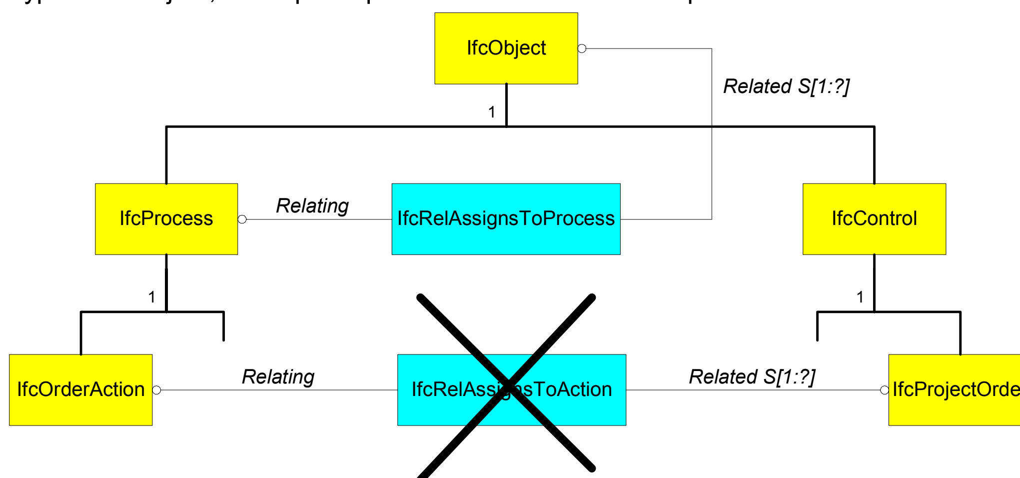
The domain model defines a relationship class IfcRelAssignsToAction that enables an identified 'relating' action from the helpdesk to be related to one or more orders. IfcRelAssignsToAction is defined as a subtype of IfcRelationship (which is the top level relationship class defined in the IfcKernel schema)

Integration

The clues to look for in this example are:

1. The 'relating' class IfcOrderAction is defined as a subtype of IfcProcess (because it describes an action).
2. The 'relationship' class describes an assignment of an action to a project order (describing a relationship between them that is not a dependency).

Within the IFC kernel schema, there exists an IfcRelAssignsToProcess class. This handles the assignment of one or more objects (instances of IfcObject or a subtype) that the process (instance of IfcProcess or a subtype) operates on[4]. Because IfcProjectOrder is ultimately a subtype of IfcObject, it can participate in is such a relationship.



---

[4] In this case, the optional quantity attribute is not asserted

Figure 17: Integrated assignment of project orders to a 'Helpdesk' action

Therefore, the IfcRelAssignsToAction class proposed can be deleted during integration since the IfcRelAssignsToProcess class that is already within the IFC model already captures the required information.

7.3 Do Not Redefine Attributes of Relationship Classes

Attributes defined at a supertype level are not redefined at the subtype level

Within a supertype/subtype hierarchy of relationship classes, the conventions for IFC modeling require that attributes that are defined at the supertype level should not be declared again (redefined) at the subtype level.

Although this is allowed within the EXPRESS data definition language, IFC modeling does not consider it to be good practice and it should be eliminated on integration.

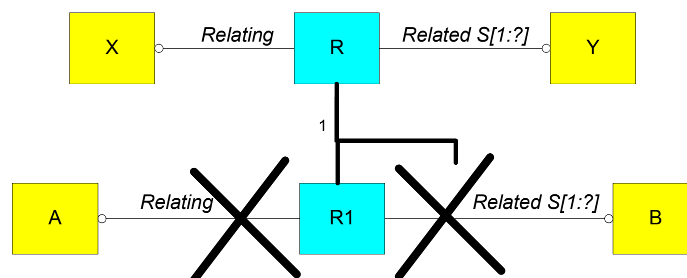


Figure 18: Principle of NOT redefining attributes

Consider the case where a domain model defines a relationship R between one relating class X and one or more related classes Y and then defines a relationship R1 between one relating class A and one or more related classes B.

