ISO TC184/SC4/WG11 N101

Date: 1999-12-21

ISO/WD 10303-28
Product data representation and exchange: Implementation methods: XML representation of EXPRESS-driven data

COPYRIGHT NOTICE

This ISO document is a working draft of International Standard and is copyright-protected by ISO. Except as permitted under the applicable laws of the user's country, neither this ISO draft nor any extract from it may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, photocopying, recording, or otherwise, without prior written permission being secured.

Requests for permission to reproduce should be addressed to ISO at the address below or ISO's member body in the country of the requester:

Copyright Manager, ISO Central Secretariat, 1 rue de Varembe, CH-1211 Geneva 20, Switzerland
telephone: +41 22 749 0111, telefacsimile: +41 22 734 0179
Internet: central@isocs.iso.ch, X.400: c=ch; a=400net; p=iso; o=isocs; s=central

Reproduction for sales purposes for any of the above-mentioned documents may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

ABSTRACT

This part specifies the way in which XML can be used to encode both EXPRESS schemas and corresponding data.

KEYWORDS:

Implementation methods XML representation

COMMENTS TO READER:

This document is the second working draft of part 28.

It has been produced with support of the BSI CDS scheme.

In a number of areas there are comments concerning the draft status of the part (given in italic.)

Project Leader:	Nigel Shaw (acting editor)	**Project Editor**:	Robin La Fontaine
Address:	Eurostep Limited Castell, Bodfari, Denbigh LL16 4HT UK	**Address**:	Monsell EDM, Monsell House, Monsell Lane, Upton-on-Severn UK WR8 0QN
Telephone:	44 (0) 1745 710677		
Facsimile:	44 (0) 1745 710688	**Telephone**:	44 (0) 1684 592144
E-mail:	nigel.shaw@eurostep.com	**Facsimile**:	44 (0) 1684 594504
		E-mail:	robin@monsell.co.uk

Contents

Page

1 Scope 7

2 Normative references 2

3 Terms and definitions 3
3.1 Terms defined in ISO 10303-1 3
3.2 Terms defined in ISO 10303-11 3
3.3 Terms defined in ISO 8879 3
3.4 Terms defined in ISO/IEC 10744 3
3.5 Other terms and definitions 3
3.6 Abbreviations 4
3.7 Terminology 4

4 Fundamental concepts and assumptions 5
4.1 Early and Late binding 5
4.2 Use of Architectural forms 5

5 Conformances 7
5.1 Conformance of a document 7
5.2 Conformance as a pre-processor 7
5.3 Conformance as a post-processor 7

6 Late bound XML representation of EXPRESS and EXPRESS-driven data. 8
6.1 Late Bound DTD elements for XML representation of EXPRESS 8
6.2 Late Bound DTD elements for XML representation of EXPRESS 9
6.3 Late Bound DTD elements for XML representation of EXPRESS 45
6.3.1 The array_literal element 45
6.3.2 The attribute_instance element 46
6.3.3 The author element 46
6.3.4 The authorisation element 46
6.3.5 The bag_literal element 46
6.3.6 The binary_literal element 46
6.3.7 The constant_instances element 47
6.3.8 The data element 47
6.3.9 The data_section_description element 47
6.3.10 The data_section_header element 47

© ISO 1999
All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

International Organization for Standardization
Case Postale 56 • CH-1211 Genève 20 • Switzerland

6.3.11 The data_section_identification_name element ... 47
6.3.12 The data_section_name element ... 47
6.3.13 The description element ... 48
6.3.14 The entity_instance element ... 48
6.3.15 The enumeration_ref element ... 48
6.3.16 The flat_complex_entity_instance element ... 48
6.3.17 The integer_literal element ... 48
6.3.18 The ISO-10303-data element ... 48
6.3.19 The list_literal element ... 48
6.3.20 The logical_literal element ... 49
6.3.21 The nested_complex_entity_instance element ... 49
6.3.22 The nested_complex_entity_instance_subitem element ... 49
6.3.23 The non_constant_instances element ... 49
6.3.24 The organisation element ... 49
6.3.25 The originating_system element ... 50
6.3.26 The partial_entity_instance element ... 50
6.3.27 The preprocessor_version element ... 50
6.3.28 The real_literal element ... 50
6.3.29 The schema_instance element ... 50
6.3.30 The set_literal element ... 50
6.3.31 The simple_entity_instance element ... 51
6.3.32 The string_literal element ... 51
6.3.33 The time_stamp element ... 51
6.3.34 The type element ... 51

7 Early bound DTD ... 51
7.1 Fundamental concepts of the ??? ... 52
7.2 Early binding specification ... 52
7.2.1 Preconditions and limitations ... 52
7.2.2 Procedural specification ... 53

Annex A (normative) Late Bound DTD elements for EXPRESS schema ... 70

Annex B (normative) Information object registration ... 71

Annex C (informative) Object serialization early binding ... 72

Annex D (informative) Computer interpretable listings ... 83

Annex E (informative) Deterministic Content Models ... 84

Annex F (informative) Late Bound DTD correspondence with EXPRESS syntax ... 86

Annex G (informative) Examples ... 115

Annex H (tbd) EXPRESS schema interchange using the OMG XMI standard ... 144

Index ... 147

Figures

Figure 1 Relationship between DTD's 6
Figure C.1The Xchange scenario 74

Tables

Table D.1 – Late Bound DTD correspondence with EXPRESS syntax 86

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

International Standard ISO 10303-28 was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data*.

This International Standard is organized as a series of parts, each published separately. The parts of ISO 10303 fall into one of the following series: description methods, integrated resources, application interpreted constructs, application protocols, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303-1. A complete list of parts of ISO 10303 is available from the Internet:

<http://www.nist.gov/sc4/editing/step/titles/>.

This part of ISO 10303 is a member of the implementation methods series. The implementation methods specify <???>.

Annexes A, B, C and D form an integral part of this part of ISO 10303. Annexes E, F and G are for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation of product information and for the exchange of product data. The objective is to provide a neutral mechanism capable of describing products throughout their life cycle. This mechanism is suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases, and as a basis for archiving.

This part of ISO 10303 specifies means by which data and schemas specified using the EXPRESS language (ISO 10303-11) can be encoded using XML.

XML provides a basic syntax that can be used in many different ways to encode information. In this part of ISO 10303, the following uses of XML are specified:

a) A late bound XML architectural Document Type Declaration (DTD) that enables any EXPRESS schema to be encoded;

b) An extension to the late bound DTD to enable data corresponding to any EXPRESS schema to be encoded as XML;

c) A canonical form for the late bound DTD that is derived from the architectural DTD;

d) The use of SGML architectures to enable early binding XML forms to be defined that are compatible with the late binding.

The use of architectures allows for different early bindings to be defined that are compatible with each other and can be processed using the architectural DTD.

Several components of this part of ISO 10303 are available in electronic form. This access is provided through the specification of Universal Resource Locators (URLs) that identify the location of these files on the Internet. If there is difficulty accessing these files contact the ISO Central Secretariat, or contact the ISO TC 184/SC4 Secretariat directly at: sc4sec@cme.nist.gov.

Industrial automation systems and integration – Product data representation and exchange – Part 28: Implementation methods: XML representation of EXPRESS-driven data

1 Scope

This part of ISO 10303 specifies use of the Extensible Markup Language (XML) to enable the transfer of both schemas and data specified using the EXPRESS information specification language (ISO 10303-11).

The following are within the scope of this part of ISO 10303.

a) The specification of an architectural XML DTD that enables any EXPRESS schema and/or data conforming to that schema to be encoded as XML.

Note 1 This generic DTD is referred to as a late-bound DTD in that it uses an approach that is independent of the schema. It allows for a number of choices in how EXPRESS-driven data is encoded.

b) The means by which to define other XML DTD's that are wholly or partly based on an EXPRESS schema such that the correspondence between the XML elements and the architectural DTD can be identified and used.

Note 2 There are many ways in which a given EXPRESS schema can be used to define an XML DTD that can be used to encode the data described by the schema. The approach used here is not to specify a single early-bound DTD but instead to specify how architectures as defined in ISO 10744 (HyTime) can be applied to enable multiple XML DTD's.

c) The specification of a procedure by which an early bound DTD can be generated that corresponds to an EXPRESS schema and enables data conforming to that schema to be encoded as XML. This procedure generates a DTD *that preserves the naming and semantics of the EXPRESS schema.????????*

d) The specification of an Object Serialization Early Binding that *enables encoding of objects described by EXPRESS and is compatible with other object models.*

Note 3 This binding is specifically suited to inter-process communication within a multi-process computing environment.

The following are outside of the scope of this part of ISO 10303.

a) The specification of any specific mapping to XML from the EXPRESS language where the form of the mapping is dependent on the specific EXPRESS schema.

b) The specification of a mapping from an XML DTD to an EXPRESS schema.

Note 4 Given an XML DTD and one or more data sets corresponding to it, it is feasible to define an EXPRESS schema describing the data. However, this requires an understanding of the semantics of the data that may not be captured by the XML DTD.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO 10303. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 8824-1:1995, *Information technology – Open systems interconnection – Abstract syntax notation one (ASN.1) – Part 1: Specification of basic notation*.

ISO 10303-1:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles*.

ISO 10303-11:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 11: Description methods: The EXPRESS language reference manual*.