- Class R1 is a subtype of class R.
- The relationships R→X and R→Y are inherited by R1
- Therefore the relationships R1→A redefines R→X and Ri→B redefined R1→Y.
- A and B may not be subtypes of X and Y in which case the redefinition may have unwanted side effects in that the relationship will not act in the manner intended at the supertype.
- A and B may be subtypes of X and Y in which case the redefinitions echo the initial definition and are redundant.

Integration

1. For any relationship class that is a subtype, check if it has relating and related attributes.
2. If yes, check the supertype to see if it also has relating and related attributes (continue up the supertype/subtype tree until these attributes are located).
3. If these attributes also exist at a supertype level, delete them from the subtype level.

Special Case

There is a special case for the redefined attribute example. That is where only one of the attributes is declared at the supertype level, the other being able to be declared at the subtype level

In this case, only the one attribute that is redefined needs to be deleted, the other can remain.

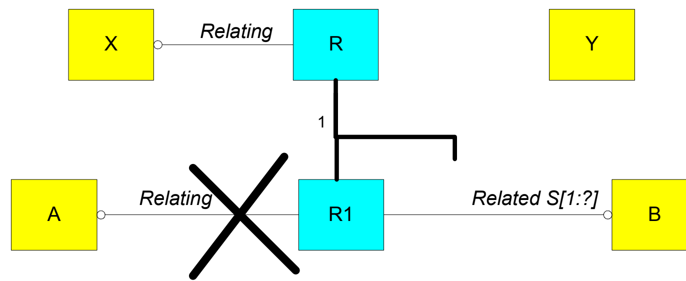


Figure 19: Redefinition of staggered attributes

7.4 Use Inverse Attributes to Relationship Classes

Attributes of Relationship Classes Should have Inverse Attributes Defined.

Each relationship class has at least a relating attribute that has as its target type the class to which the relationship is being made and a related attribute that has as its target type the class that is being related. Each of these attributes should have, on integration, an inverse attribute that points from the target class to the relationship class.

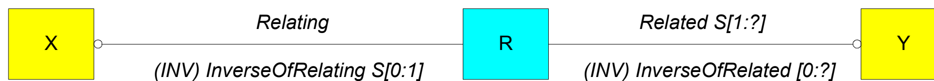


Figure 20: Describe inverse attributes on relationship classes

The nature of the inverse relationship in terms of its name, and its cardinality, needs to be determined in the context of what the relationship is intended to achieve.

Example

In the case of the connection relationship between elements in the IfcKernel schema, as well as determining which is the relating and which the related IfcElement in the connection relationship (both defined as 1:1 relationships), the inverse relationships describe the elements which the connection relationship connects. Thus, the relationship is described fully.

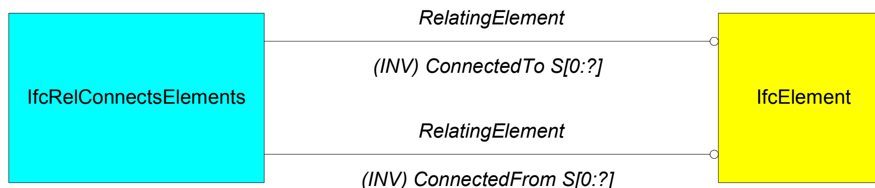


Figure 21: Inverse attributes on IfcRelConnectsElements

Warning

Software tools used for creating IFC schema do not allow inverse relationships to be defined between classes that are defined in different schema. However, if all of the separate schema are then to be merged into a single, long form schema, the inverse relationships need to be defined within the merged schema.

To avoid the need for inverse relationships crossing schema boundaries, relationship classes need to be carefully designed, located (within a schema) and integrated such that the classes and relationships concerned all occur within the same schema. This can be difficult. In general, it is better to try to achieve relationships between classes using existing relationships wherever possible.

8 Rule and Function Integration

8.1 Test Requirement for UNIQUE Attributes

Attributes that need to have unique values should be identified by a UNIQUE rule.

An attribute of a class may be required to take a unique value for each instance of the class to which it is related. This ‘unique’ attribute provides an identity key to an entity. It may be a single value such as an identifier as in the example below, or it may be a combination of attributes.

Attributes of a class that should take a unique value should be explicitly identified through a UNIQUE rule in the ERXPRESS code. A unique attribute is identified in EXPRESS-G by an asterisk that prefixes the attribute name (i.e. the name of the relationship in the diagram).

Integration

1. For each class having attributes, review the list of attributes on the class
2. Are there any unique attributes already defined?
3. Should any of the attributes not currently defined as unique be specified so that they are unique,
4. If so, add UNIQUE rule to the relevant class and specify the attribute(s) that are to be unique.

Example

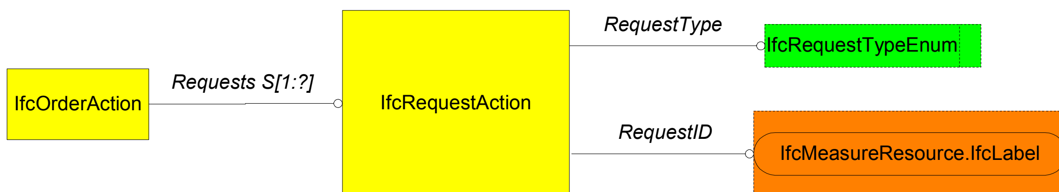


Figure 22: Class with attributes not defined as unique

In the example, the class IfcRequestAction has two attributes; a request type and a request identifier. This is shown in EXPRESS by the following:-

```
ENTITY IfcRequestAction
   SUBTYPE OF (IfcControl);
      RequestType : IfcRequestTypeEnum;
      RequestID : IfcLabel;
END_ENTITY;
```

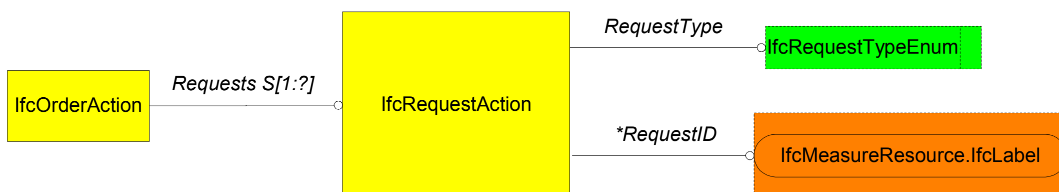


Figure 23 : Class with unique attribute defined (not asterisk on relationship name).

There is a need however to provide for the unique identification of every request so that resulting actions can be tracked against those requests and the persons issuing the request notified of the response. This is done by making the request identifier into a unique attribute. The result is seen in EXPRESS-G by the asterisk adjacent to RequestID and in EXPRESS by the following:-

```
ENTITY IfcRequestAction
   SUBTYPE OF (IfcControl);
      RequestType : IfcRequestTypeEnum;
      RequestID : IfcLabel;
   UNIQUE
      UR1 : RequestID;
END_ENTITY;
```

8.2 Are WHERE Rules Required?

Provide a WHERE rule where there is a need to constrain the population of a class.

A WHERE rule identifies a constraint that can exist on the data population of an instantiated class. For instance, in a system, a WHERE rule is used to limit the population of classes that can participate to being of type IfcElement.

A WHERE rule is developed in response to a proposition on the class, usually expressed in a usage requirement.

Integration

1. Check the usage requirements for a class and determine if propositions have been made OR,
2. Check whether propositions should have been made according to the description of requirement.
3. Confirm any additional propositions with the project.
4. Add WHERE rules as necessary.

8.3 Test That WHERE Rules Perform the Stated Propositions

WHERE rules should respond to a proposition dated in a usage requirement.