ISO 10303-11: ????, *Industrial automation systems and integration - product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual. Amendment 1* ???

ISO 8879:1986, *Information processing – Text and office systems – Standard Generalized Markup Language.*

W3C XML *What is the most authoritative source?*

ISO/IEC 10744:1997, *Information processing – Hypermedia/Time-based structuring language (HyTime)*.

OMG XMI documentation.

3 Terms and definitions

3.1 Terms defined in ISO 10303-1

For the purposes of this part of ISO 10303, the following terms defined in ISO 10303-1 apply.

— data;

— information.

3.2 Terms defined in ISO 10303-11

For the purposes of this part of ISO 10303, the following terms defined in ISO 10303-11 apply.

— entity instance;

— ???.

3.3 Terms defined in ISO 8879

For the purposes of this part of ISO 10303, the following terms defined in ISO 8879 apply.

— element;

— empty element;

— tag.

3.4 Terms defined in ISO/IEC 10744

For the purposes of this part of ISO 10303, the following terms defined in ISO/IEC 10744 apply.

— Architectural Engine;

— Architectural forms;

— Base architecture

— Client DTD.

3.5 Other terms and definitions

For the purposes of this part of ISO 10303, the following terms and definitions apply.

3.5.1
EXPRESS-driven data
Data that is known to correspond to an identified EXPRESS schema.

Note: Such data can always be transformed into a set of entity instances according to one of the ISO 10303 implementation methods.

3.5.2
Entity Data Type Reference
An XML-element that specifies the instance identifier of an element and serves as a reference *to* that element *from* the point at which the Entity Data Type Reference element appears.

Note: In other words, an Entity Data Type Reference is a "pointer" to another element instance. The Entity Data Type Reference Declaration does not correspond to any EXPRESS declaration, but rather is derived from an EXPRESS entity data type declaration. The name of Entity Data Type Reference declaration is called the Entity Data Type Reference Identifier. See **Error! Reference source not found.** for an explanation of this concept.

3.5.3
Synthetic Element Type
An XML element declaration that corresponds to an EXPRESS complex entity data type which contains multiple supertypes.

Note: See 4.6 for a further explanation of the term and the rationale behind it.

3.6 Abbreviations

For the purpose of this part of ISO 10303, the following abbreviations apply.

— DTD Document Type Declaration;

— HyTime Hypermedia/Time-based structuring language

— SGML Standard Generalized Markup Language;

— XML Extensible Markup Language.

3.7 Terminology

EXPRESS and XML use similar or identical words for different concepts. Where there is scope for confusion the prefixes of XML- and EXPRESS- are used to distinguish between the different cases. These prefixes are used for the following terms:

— Attribute.

4 Fundamental concepts and assumptions

The EXPRESS language is used to specify information. Such a specification is given as an EXPRESS schema. ISO 10303 provides multiple implementation methods that can be used for data described by means of an EXPRESS schema.

The Extensible Markup Language (XML) is a subset of SGML that is has been specified to enable generic SGML to be served, received, and processed on the World-Wide Web. It provides a syntax for describing and encoding of documents, where the content of the document may be structured information as well as or instead of free text.

This part of ISO 10303 specifies how XML can be applied to both EXPRESS schemas and data corresponding to EXPRESS schemas. It is assumed that an EXPRESS schema is available.

4.1 Early and Late binding

Given an EXPRESS schema specifying some information, it is possible to use two different approaches in defining an XML DTD for the same information. These two approaches are: Late Binding and Early Binding.

— A Late Bound DTD can be used in the same manner for any EXPRESS schema. It does not define any constructs that are specific to the schema.

— An Early Bound DTD is based on the specific schema and embeds specific aspects, such as names or structures, from the schema in the DTD.

There are many possible DTD's that can be constructed for both the late and early bindings. In this part of ISO 10303 one DTD is specified for the late bound case in clause ??. This DTD also includes the ability to represent EXPRESS schemas in XML as well as data corresponding to an EXPRESS schema. The elements used for representing data in the late-bound DTD may also be used as a base architecture for defining early bound DTD’s. Clause ?? presents a procedure for constructing early bound DTD’s that conform to the late bound DTD by means of using the architectural forms provided by ISO 10744.

Note: Annex ?? presents a different approach to generating early bound DTD’s corresponding to an EXPRESS schema. This alternative does not make use of architectural forms.

4.2 Use of Architectural forms

This part of ISO 10303 makes use of the SGML/HyTime (ISO 10744) concept of architectural forms. Given a document in XML that corresponds with a particular DTD, architectural forms provide a standard mechanism for processing it as if it were consistent with another DTD (the meta-DTD or base architecture).

This is achieved by identifying, by means of XML-attributes, the relationship between the two DTD's such that an application can recognise an element defined in one DTD as equivalent to an element in the meta-DTD and process the data according to the meta-DTD.

SGML/HyTime architectures place constraints on the extent of differences between the two DTD's. The allowed flexibility is as follows:

— Choice of names for element types;

— Choice of names for attributes;

— Choice of using attributes or content for data items;

— Ability to define additional ‘wrapper’ element types which do not appear in the base architecture;

— Ability to define extra data which does not appear at all in the base architecture.

This part of ISO 10303 defines a late bound DTD that is used as the base architecture that can be used to define multiple early-bound DTD's. This will allow early-bound data sets to be viewed as if they were defined in terms of either the late-bound DTD. Thus software written against the late bound DTD can, without modification, process data that complies with any compliant early-bound DTD.

An early-bound DTD is compliant if it has the late-bound DTD as its base architecture.

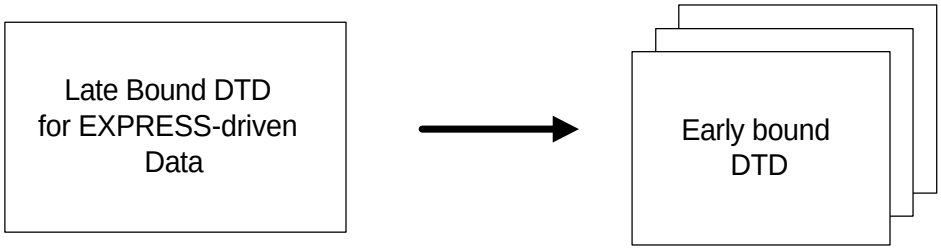

Figure 1 Relationship between DTD's

Note: Figure 1 shows the relationship between the different DTD's included and enabled by this part of ISO 10303. This approach gives flexibility in defining early-bound DTD's that can be optimised for different purposes, e.g., for display, for data exchange, for compactness.

5 Conformances

5.1 Conformance of a document

A document (or “file”) conforms to this part of this International Standard if it conforms to one of the DTDs specified herein, as defined by ISO 8879. Any document that conforms to this part of this International Standard is governed by an EXPRESS schema and shall also conform to that schema in one of the following three ways, called *conformance classes*:

Conformance class 1: A document that conveys an EXPRESS data model comprising one or more EXPRESS schemas *conforms to the EXPRESS binding* if it conveys all of the elements of that data model, encoded as specified in clause 7.

Conformance class 2: A document that conveys a data set described by an EXPRESS schema *conforms to a late binding of that EXPRESS schema* if it contains all of the information units in that data set, encoded as specified in clause 6.

Conformance class 3: A document that conveys a data set described by an EXPRESS schema *conforms to an early binding of that EXPRESS schema* if it contains all of the information units in that data set, encoded as specified in clause 5.

[SOME clause should specify the form of a document that conveys the schema AND the late-bound or early-bound data.]

5.2 Conformance as a pre-processor

A software application that produces documents that conform to this part of this International Standard as specified in 4.1 is said to conform *as a pre-processor*. An application may conform as a pre-processor for arbitrary EXPRESS schemas, or it may conform as a pre-processor for certain EXPRESS schemas only. A conforming pre-processor shall specify the classes of conforming documents it can produce and the EXPRESS schemas to which they conform.

5.3 Conformance as a post-processor

A software application is said to conform *as a post-processor* if it accepts documents that conform to this part of this International Standard as specified in 4.1 and interprets them as specified in Part 11. An application may conform as a post-processor for arbitrary EXPRESS schemas, or it may conform as a post-processor for certain EXPRESS schemas only. A conforming post-processor shall specify the classes of conforming documents it can accept and the EXPRESS schemas to which they conform.

[Whereas Part 21 and the Burkett early-binding document both describe two levels of conformance -- syntactic and "semantic" -- I think "semantic" conformance is really out of scope of Part 28 (and Part 21). The semantics of Part 28 is limited to the semantics of Part 11 and the semantics of SGML/XML. /`

6 Late bound XML representation of EXPRESS and EXPRESS-driven data.

An XML DTD's are specified that can be used to encode the following:

— One or more EXPRESS schemas;

— One or more data sets, each corresponding to an EXPRESS schema;

— A combination of EXPRESS schemas and corresponding data.

For each data set the EXPRESS schema shall always be identified although it need not be encoded as XML.

The late bound DTD is designed to act as a base architecture for the definition of early bound DTD's that can then be processed according to the late bound DTD. The design of this DTD includes options that facilitate the specification of early-bound DTD's.

The principle of the late-bound format is a single DTD for any Express model. Multiple data sections can be put into a single file and each will have data for a particular schema. All entities and attributes are referenced explicitly: The model may or may not be in the same file.