For a WHERE rule that has been incorporated, the proposition should be checked and the rule as written tested to ensure that it is actually doing what the proposition says it should be doing.

Integration

1. Check each rule against the proposition in the usage requirements to ensure that it is fulfilling the stated need.

8.4 Check Syntax of Rules, Procedures and Functions

Syntax for Rules, Procedures and Functions must be valid.

Checking procedures of this nature can be done typically using the software tools used for development of the model.

Integration

1. Check the syntax of each rule, procedure and function for validity.

9 Property Set Integration

This section provides broad guidance on the integration of property sets within the IFC model. Further important information is given in the IFC Property Set Guide.

Integration of property sets is more concerned with good practice in their definition than with formal integration in the manner of classes and attributes. Whilst property sets are a valid part of the IFC model (as valid as the classes that are defined in EXPRESS), the tools to provide for quality checking on integration do not exist in the same way.

A dictionary of the properties defined in IFC property sets exists. It should be used during project model definition and also during integration to check that properties are beiong used consistently.

9.1 Should A Property Set Exist Where None Is Defined?

Where several attributes of a class resolve to simple datatypes, consider migrating them to being properties in a property set

An IFC Property Set is a collection of properties that ultimately resolve to simple data types (REAL, INTEGER, STRING etc.). There are circumstances when it might be better to use a property set rather than defining attributes within a class:

Integration

1. Does a class have a lot of properties that resolve to simple data types
2. If yes, consider whether these attributes can be collected together into a property set.

9.2 Should A Property Set That Is Defined Exist?

Where a property set has few properties or where the properties do not collectively perform a useful function, consider whether the property set should exist at all.

There may also be situations where a project has defined a property set that is not really valid. As a 'rule of thumb', a property set should perform a describable, useful function e.g. provide performance data about something. If this cannot be done, the existence of the property set should be questioned.

Integration

1. Does a property set perform a valid function in entirety?
2. If not, then it may be appropriate to delete it and migrate properties back to attributes.

9.3 Look To Reuse Existing Property Set:

If a property set already exists that fulfils the purpose required, it should be reused.

Integration

1. Does a property set that performs the required function already exist?
2. If yes, use the existing property set.

9.4 Check For Synonym

A definition of a named property should always have the same property name.

A synonym is where two different names are used for the same property definition.

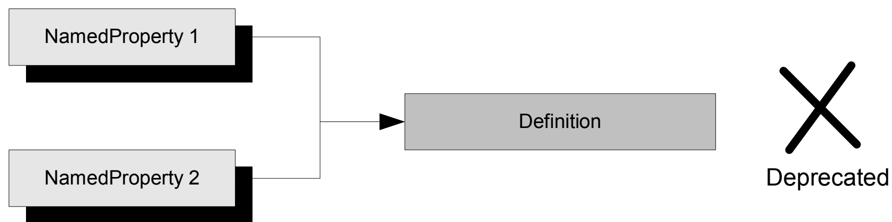


Figure 24: Synonyms in property sets

Integration

1. Test for each property in a property set
2. Does the same or an equivalent definition of a property exist within another, already defined, property set?
3. If yes, a synonym exists. Rename the property to be the same as that already defined.

9.5 Check For Homonym

A named property should not have a different definition to an existing property of the same name.

A homonym is where the same property name is used for two different definitions.

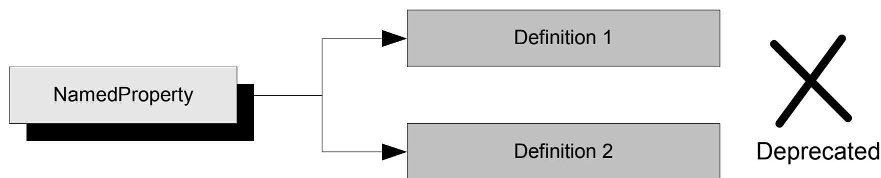


Figure 25: Homonyms in property sets

Integration

1. Test for each property in a property set
2. Does a property with the same name exist within another, already defined, property set?
3. Is the definition the same?
4. If not, a homonym exists. Rename the property to be different to that already defined.