The various '_ref' elements are used for these references. The '_literal' values are defined in a way that is consistent with the language definition.

6.1 Late Bound DTD elements for XML representation of EXPRESS

The following XML element declarations are based on the syntax of the EXPRESS language (ISO 10303-11). They are specified in alphabetical order. The XML element ??? defines the root of the structure.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ISO-10303-data [
<!ELEMENT ISO_10303_data (documentation?, (schema_decl | data)*)>
]>
```

The details of the elements used to encode an EXPRESS schema (schema_decl) and a data set corresponding to an EXPRESS schema (data) are specified in this sub-clause.

```
<!ELEMENT express_driven_data ((embedded_remark | tail_remark)?,
(schema_decl* | data)*)>
<!ATTLIST express_driven_data
      express-production CDATA #FIXED "#NONE" >

<!ELEMENT data (TBA)>
<!ATTLIST data
      express-production CDATA #FIXED "#NONE" >
```

Annex ??? provides a table identifying the correspondence between the syntax of EXPRESS and the elements specified here.

The documentation element allows for the inclusion of character data as annotation of the EXPRESS and data. The documentation element is used instead of providing an XML mapping from the comment syntax of EXPRESS. ?????

6.2 Late Bound DTD elements for XML representation of EXPRESS

The details of the elements used to encode an EXPRESS schema (schema_decl) are specified in this sub-clause. The elements are listed in alphabetic order.

Each element specified in this sub-clause shall be used to represent the EXPRESS construct specified by an EXPRESS syntax production. The applicable production has the same name as the lement unless the XML-attribute express-production specifies a different name, in which case the element

shall be used to represent the construct specified by the syntax rule named in the express-production attribute value.

```
<!ELEMENT abs EMPTY>
```

```
<!ELEMENT abstract_supertype ((embedded_remark | tail_remark)?,
(entity_ref
| supertype_one_of | supertype_and_or | supertype_and)?)>
<!ATTLIST abstract_supertype

express-production CDATA #FIXED "abstract_supertype_declaration" >
```

```
<!ELEMENT acos EMPTY>
```

```
<!ELEMENT add EMPTY>
<!ATTLIST add

express-production CDATA #FIXED "+" >
```

```
<!ELEMENT aggregate_initializer ((embedded_remark | tail_remark)?,
element_list)>
```

```
<!ELEMENT aggregate_source ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression))>
```

```
<!ELEMENT aggregate_type ((embedded_remark | tail_remark)?,
(aggregate_type
| general_array_type | general_bag_type | general_list_type |
general_set_type | generic_type | entity_ref | type_ref | binary |
boolean
| integer | logical | number | real | string), (type_label_id |
type_label_ref)?)>



<!ELEMENT algorithm_head ((embedded_remark | tail_remark)?,
declaration_block?, constant_block?, local_variable_block?)>



<!ELEMENT alias_stmt ((embedded_remark | tail_remark)?, variable_id,
(parameter_ref | variable_ref), qualifier?, statement_block)>



<!ELEMENT and EMPTY>



<!ELEMENT applies_to_entities ((embedded_remark | tail_remark)?,
entity_ref+)>
<!ATTLIST applies_to_entities

 express-production CDATA #FIXED "rule_head" >



<!ELEMENT array_type ((embedded_remark | tail_remark)?, index_spec,
base_type, optional?, unique?)>



<!ELEMENT asin EMPTY>
```

```
<!ELEMENT assignment_stmt ((embedded_remark | tail_remark)?,
(parameter_ref
| variable_ref), qualifier?, (aggregate_initializer |
entity_constructor | enumeration_reference | interval | query |
binary_literal |
integer_literal | logical_literal | real_literal | string_literal |
attribute_ref | const_e | pi | self | unknown | constant_ref |
function_call | parameter_ref | variable_ref | population |
qualified_factor | bracketed_expression | unary_op | factor | term |
simple_expression | relation_expression))>
```

```
<!ELEMENT atan EMPTY>
```

```
<!ELEMENT attribute_id (#PCDATA)>
<!ATTLIST attribute_id
      id ID #IMPLIED >
```

```
<!ELEMENT attribute_ref (#PCDATA)>
<!ATTLIST attribute_ref
      refid IDREF #IMPLIED
      reftype CDATA "attribute_id" >
```

```
<!ELEMENT bag_type ((embedded_remark | tail_remark)?, bound_spec?,
base_type)>
```

```
<!ELEMENT base_type ((embedded_remark | tail_remark)?, (array_type |
bag_type | list_type | set_type | binary | boolean | integer |
logical |
number | real | string | entity_ref | type_ref))>
```

```
<!ELEMENT binary ((embedded_remark | tail_remark)?, width_spec?)>
<!ATTLIST binary

 express-production CDATA #FIXED "binary_type" >



<!ELEMENT binary_literal (#PCDATA)>




<!ELEMENT blength EMPTY>




<!ELEMENT boolean EMPTY>
<!ATTLIST boolean

 express-production CDATA #FIXED "boolean_type" >



<!ELEMENT bound_spec ((embedded_remark | tail_remark)?, lower_bound,
upper_bound)>




<!ELEMENT bracketed_expression ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self
|unknown | constant_ref | function_call | parameter_ref |
variable_ref |
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression | relation_expression))>
<!ATTLIST bracketed_expression

 express-production CDATA #FIXED "()" >
```

```
<!ELEMENT case_action ((embedded_remark | tail_remark)?, case_label,
(alias_stmt | assignment_stmt | case_stmt | statement_block |
escape_stmt |
if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt
|
skip_stmt))>



<!ELEMENT case_label ((embedded_remark | tail_remark)?,
(aggregate_initializer |
entity_constructor | enumeration_reference | interval | query |
binary_literal | integer_literal | logical_literal | real_literal |
string_literal | attribute_ref | const_e | pi | self | unknown |
constant_ref | function_call | parameter_ref | variable_ref |
population |
qualified_factor | bracketed_expression | unary_op | factor | term |
simple_expression | relation_expression)+)>



<!ELEMENT case_stmt ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression | relation_expression), case_action*,
otherwise?)>



<!ELEMENT complex_entity_constructor EMPTY>
<!ATTLIST complex_entity_constructor


express-production CDATA #FIXED "||" >



<!ELEMENT constant_block ((embedded_remark | tail_remark)?,
constant_decl*)>
<!ATTLIST constant_block
```

```
express-production CDATA #FIXED "constant_decl" >



<!ELEMENT constant_decl ((embedded_remark | tail_remark)?,
constant_id,
base_type, (aggregate_initializer | entity_constructor |
enumeration_reference | interval | query | binary_literal |
integer_literal
| logical_literal | real_literal | string_literal | attribute_ref |
const_e
| pi | self | unknown | constant_ref | function_call | parameter_ref
| variable_ref |
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression | relation_expression))>
<!ATTLIST constant_decl


express-production CDATA #FIXED "constant_body" >



<!ELEMENT constant_id (#PCDATA)>
<!ATTLIST constant_id
      id ID #IMPLIED >




<!ELEMENT constant_import ((embedded_remark | tail_remark)?,
constant_id,
constant_ref)>
<!ATTLIST constant_import


express-production CDATA #FIXED "resource_or_rename" >



<!ELEMENT constant_ref (#PCDATA)>
<!ATTLIST constant_ref
      refid IDREF #IMPLIED
      reftype CDATA "constant_id" >




<!ELEMENT const_e EMPTY>
```

```
<!ELEMENT cos EMPTY>
```

```
<!ELEMENT declaration_block ((embedded_remark | tail_remark)?,
(entity_decl
| function_decl | procedure_decl | type_decl)*)>
<!ATTLIST declaration_block

express-production CDATA #FIXED "declaration" >
```

```
<!ELEMENT derived_attr ((embedded_remark | tail_remark)?,
(attribute_id | qualified_attribute),
base_type, (aggregate_initializer | entity_constructor |
enumeration_reference | interval | query | binary_literal |
integer_literal
| logical_literal | real_literal | string_literal | attribute_ref |
const_e
| pi | self | unknown | constant_ref | function_call | parameter_ref
|
variable_ref | population | qualified_factor | bracketed_expression |
unary_op | factor | term | simple_expression | relation_expression))>
```

```
<!ELEMENT derive_clause ((embedded_remark | tail_remark)?,
derived_attr+)>
```

```
<!ELEMENT domain_rule ((embedded_remark | tail_remark)?, label?,
logical_expression)>
```

```
<!ELEMENT element_item ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor | term |
simple_expression | relation_expression), repetition?)>
<!ATTLIST element_item


express-production CDATA #FIXED "element" >



<!ELEMENT element_list ((embedded_remark | tail_remark)?,
element_item*)>
<!ATTLIST element_list


express-production CDATA #FIXED "element" >



<!ELEMENT embedded_remark (#PCDATA | embedded_remark)*>




<!ELEMENT entity_constructor ((embedded_remark | tail_remark)?,
entity_ref,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal |
integer_literal | logical_literal | real_literal | string_literal |
attribute_ref | const_e | pi | self | unknown | constant_ref |
function_call | parameter_ref | variable_ref | population |
qualified_factor | bracketed_expression | unary_op | factor | term |
simple_expression | relation_expression)*)>



<!ELEMENT entity_decl ((embedded_remark | tail_remark)?, entity_id,
(abstract_supertype | supertype_of)?, subtype_of?,
explicit_attr_block?,derive_clause?, inverse_clause?, unique_clause?,
where_clause?)>
```

```
<!ELEMENT entity_id (#PCDATA)>
<!ATTLIST entity_id
      id ID #IMPLIED >


<!ELEMENT entity_import ((embedded_remark | tail_remark)?, entity_id,
entity_ref)>
<!ATTLIST entity_import

 express-production CDATA #FIXED "resource_or_rename" >


<!ELEMENT entity_ref (#PCDATA)>
<!ATTLIST entity_ref
      refid IDREF #IMPLIED
      reftype CDATA "entity_id" >


<!ELEMENT enumeration ((embedded_remark | tail_remark)?,
enumeration_id+)>
<!ATTLIST enumeration

 express-production CDATA #FIXED "enumeration_type" >


<!ELEMENT enumeration_id (#PCDATA)>
<!ATTLIST enumeration_id
      id ID #IMPLIED >


<!ELEMENT enumeration_ref (#PCDATA)>
<!ATTLIST enumeration_ref
      refid IDREF #IMPLIED
      reftype CDATA "enumeration_id" >
```

```
<!ELEMENT enumeration_reference ((embedded_remark | tail_remark)?,
type_ref?, enumeration_ref)>



<!ELEMENT equal EMPTY>



<!ELEMENT escape_stmt EMPTY>



<!ELEMENT exists EMPTY>



<!ELEMENT exp EMPTY>



<!ELEMENT explicit_attr ((embedded_remark | tail_remark)?,
(attribute_id |
qualified_attribute), optional?, base_type)>



<!ELEMENT explicit_attr_block ((embedded_remark | tail_remark)?,
explicit_attr+)>
<!ATTLIST explicit_attr_block

express-production CDATA #FIXED "explicit_attr" >
```

```
<!ELEMENT factor (raise_to_power, (aggregate_initializer |
entity_constructor | enumeration_reference | interval | query |
binary_literal |
integer_literal | logical_literal | real_literal | string_literal |
attribute_ref | const_e | pi | self | unknown | constant_ref |
function_call | parameter_ref | variable_ref | population |
qualified_factor | bracketed_expression | unary_op),
(aggregate_initializer
| entity_constructor | enumeration_reference | interval | query |
binary_literal | integer_literal | logical_literal | real_literal |
string_literal | attribute_ref | const_e |pi | self | unknown |
constant_ref | function_call | parameter_ref |
variable_ref | population | qualified_factor | bracketed_expression |
unary_op))>



<!ELEMENT false EMPTY>



<!ELEMENT fixed EMPTY>



<!ELEMENT formal_parameter ((embedded_remark | tail_remark)?,
parameter_id,
(aggregate_type | general_array_type | general_bag_type |
general_list_type
| general_set_type | generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string))>



<!ELEMENT formal_parameter_block ((embedded_remark | tail_remark)?,
formal_parameter*)>
<!ATTLIST formal_parameter_block

 express-production CDATA #FIXED "formal_parameter" >



<!ELEMENT format EMPTY>
```

```
<!ELEMENT function_call ((embedded_remark | tail_remark)?, (abs |
acos |
asin | atan | blength | cos | exists | exp | format | hibound |
hiindex |
length | lobound | loindex | log | log2 | log10 | nvl | odd | rolesof
| sin
| sizeof | sqrt | tan | typeof | usedin | value | value_in |
value_unique |
function_ref), (aggregate_initializer |entity_constructor |
enumeration_reference | interval | query |
binary_literal | integer_literal | logical_literal | real_literal |
string_literal | attribute_ref | const_e | pi | self | unknown |
constant_ref | function_call | parameter_ref | variable_ref |
population |
qualified_factor | bracketed_expression | unary_op | factor | term |
simple_expression | relation_expression)*)>



<!ELEMENT function_decl ((embedded_remark | tail_remark)?,
function_id,
formal_parameter_block?, function_return_type, algorithm_head?,
statement_block)>



<!ELEMENT function_id (#PCDATA)>
<!ATTLIST function_id
      id ID #IMPLIED >



<!ELEMENT function_import ((embedded_remark | tail_remark)?,
function_id,
function_ref)>
<!ATTLIST function_import


express-production CDATA #FIXED "resource_or_rename" >



<!ELEMENT function_ref (#PCDATA)>
<!ATTLIST function_ref
      refid IDREF #IMPLIED
      reftype CDATA "function_id" >
```

```
<!ELEMENT function_return_type (aggregate_type | general_array_type |
general_bag_type | general_list_type | general_set_type |
generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string)>
<!ATTLIST function_return_type


 express-production CDATA #FIXED "parameter_type" >



<!ELEMENT general_array_type ((embedded_remark | tail_remark)?,
(aggregate_type | general_array_type | general_bag_type |
general_list_type
| general_set_type | generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string), bound_spec?,
optional?, unique?)>




<!ELEMENT general_bag_type ((embedded_remark | tail_remark)?,
(aggregate_type | general_array_type | general_bag_type |
general_list_type
| general_set_type | generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string), bound_spec?)>




<!ELEMENT general_list_type ((embedded_remark | tail_remark)?,
(aggregate_type | general_array_type | general_bag_type |
general_list_type
| general_set_type | generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string), bound_spec?,
unique?)>




<!ELEMENT general_set_type ((embedded_remark | tail_remark)?,
(aggregate_type | general_array_type | general_bag_type |
general_list_type
| general_set_type | generic_type | entity_ref | type_ref | binary |
boolean | integer | logical | number | real | string), bound_spec?)>
```

```
<!ELEMENT generic_type ((embedded_remark | tail_remark)?,
(type_label_id |
type_label_ref)?)>



<!ELEMENT greater_than EMPTY>



<!ELEMENT greater_than_or_equal EMPTY>



<!ELEMENT hibound EMPTY>



<!ELEMENT high_index ((embedded_remark | tail_remark)?,
(integer_literal |
numeric_expression))>
<!ATTLIST high_index

 express-production CDATA #FIXED "index_2" >



<!ELEMENT hiindex EMPTY>



<!ELEMENT if_stmt ((embedded_remark | tail_remark)?,
logical_expression,
statement_block, statement_block?)>



<!ELEMENT import_all EMPTY>
<!ATTLIST import_all
```

```
express-production CDATA #FIXED "#NONE" >


<!ELEMENT in EMPTY>



<!ELEMENT increment ((embedded_remark | tail_remark)?,
(integer_literal | numeric_expression))>



<!ELEMENT increment_control ((embedded_remark | tail_remark)?,
variable_id,
lower_bound, upper_bound, increment?)>



<!ELEMENT indeterminate EMPTY>
<!ATTLIST indeterminate

express-production CDATA #FIXED "?" >


<!ELEMENT index_qualifier ((embedded_remark | tail_remark)?,
low_index,
high_index?)>



<!ELEMENT index_spec ((embedded_remark | tail_remark)?, low_index,
high_index?)>
<!ATTLIST index_spec

express-production CDATA #FIXED "bound_spec" >


<!ELEMENT insert EMPTY>
```

```
<!ELEMENT instance_equal EMPTY>


<!ELEMENT instance_not_equal EMPTY>


<!ELEMENT integer EMPTY>
<!ATTLIST integer

 express-production CDATA #FIXED "integer_type" >


<!ELEMENT integer_divide EMPTY>
<!ATTLIST integer_divide

 express-production CDATA #FIXED "DIV" >


<!ELEMENT integer_literal (#PCDATA)>


<!ELEMENT interface_specification_block ((embedded_remark |
tail_remark)?,
(reference_from | use_from)+)>
<!ATTLIST interface_specification_block

 express-production CDATA #FIXED "interface_specification" >


<!ELEMENT interval ((embedded_remark | tail_remark)?,
(interval_low_inclusive | interval_low_exclusive), interval_item,
(interval_high_inclusive | interval_high_exclusive))>
```

```
<!ELEMENT interval_high_exclusive ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call |parameter_ref | variable_ref
| population | qualified_factor |
bracketed_expression | unary_op | factor | term |
simple_expression))>
<!ATTLIST interval_high_exclusive


 express-production CDATA #FIXED "interval_high" >



<!ELEMENT interval_high_inclusive ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression))>
<!ATTLIST interval_high_inclusive


 express-production CDATA #FIXED "interval_high" >



<!ELEMENT interval_item ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call |
parameter_ref | variable_ref | population | qualified_factor |
bracketed_expression | unary_op | factor | term |
simple_expression))>
```

```
<!ELEMENT interval_low_exclusive ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
| population | qualified_factor |
bracketed_expression | unary_op | factor | term |
simple_expression))>
<!ATTLIST interval_low_exclusive

 express-production CDATA #FIXED "interval_low" >



<!ELEMENT interval_low_inclusive ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression))>
<!ATTLIST interval_low_inclusive

 express-production CDATA #FIXED "interval_low" >



<!ELEMENT inverse_attr ((embedded_remark | tail_remark)?,
(attribute_id |
qualified_attribute), entity_ref, attribute_ref, (inverse_set |
inverse_bag)?)>