9.6 Check For Datatype Consistency

A named property should not have a different datatype to an existing property of the same name.

Where a property that has already been defined is reused in another property set, the specification of the datatype should be the same.

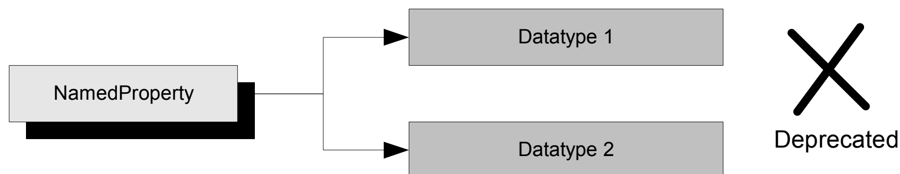


Figure 26: Datatype consistency in property sets

Integration

1. Test for each property in a property set
2. Does a property with the same name exist within another, already defined, property set?
3. Is the datatype the same?
4. If not, should it be the same?
5. If not, a homonym exists. Rename the property to be different to that already defined.

9.7 Check For Legal Object Referencing

A property may reference a class that exists at a resource layer of the IFC model but may not reference classes at any other layer.

There is a strict rule concerning the referencing of classes that exist in the IFC model from within a property set. This can only be done with classes that exist at the resource layer of the model and not for classes at any other layer. This is because resource classes do not carry their own globally unique identifier; they are existence dependent on the object for which they are an attribute. A property set in IFC does have its own globally unique identifier and so can have existence dependent attributes.

Classes that can be referenced are identified in the datatype IfcObjectReferenceSelect and integration should check that any classes referenced from within a property set are in the allowed select list.

Integration

1. Test for each property in a property set that resolves to a class within the IFC model
2. Does the class referenced exist in the allowed select list defined by IfcObjectReferenceSelect.
3. If not, then the reference is not valid.
4. Confirm the intention of the property with the project and change the reference to being a valid class or another property subtype.

9.8 Ensure Existence Of Property Set For Each ‘Type’ Enumeration

An enumeration that is used to specify the existence of type driven property sets has the suffix "TypeEnum".

Type enumerations are used to drive property sets that contain extended information concerning an object. Each type enumeration is the datatype for a ‘type’ attribute whose name is constructed form the name of the class + the word ‘Type’.

The values in the enumeration list represent the different type driven property sets that have been defined for that class. The only exceptions to this are the enumeration values USERDEFINED, NOTDEFINED, UNSET. When a particular value is set from the enumeration list, the property set whose name corresponds to the ‘type’ entity plus the selected enumeration value is also instantiated.

Integration should therefore check that, for each value in a type enumeration, there is a corresponding property set defined.

Integration

1. Check for type enumerations in the model
2. For each type enumeration, check for the existence of a property set whose name corresponds to that of the enumeration value
3. Check that, for a type driven property set, that the type class name is included in the property set name as a prefix to the enumeration value.

Example

Consider the class IfcValveType that has a ValveType attribute whose datatype if IfcValveTypeEnum.