<!ELEMENT inverse_bag ((embedded_remark | tail_remark)?,
bound_spec?)>
<!ATTLIST inverse_bag

 express-production CDATA #FIXED "BAG" >
```

```
<!ELEMENT inverse_clause ((embedded_remark | tail_remark)?,
inverse_attr+)>



<!ELEMENT inverse_set ((embedded_remark | tail_remark)?,
bound_spec?)>
<!ATTLIST inverse_set

 express-production CDATA #FIXED "SET" >



<!ELEMENT label (#PCDATA)>



<!ELEMENT length EMPTY>



<!ELEMENT less_than EMPTY>



<!ELEMENT less_than_or_equal EMPTY>



<!ELEMENT like EMPTY>



<!ELEMENT list_type ((embedded_remark | tail_remark)?, bound_spec?,
base_type, unique?)>



<!ELEMENT lobound EMPTY>
```

```
<!ELEMENT local_variable_block ((embedded_remark | tail_remark)?,
local_variable_decl*)>
<!ATTLIST local_variable_block


 express-production CDATA #FIXED "local_decl" >



<!ELEMENT local_variable_decl ((embedded_remark | tail_remark)?,
variable_id, (aggregate_type | general_array_type | general_bag_type
|
general_list_type | general_set_type | generic_type | entity_ref |
type_ref
| binary | boolean | integer | logical | number | real | string),
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal |
logical_literal | real_literal | string_literal | attribute_ref |
const_e |
pi | self | unknown | constant_ref | function_call | parameter_ref |
variable_ref | population | qualified_factor | bracketed_expression |
unary_op | factor | term | simple_expression |
relation_expression)?)>
<!ATTLIST local_variable_decl


 express-production CDATA #FIXED "local_variable" >



<!ELEMENT log EMPTY>




<!ELEMENT log10 EMPTY>




<!ELEMENT log2 EMPTY>




<!ELEMENT logical EMPTY>
<!ATTLIST logical
```

```
express-production CDATA #FIXED "logical_type" >



<!ELEMENT logical_expression ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor | term | simple_expression | relation_expression))>




<!ELEMENT logical_literal ((embedded_remark | tail_remark)?, (false |
true
| unknown))>




<!ELEMENT loindex EMPTY>




<!ELEMENT lower_bound ((embedded_remark | tail_remark)?,
(integer_literal |
numeric_expression))>
<!ATTLIST lower_bound


express-production CDATA #FIXED "bound_1" >



<!ELEMENT low_index ((embedded_remark | tail_remark)?,
(integer_literal |
numeric_expression))>
<!ATTLIST low_index


express-production CDATA #FIXED "index_1" >



<!ELEMENT mod EMPTY>
```

```
<!ELEMENT multiply EMPTY>
<!ATTLIST multiply

 express-production CDATA #FIXED "*" >


<!ELEMENT negate EMPTY>
<!ATTLIST negate

 express-production CDATA #FIXED "-" >


<!ELEMENT not EMPTY>


<!ELEMENT not_equal EMPTY>


<!ELEMENT null_stmt EMPTY>


<!ELEMENT number EMPTY>


<!ELEMENT numeric_expression ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
| real_literal | string_literal | attribute_ref | const_e | pi |
self | unknown | constant_ref | function_call | parameter_ref |
variable_ref | population | qualified_factor | bracketed_expression |
unary_op | factor | term | simple_expression))>


<!ELEMENT nvl EMPTY>
```

```
<!ELEMENT odd EMPTY>


<!ELEMENT optional EMPTY>


<!ELEMENT or EMPTY>


<!ELEMENT otherwise ((embedded_remark | tail_remark)?, (alias_stmt |
assignment_stmt | case_stmt | statement_block | escape_stmt | if_stmt
|
null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
skip_stmt))>


<!ELEMENT parameter_id (#PCDATA)>
<!ATTLIST parameter_id
      id ID #IMPLIED >


<!ELEMENT parameter_ref (#PCDATA)>
<!ATTLIST parameter_ref
      refid IDREF #IMPLIED
      reftype CDATA "parameter_id" >


<!ELEMENT pi EMPTY>


<!ELEMENT plus EMPTY>
<!ATTLIST plus
```

```
express-production CDATA #FIXED "+" >


<!ELEMENT population ((embedded_remark | tail_remark)?, entity_ref)>



<!ELEMENT precision_spec ((embedded_remark | tail_remark)?,
(integer_literal | numeric_expression))>



<!ELEMENT procedure_call_stmt ((embedded_remark | tail_remark)?,
((insert |
remove | procedure_ref), (aggregate_initializer | entity_constructor
|
enumeration_reference | interval | query | binary_literal |
integer_literal
| logical_literal | real_literal | string_literal | attribute_ref |
const_e
| pi | self | unknown | constant_ref | function_call | parameter_ref
|
variable_ref | population | qualified_factor | bracketed_expression |
unary_op | factor | term | simple_expression |
relation_expression)*))>



<!ELEMENT procedure_decl ((embedded_remark | tail_remark)?,
procedure_id,
procedure_formal_parameter_block?, algorithm_head?,
statement_block?)>



<!ELEMENT procedure_formal_parameter_block ((embedded_remark |
tail_remark)?, (formal_parameter | var_formal_parameter)*)>
<!ATTLIST procedure_formal_parameter_block


express-production CDATA #FIXED "formal_parameter" >


<!ELEMENT procedure_id (#PCDATA)>
<!ATTLIST procedure_id
      id ID #IMPLIED >
```

```
<!ELEMENT procedure_import ((embedded_remark | tail_remark)?,
procedure_id, procedure_ref)>
<!ATTLIST procedure_import


 express-production CDATA #FIXED "resource_or_rename" >



<!ELEMENT procedure_ref (#PCDATA)>
<!ATTLIST procedure_ref
      refid IDREF #IMPLIED
      reftype CDATA "procedure_id" >




<!ELEMENT qualified_attribute ((embedded_remark | tail_remark)?,
entity_ref, attribute_ref)>




<!ELEMENT qualified_factor ((embedded_remark | tail_remark)?,
(attribute_ref | const_e | pi | self | unknown | constant_ref |
function_call |parameter_ref | variable_ref | population),
qualifier)>
<!ATTLIST qualified_factor


 express-production CDATA #FIXED "qualifiable_factor" >



<!ELEMENT qualifier ((embedded_remark | tail_remark)?, (attribute_ref
|
entity_ref | index_qualifier)*)>




<!ELEMENT query ((embedded_remark | tail_remark)?, variable_id,
(aggregate_source, logical_expression))>
<!ATTLIST query


 express-production CDATA #FIXED "query_expression" >
```

```
<!ELEMENT raise_to_power EMPTY>
<!ATTLIST raise_to_power

 express-production CDATA #FIXED "**" >



<!ELEMENT real ((embedded_remark | tail_remark)?, precision_spec?)>
<!ATTLIST real

 express-production CDATA #FIXED "real_type" >



<!ELEMENT real_divide EMPTY>
<!ATTLIST real_divide

 express-production CDATA #FIXED "/" >



<!ELEMENT real_literal (#PCDATA)>




<!ELEMENT reference_from ((embedded_remark | tail_remark)?,
schema_ref,
(import_all | (constant_import | entity_import | function_import |
procedure_import | type_import)+))>
<!ATTLIST reference_from

 express-production CDATA #FIXED "reference_clause" >
```

```
<!ELEMENT relation_expression ((embedded_remark | tail_remark)?,
(less_than
| greater_than | less_than_or_equal | greater_than_or_equal |
not_equal |
equal | instance_not_equal | instance_equal | in | like),
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
| population | qualified_factor |
bracketed_expression | unary_op | factor | term | simple_expression),
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor |
term | simple_expression))>
<!ATTLIST relation_expression


 express-production CDATA #FIXED "expression" >



<!ELEMENT remove EMPTY>




<!ELEMENT repeat_control ((embedded_remark | tail_remark)?,
increment_control, while?, until?)>




<!ELEMENT repeat_stmt ((embedded_remark | tail_remark)?,
repeat_control,
statement_block)>




<!ELEMENT repetition ((embedded_remark | tail_remark)?,
(integer_literal |
numeric_expression))>
```

```
<!ELEMENT return_stmt ((embedded_remark | tail_remark)?,
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal |
logical_literal | real_literal | string_literal | attribute_ref |
const_e |
pi | self | unknown | constant_ref | function_call | parameter_ref |
variable_ref | population | qualified_factor | bracketed_expression |
unary_op | factor | term | simple_expression |
relation_expression)?)>



<!ELEMENT rolesof EMPTY>



<!ELEMENT rule_decl ((embedded_remark | tail_remark)?, rule_id,
applies_to_entities, algorithm_head?, statement_block?,
where_clause)>



<!ELEMENT rule_id (#PCDATA)>



<!ELEMENT schema_decl ((embedded_remark | tail_remark)?, schema_id,
interface_specification_block?, constant_block?, (entity_decl |
function_decl | procedure_decl | type_decl | rule_decl)*)>



<!ELEMENT schema_id (#PCDATA)>
<!ATTLIST schema_id
      id ID #IMPLIED >



<!ELEMENT schema_ref (#PCDATA)>
<!ATTLIST schema_ref
      refid IDREF #IMPLIED
      reftype CDATA "schema_id" >
```

```
<!ELEMENT select ((embedded_remark | tail_remark)?, (entity_ref |
type_ref)+)>
<!ATTLIST select

 express-production CDATA #FIXED "select_type" >



<!ELEMENT self EMPTY>




<!ELEMENT set_type ((embedded_remark | tail_remark)?, bound_spec?,
base_type)>




<!ELEMENT simple_expression ((embedded_remark | tail_remark)?, (add |
subtract | or | xor), (aggregate_initializer | entity_constructor |
enumeration_reference | interval | query | binary_literal |
integer_literal
| logical_literal | real_literal | string_literal | attribute_ref |
const_e
| pi | self | unknown | constant_ref | function_call | parameter_ref
| variable_ref | population | qualified_factor |
bracketed_expression | unary_op | factor | term),
(aggregate_initializer |
entity_constructor | enumeration_reference | interval | query |
binary_literal | integer_literal | logical_literal | real_literal |
string_literal | attribute_ref | const_e | pi | self | unknown |
constant_ref | function_call | parameter_ref | variable_ref |
population |
qualified_factor | bracketed_expression | unary_op | factor | term))>




<!ELEMENT sin EMPTY>




<!ELEMENT sizeof EMPTY>
```

```
<!ELEMENT skip_stmt EMPTY>



<!ELEMENT sqrt EMPTY>



<!ELEMENT statement_block ((embedded_remark | tail_remark)?,
(alias_stmt |
assignment_stmt | case_stmt | statement_block | escape_stmt | if_stmt
|
null_stmt | procedure_call_stmt | repeat_stmt | return_stmt |
skip_stmt)+)>
<!ATTLIST statement_block


express-production CDATA #FIXED "stmt" >



<!ELEMENT string ((embedded_remark | tail_remark)?, width_spec?)>
<!ATTLIST string


express-production CDATA #FIXED "string_type" >



<!ELEMENT string_literal (#PCDATA)>



<!ELEMENT subtract EMPTY>
<!ATTLIST subtract


express-production CDATA #FIXED "-" >



<!ELEMENT subtype_of ((embedded_remark | tail_remark)?, entity_ref+)>
<!ATTLIST subtype_of
```

```
express-production CDATA #FIXED "subtype_declaration" >



<!ELEMENT supertype_and ((embedded_remark | tail_remark)?,
(entity_ref |
supertype_one_of | supertype_and_or | supertype_and)+)>
<!ATTLIST supertype_and


express-production CDATA #FIXED "supertype_rule" >



<!ELEMENT supertype_and_or ((embedded_remark | tail_remark)?,
(entity_ref | supertype_one_of |
supertype_and_or | supertype_and)+)>
<!ATTLIST supertype_and_or


express-production CDATA #FIXED "supertype_rule" >



<!ELEMENT supertype_of ((embedded_remark | tail_remark)?, (entity_ref
|
supertype_one_of | supertype_and_or | supertype_and))>
<!ATTLIST supertype_of


express-production CDATA #FIXED "supertype_rule" >



<!ELEMENT supertype_one_of ((embedded_remark | tail_remark)?,
(entity_ref |
supertype_one_of | supertype_and_or | supertype_and)+)>
<!ATTLIST supertype_one_of


express-production CDATA #FIXED "one_of" >



<!ELEMENT tail_remark (#PCDATA)*>



<!ELEMENT tan EMPTY>
```

```
<!ELEMENT term ((multiply | real_divide | integer_divide | mod | and
|
complex_entity_constructor), (aggregate_initializer |
entity_constructor |
enumeration_reference | interval | query | binary_literal |
integer_literal
| logical_literal | real_literal | string_literal | attribute_ref |
const_e | pi | self | unknown | constant_ref |
function_call | parameter_ref | variable_ref | population |
qualified_factor | bracketed_expression | unary_op | factor | term),
(aggregate_initializer | entity_constructor | enumeration_reference |
interval | query | binary_literal | integer_literal | logical_literal
|
real_literal | string_literal | attribute_ref | const_e | pi | self |
unknown | constant_ref | function_call | parameter_ref | variable_ref
|
population | qualified_factor | bracketed_expression | unary_op |
factor | term))>




<!ELEMENT true EMPTY>




<!ELEMENT typeof EMPTY>




<!ELEMENT type_decl ((embedded_remark | tail_remark)?, type_id,
underlying_type, where_clause?)>




<!ELEMENT type_id (#PCDATA)>
<!ATTLIST type_id
      id ID #IMPLIED >




<!ELEMENT type_import ((embedded_remark | tail_remark)?, type_id,
type_ref)>
<!ATTLIST type_import
```

```
express-production CDATA #FIXED "resource_or_rename" >


<!ELEMENT type_label_id (#PCDATA)>
<!ATTLIST type_label_id
      id ID #IMPLIED >


<!ELEMENT type_label_ref (#PCDATA)>
<!ATTLIST type_label_ref
      refid IDREF #IMPLIED
      reftype CDATA "type_label_id" >


<!ELEMENT type_ref (#PCDATA)>
<!ATTLIST type_ref
      refid IDREF #IMPLIED
      reftype CDATA "type_id" >


<!ELEMENT unary_op ((embedded_remark | tail_remark)?, (plus | negate
|
not), (binary_literal | integer_literal | logical_literal |
real_literal |
string_literal | attribute_ref | const_e | pi | self | unknown |
constant_ref | function_call | parameter_ref | variable_ref
|population | qualified_factor | bracketed_expression))>


<!ELEMENT underlying_type ((embedded_remark | tail_remark)?,
(enumeration |
select | array_type | bag_type | list_type | set_type | binary |
boolean |
integer | logical | number | real | string | type_ref))>


<!ELEMENT unique EMPTY>
```

```
<!ELEMENT unique_clause ((embedded_remark | tail_remark)?,
unique_rule+)>



<!ELEMENT unique_rule ((embedded_remark | tail_remark)?, label?,
(attribute_ref | qualified_attribute)+)>



<!ELEMENT unknown EMPTY>



<!ELEMENT until ((embedded_remark | tail_remark)?,
logical_expression)>
<!ATTLIST until

 express-production CDATA #FIXED "until_control" >


<!ELEMENT upper_bound ((embedded_remark | tail_remark)?,
(indeterminate |
integer_literal | numeric_expression))>
<!ATTLIST upper_bound

 express-production CDATA #FIXED "bound_2" >


<!ELEMENT usedin EMPTY>



<!ELEMENT use_from ((embedded_remark | tail_remark)?, schema_ref,
(import_all | (entity_import | type_import)+))>
<!ATTLIST use_from

 express-production CDATA #FIXED "use_clause" >
```

```
<!ELEMENT value EMPTY>



<!ELEMENT value_in EMPTY>



<!ELEMENT value_unique EMPTY>



<!ELEMENT variable_id (#PCDATA)>
<!ATTLIST variable_id
      id ID #IMPLIED >



<!ELEMENT variable_ref (#PCDATA)>
<!ATTLIST variable_ref
      refid IDREF #IMPLIED
      reftype CDATA "variable_id" >



<!ELEMENT var_formal_parameter ((embedded_remark | tail_remark)?,
parameter_id, (aggregate_type | general_array_type | general_bag_type
|
general_list_type | general_set_type | generic_type | entity_ref
|type_ref | binary | boolean | integer | logical | number | real |
string))>
<!ATTLIST var_formal_parameter


 express-production CDATA #FIXED "formal_parameter" >



<!ELEMENT where_clause ((embedded_remark | tail_remark)?,
domain_rule+)>
```

```
<!ELEMENT while ((embedded_remark | tail_remark)?,
logical_expression)>
<!ATTLIST while

 express-production CDATA #FIXED "while_control" >



<!ELEMENT width_spec ((embedded_remark | tail_remark)?,
(integer_literal |
numeric_expression), fixed?)>




<!ELEMENT xor EMPTY>
```

6.3 Late Bound DTD elements for XML representation of EXPRESS

The details of the elements used to encode a data set corresponding to an EXPRESS schema (data) are specified in this sub-clause.

This clause specifies DTD elements necessary for the late-bound XML representation of EXPRESS-driven data. It uses some elements specified in clause 6.2.

Note: The DTD allows for two different ways to handle instances of EXPRESS entities that are part of a supertype/subtype hierarchy: these use the nested_complex_entity_instance and flat_complex_entity_instance.

6.3.1 The array_literal element

```
<!ELEMENT array_literal (binary_literal | integer_literal |
logical_literal
| real_literal | string_literal | bag_literal | list_literal |
set_literal
| array_literal | type_literal | nested_complex_entity_instance |
flat_complex_entity_instance | simple_entity_instance |
entity_instance_ref
| unknown)*>
```

6.3.2 The attribute_instance element

```
<!ELEMENT attribute_instance (binary_literal | integer_literal |
logical_literal | real_literal | string_literal | bag_literal |
list_literal | set_literal | array_literal | type_literal |
nested_complex_entity_instance | flat_complex_entity_instance |
simple_entity_instance | entity_instance_ref)>
<!ATTLIST attribute_instance
      express_attribute_name NMTOKEN #REQUIRED >
```

6.3.3 The author element

```
<!ELEMENT author (#PCDATA)>
```

6.3.4 The authorisation element

```
<!ELEMENT authorisation (#PCDATA)>
```

6.3.5 The bag_literal element

```
<!ELEMENT bag_literal (binary_literal* | integer_literal* |
logical_literal* | real_literal* | string_literal* | bag_literal* |
list_literal* |
set_literal* | array_literal* | type_literal* |
(nested_complex_entity_instance | flat_complex_entity_instance |
simple_entity_instance | entity_instance_ref)*)>
```

6.3.6 The binary_literal element