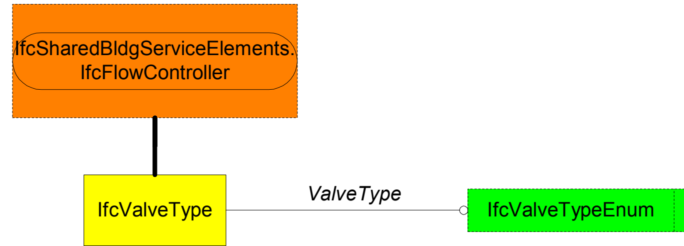


Figure 27: Type enumeration for valves

Values in the enumeration are	*Corresponding property sets are*
Valve_AirRelease	Pset_ValveTypeValve_AirRelease
Valve_AntiVacuum	Pset_ValveTypeValve_AntiVacuum
Valve_Changeover	Pset_ValveTypeValve_Changeover
Valve_Check	Pset_ValveTypeValve_Check
Valve_Commissioning	Pset_ValveTypeValve_Commissioning
Valve_Diverting	Pset_ValveTypeValve_Diverting
Valve_DoubleRegulating	Pset_ValveTypeValve_DoubleRegulating
Valve_Flushing	Pset_ValveTypeValve_Flushing
Valve_Isolating	Pset_ValveTypeValve_Isolating
Valve_Mixing	Pset_ValveTypeValve_Mixing
Valve_PressureReducing	Pset_ValveTypeValve_PressureReducing
Valve_PressureRelief	Pset_ValveTypeValve_PressureRelief
Valve_Regulating	Pset_ValveTypeValve_Regulating
Valve_SafetyCutOff	Pset_ValveTypeValve_SafetyCutOff

9.9 Ensure Property Set Conversion To XML

IFC property sets shall be delivered in the Property Set Definition (PSD) form of XML

All IFC property set definitions that are to form part of a published model need to be converted in the PSD XML language defined for this purpose by the IAI Model Support Group. Integration needs to check that property sets are delivered in this form and that the XML code is valid[5].

Integration

1. Check that all property sets have been converted into PSD XML
2. Where a property set has not been converted, either contact the project so that they can carry out the necessary conversion OR make the conversion directly.
3. Validate the XML syntax of all property sets delivered.

---

[5] A database driven software tool is available through the IAI Model Support Group to support the development of IFC property sets in PSD XML.

10 Documentation Integration

10.1 Ensure Existence Of Documentation

A project model should be fully documented

Integration should ensure that documentation is provided for the schema, classes, attributes, datatypes, rules, property sets and properties.

Integration

Check documentation for:-

1. Schema

   Schema level documentation should describe the scope of the schema (including restrictions on scope) and give an introduction to its content. Business cases that the schema is intended to support should be elaborated.

2. Class

   There should be a full definition for each class. Where a definition is derived from an external source, this should be quoted. If it is a definition developed specifically in connection with the IFC model, it should be designated as '*Definition from IAI*'

   Usage definitions should be given for a class.

   The history of the development of the class should be given. Where a class is varied during integration, the history should refer to current and previous versions of the IFC model and not to the project model.

3. Attribute

   There should be a full definition given for each attribute of a class. Where a definition is derived from an external source, this should be quoted.

4. Datatype

   There should be a full definition given for each select, enumerated and defined datatype. Where a definition is derived from an external source, this should be quoted.

5. Rule

   The requirement for formal rule should be the elaborated in a formal proposition.

   Each rule should be identified by a rule identifier (e.g. UR1 etc. for a unique rule, WR1 etc. for a WHERE rule).

   There should be a full definition given for each rule, whether global or on a class.

6. Property Set

   There should be a full definition given for each property set. Where a definition is derived from an external source, this should be quoted.

   Usage definitions should be given for property sets.

7. Property

   There should be a full definition given for each property. Where a definition is derived from an external source, this should be quoted.

10.2 Documentation Should Make Sense

The integration should check and ensure that the documentation makes sense and can be understood by other than the originating author(s). It is particularly important that documentation should be comprehensible to software developers.

The usage of English should be checked during the integration process to ensure that spelling and grammar help others to comprehend the documentation.

Integration

1. Check that documentation is clear and can be understood
2. Check and edit spelling and grammar as appropriate

10.3 Provide Usage Definitions

Usage definitions should be given for classes to provide guidance to software developers on the intended usage and interpretation of the class. The usage definition should attempt to minimize ambiguity in the interpretation of the class.

Integration

1. Integration should check for potential ambiguity in documentation.
2. Where ambiguity exists, it should be removed either by making the usage definition clearer or by expanding it to cover cases where ambiguity might exist.