```
<!ELEMENT binary_literal (#PCDATA)>
<!ATTLIST binary_literal
      notation  (hex | base64) #REQUIRED >
```

6.3.7 The constant_instances element

```
<!ELEMENT constant_instances (nested_complex_entity_instance |
flat_complex_entity_instance | simple_entity_instance)*>
```

6.3.8 The data element

```
<!ELEMENT data (data_section_header?, schema_instance)>
<!ATTLIST data
      data_id ID #REQUIRED >
```

6.3.9 The data_section_description element

```
<!ELEMENT data_section_description (description*)>
```

6.3.10 The data_section_header element

```
<!ELEMENT data_section_header (data_section_description,
data_section_name)>
```

6.3.11 The data_section_identification_name element

```
<!ELEMENT data_section_identification_name (#PCDATA)>
```

6.3.12 The data_section_name element

```
<!ELEMENT data_section_name (data_section_identification_name,
time_stamp,
author, organisation, preprocessor_version, originating_system,
authorisation)>
```

6.3.13 The description element

```
<!ELEMENT description (#PCDATA)>
```

6.3.14 The entity_instance element

```
<!ELEMENT entity_instance_ref EMPTY>
<!ATTLIST entity_instance_ref
      entity_instance_idref IDREF #REQUIRED >
```

6.3.15 The enumeration_ref element

```
<!ELEMENT enumeration_ref (#PCDATA)>
```

6.3.16 The flat_complex_entity_instance element

```
<!ELEMENT flat_complex_entity_instance (partial_entity_instance+)>
<!ATTLIST flat_complex_entity_instance
      entity_instance_id ID #REQUIRED >
```

6.3.17 The integer_literal element

```
<!ELEMENT integer_literal (#PCDATA)>
```

6.3.18 The ISO-10303-data element

```
<!ELEMENT ISO-10303-data (data+)>
```

6.3.19 The list_literal element

```
<!ELEMENT list_literal (binary_literal* | integer_literal* |
logical_literal* | real_literal* | string_literal* | bag_literal* |
list_literal* | set_literal* | array_literal* | type_literal* |
(nested_complex_entity_instance | flat_complex_entity_instance |
simple_entity_instance | entity_instance_ref)*)>
```

6.3.20 The logical_literal element

```
<!ELEMENT logical_literal (false | true | unknown)>
```

6.3.21 The nested_complex_entity_instance element

```
<!ELEMENT nested_complex_entity_instance (attribute_instance*,
nested_complex_entity_instance_subitem*)>
<!ATTLIST nested_complex_entity_instance
      express_entity_name NMTOKEN #REQUIRED
      express_schema_name NMTOKEN #IMPLIED >
<!ATTLIST nested_complex_entity_instance
      entity_instance_id ID #REQUIRED >
```

6.3.22 The nested_complex_entity_instance_subitem element

```
<!ELEMENT nested_complex_entity_instance_subitem
(attribute_instance*,
nested_complex_entity_instance_subitem*)>
<!ATTLIST nested_complex_entity_instance_subitem
      express_entity_name NMTOKEN #REQUIRED
      express_schema_name NMTOKEN #IMPLIED
      entity_instance_id ID #IMPLIED >
```

6.3.23 The non_constant_instances element

```
<!ELEMENT non_constant_instances (nested_complex_entity_instance |
flat_complex_entity_instance | simple_entity_instance)*>
```

6.3.24 The organisation element

```
<!ELEMENT organisation (#PCDATA)>
```

6.3.25 The originating_system element

```
<!ELEMENT originating_system (#PCDATA)>
```

6.3.26 The partial_entity_instance element

```
<!ELEMENT partial_entity_instance (attribute_instance*)>
<!ATTLIST partial_entity_instance
      express_entity_name NMTOKEN #REQUIRED
      express_schema_name NMTOKEN #IMPLIED
      entity_instance_id ID #IMPLIED >
```

6.3.27 The preprocessor_version element

```
<!ELEMENT preprocessor_version (#PCDATA)>
```

6.3.28 The real_literal element

```
<!ELEMENT real_literal (#PCDATA)>
```

6.3.29 The schema_instance element

```
<!ELEMENT schema_instance (constant_instances,
non_constant_instances)>
<!ATTLIST schema_instance
      express_schema_name NMTOKEN #REQUIRED >
```

6.3.30 The set_literal element

```
<!ELEMENT set_literal (binary_literal* | integer_literal* |
logical_literal* | real_literal* | string_literal* | bag_literal* |
list_literal* | set_literal* | array_literal* | type_literal* |
(nested_complex_entity_instance | flat_complex_entity_instance |
simple_entity_instance | entity_instance_ref)*)>
```

6.3.31 The simple_entity_instance element

```
<!ELEMENT simple_entity_instance (attribute_instance*)>
<!ATTLIST simple_entity_instance
      express_entity_name NMTOKEN #REQUIRED
      express_schema_name NMTOKEN #IMPLIED
      entity_instance_id ID #REQUIRED >
```

6.3.32 The string_literal element

```
<!ELEMENT string_literal (#PCDATA)>
```

6.3.33 The time_stamp element

```
<!ELEMENT time_stamp (#PCDATA)>
```

6.3.34 The type element

```
<!ELEMENT type_literal (binary_literal | integer_literal |
logical_literal
| real_literal | string_literal | bag_literal | list_literal |
set_literal
| array_literal | type_literal | nested_complex_entity_instance |
flat_complex_entity_instance | simple_entity_instance |
entity_instance_ref
| enumeration_ref)>
<!ATTLIST type_literal
      express_type_name NMTOKEN #REQUIRED
      express_schema_name NMTOKEN #IMPLIED >
```

7 Early bound DTD

This clause presents a procedure by which early-bound DTD's corresponding to an EXPRESS schema may be generated such that the results are architecturally compatible with the late bound DTD defined in clause 5.

7.1 Fundamental concepts of the ???

XML principally establishes relationships among tagged elements through embedded structure. In order to mirror the referencing behavior of EXPRESS, this specification introduces the convention of an artificial, empty XML element that serves as the "pointer" from one element (i.e., an XML encoding of an instance of an EXPRESS entity data type) to another. This artificial element is created by appending the Entity Data Type identifier with the characters "-ref". The new element is referred to as an Entity Data Type Reference Identifier.

NOTE: A hyphen was chosen as a delimiter to prevent possible name conflicts with EXPRESS identifiers. Since hyphens are not legal characters in EXPRESS identifiers, but are legal in XML, there is no possibility that the name created for the Entity Data Type Reference element declaration will clash with an EXPRESS identifier.

EXAMPLE The following is an example of how the "pointing behavior" of EXPRESS instances is mirrored in the early-bound XML mapping. For the EXPRESS declarations:

```
ENTITY product;
END_ENTITY;
ENTITY product_definition_formation;
  of_product : product;
END_ENTITY;
```

The XML declaration is:

```
<!ELEMENT product EMPTY>
<!ATTLIST product
  id ID #REQUIRED>
<!ELEMENT product-ref EMPTY>
<!ATTLIST product-ref
  refid IDREF #REQUIRED>

<!ELEMENT product_definition_formation
(product_definition_formation.of_product)>
<!ELEMENT product_definition_formation.of_product (product-ref)>
```

7.2 Early binding specification

7.2.1 Preconditions and limitations

Use/apply outline numbered procedural steps that process an EXPRESS schema and produce a set of XML declarations.

The following preconditions are assumed:

(1) Syntactically correct EXPRESS schema;

(2) The schema shall not contain EXPRESS identifiers that begin with the characters "xml".

Note: Interface Specifications and Constants are two features of EXPRESS that are not handled in the current algorithm, yet affect the instance population and should be part of the algorithm. Once questions concerning the meaning of these features are resolved, then the algorithm will be modified to accommodate them. ???

7.2.2 Procedural specification

The following procedure assumes that a DTD file is being constructed and that the outputs of the actions will add to or modify the contents of this DTD file.

1 Preparatory actions

1.1 Insert declarations for simple types into DTD (output file):

```
<!ELEMENT real  (#PCDATA)>
```

```
<!ATTLIST real  precision CDATA  #IMPLIED >

<!ELEMENT integer  (#PCDATA)>

<!ELEMENT number  (real | integer)>

<!ELEMENT logical EMPTY>
<!ATTLIST logical  value  (false | unknown | true )  #REQUIRED >

<!ELEMENT boolean  EMPTY>
<!ATTLIST boolean  value  (false | true )  #REQUIRED >

<!ELEMENT string  (#PCDATA )>
<!ATTLIST string  width CDATA  #IMPLIED
                  fixed (fixed | not-fixed) "not-fixed" >

<!ELEMENT binary  (#PCDATA )>
<!ATTLIST binary  width   CDATA    #IMPLIED
                  fixed   (fixed | not-fixed) "not-fixed">
```

NOTE – These simple type declarations could comprise a separate, standardized DTD apart from this specification.

2 Step 2: For **SCHEMA** declaration:

2.1 create an ELEMENT declaration:

2.1.1 the ELEMENT name is the **SCHEMA** identifier (i.e,. the schema name) appended with the string "-schema";

NOTE - the appended string is necessary to disambiguate possible clashes of schema name and entity or type names.

2.1.2 create content model and add the identifiers:

of all entity data types in the EXPRESS schema that are not subtypes, all synthetic element types created to resolve multiple supertypes, to the content model of the **SCHEMA** ELEMENT separated with the XML choice operator

2.1.3 Add the zero-or-more cardinality operator ("*") to the schema content model. This denotes that zero or more entity instances may be contained in the schema.

EXAMPLE For the schema:

```
SCHEMA application_content_schema;
ENTITY entity_1; END_ENTITY;
ENTITY entity_2; END_ENTITY;
END_SCHEMA;
```

the corresponding XML element declaration is

```
<!ELEMENT application_context_schema-schema ( (entity_1 |
entity_2)*)>
```

2.2 create an `ATTLIST` declaration for the **`SCHEMA`** `ELEMENT`

2.2.1 create an XML attribute called “id” which is of XML type ID and #REQUIRED;

2.2.2 create an XML attribute called “version” which of XML type CDATA which is #FIXED and has a value assigned according to an external convention.

2.2.3 Create an XML attribute called " express_source_concept_type", which is of XML type CDATA which is #FIXED and has a value of “schema”, and "express_name", which is of type CDATA, is #FIXED, and has a value equal to the unqualified schema name.

EXAMPLE For the application_context_schema, the ATTLIST declaration would be:

```
<!ATTLIST application_context_schema
  id ID #REQUIRED
  version CDATA #FIXED " ... "
  express_source_concept_type CDATA #FIXED ”schema”
  express_name CDATA #FIXED "application_context_schema">
```

2.3 Add the ELEMENT and ATTLIST declarations to the output DTD

3 For each EXPRESS TYPE declaration:

3.1 If it is not a constructed type (i.e., it is not an ENUMERATION TYPE or a SELECT TYPE):

3.1.1 Create an ELEMENT declaration where the identifier of the element is the same as the identifier of the TYPE.

3.1.2 Create an ATTLIST declaration for the TYPE ELEMENT with the XML attribute “express_source_concept_type” which is of XML type CDATA which is #FIXED and has a value of “defined type”, , and "express_name", which is of type CDATA, is #FIXED, and has a value equal to the unqualified TYPE name.

3.1.3 Depending on the underlying type:

NOTE - the underlying type of an EXPRESS TYPE cannot be an entity identifier.

3.1.3.1 If the underlying type is a simple type, add the corresponding simple type ELEMENT identifier to the content model of the TYPE element declaration.

EXAMPLE For the EXPRESS declaration:

```
TYPE label = STRING;
END_TYPE;
```

the corresponding XML element declaration is

```
<!ELEMENT label (string)>
<!ATTLIST label
  source_express_concept_type CDATA #FIXED "defined-type"
  express_name CDATA #FIXED "label">
```

3.1.3.2 If the underlying type is a defined type, enumeration type, or select type, add the TYPE ELEMENT identifier of the underlying type to the content model of the TYPE ELEMENT declaration.

EXAMPLE For the EXPRESS declarations:

```
TYPE label = STRING; END_TYPE;
TYPE position = label; END_TYPE;
```

the XML element declaration for the second TYPE is

```
<!ELEMENT position (label)>
<!ATTLIST position
  source_express_concept_type CDATA #FIXED
    "entity-data-type"
  express_name CDATA #FIXED "position">
```

3.1.3.3 If the underlying type is an aggregate:

3.1.3.3.1 Add an XML attribute to the ATTLIST for the TYPE called “aggregate_type” of type CDATA with a #FIXED value corresponding to the aggregate type name (i.e., “set”, “list”, "bag", "array").

3.1.3.3.2 If the aggregate is a SET, BAG, or LIST, add XML attributes to the ATTLIST for the TYPE ELEMENT called “min_bound” and “max_bound” of type CDATA with a #FIXED value corresponding to the bound specs of the aggregate.

3.1.3.3.3 If the aggregate is a LIST, add an XML attribute called "list_unique" to the ATTLIST with value choice "(true | false)" and default " 'false' ".

3.1.3.3.4 If the aggregate is an ARRAY, add XML attributes to the ATTLIST for the TYPE ELEMENT called “low_index” and “high_index” of type CDATA with a #FIXED value corresponding to the bound specs of the aggregate. Add the XML attributes "array_optional" and "array_unique" with value choice "(true | false)" and default " 'false' ".

3.1.3.3.5 Add the appropriate XML cardinality indicator (i.e., +, *) to the content model of the Defined Type ELEMENT.

EXAMPLE For the EXPRESS declaration:

```
TYPE names = SET [1:3] OF STRING;
END_TYPE;
```

the corresponding XML element declaration is

```
<!ELEMENT names (string+)>
<!ATTLIST names
   source_express_concept_type CDATA
     #FIXED "defined-type"
   express_name CDATA #FIXED "names"
 aggregate_type CDATA #FIXED “set”
 min_bound CDATA #FIXED “1”
 max_bound CDATA #FIXED “3”>
```

Description	EXPRESS Cardinality	XML Cardinality Operator
One or More	[1:?], [1:n], [n:m]. [n:?]	+
Zero or More	[0:?], [0:n]	*

NOTE – Because EXPRESS ARRAYS always contain at least one element, the cardinality of an ARRAY is always One or More (thus requiring a "+" XML operator).

3.1.3.3.6 If the base type of the aggregate is not another aggregate and: (1) is a simple type, use 3.1.3.1 to complete the definition of the underlying type; (2) is a declared TYPE, use 3.1.3.2 to complete the definition of the underlying type; (3) is an entity, add the entity data type reference identifier to the content model of the TYPE element declaration.

3.1.3.3.7 If base type of the aggregate is another aggregate, create an aggregate ELEMENT declaration with an identifier equivalent to the TYPE ELEMENT identifier with the string “-item” appended. Add this ELEMENT identifier to the content model of the TYPE ELEMENT declaration and go to 3.1.3 and process the new “-item” ELEMENT as if it were a TYPE. Add "source_express_concept_type" attribute to ATTLIST of aggregate element declaration with value of "aggregate-element". No "express_name" attribute is specified for unnamed aggregate elements.

EXAMPLE For the EXPRESS declaration:

```
TYPE matrix = ARRAY [1:3] OF
ARRAY[1:4] OF REAL; END_TYPE;
```

the corresponding XML element declaration is

```
<!ELEMENT matrix (matrix-item+)>
<!ATTLIST matrix
   source_express_concept_type CDATA
     #FIXED "defined-type"
   express_name CDATA #FIXED
"matrix"
 aggregate_type CDATA #FIXED “array”
 low_index CDATA #FIXED “1”
high_index CDATA #FIXED “3”
array_unique (true | false) "false"
array_optional (true | false) "false">

<!ELEMENT matrix-item (real+)>
<!ATTLIST matrix-item
   source_express_concept_type CDATA
     #FIXED "aggregate-element"
 aggregate_type CDATA #FIXED “array”
 low_index CDATA #FIXED “1”
high_index CDATA #FIXED “4”
 array_unique (true | false) "false"
array_optional (true | false) "false">
```

3.2 If the EXPRESS TYPE declaration is an ENUMERATION Type:

3.2.1 Create an ELEMENT declaration where the identifier of the element is the same as the Enumeration TYPE identifier.

3.2.2 Replace content model with XML keyword EMPTY (i.e., make the declaration an EMPTY ELEMENT).

3.2.3 Create an ATTLIST declaration for the Enumeration TYPE ELEMENT with the following XML attributes:

3.2.3.1 “source_express_concept_type”, which is of XML type CDATA which is #FIXED and has a value of "enumeration type",

3.2.3.2 "express_name", which is of XML type CDATA which is #FIXED and has a value of the EXPRESS TYPE identifier.

3.2.3.3 “value” to the ATTLIST for the Enumeration TYPE ELEMENT with a choice of the enumeration values.

EXAMPLE For the EXPRESS declaration:

```
TYPE colour = ENUMERATION OF (red, green, blue);
END_TYPE;
```

the corresponding XML element declaration is

```
<!ELEMENT colour EMPTY>
<!ATTLIST colour
      source_express_concept_type CDATA #FIXED
        "enumeration-type"
      express_name CDATA #FIXED "colour"
```

```
value (red | green | blue) #REQUIRED>
```

3.3 If the EXPRESS TYPE Declaration is a select TYPE,

3.3.1 Create an ELEMENT declaration where the identifier of the element is the same as the Select TYPE identifier.

3.3.2 Create an ATTLIST declaration for the select TYPE ELEMENT with the following XML attributes:

3.3.2.1 “source_express_concept_type”, which is of XML type CDATA which is #FIXED and has a value of “select type”,

3.3.2.2 “express_name”, which is of XML type CDATA which is #FIXED and has a value of the EXPRESS TYPE identifier.

3.3.3 For each identifier in the EXPRESS Select list:

3.3.3.1 If the identifier is that of an EXPRESS entity data type, add the entity data type reference identifier to the content model of the select TYPE ELEMENT declaration.

3.3.3.2 If the identifier is that of an EXPRESS Type, add its identifier to the content model of the select TYPE ELEMENT declaration.

3.3.3.3 If there are two or more elements in the select list, separate the elements in the content model using the choice operator “|”.

EXAMPLE For the EXPRESS declaration:

```
TYPE vehicle = SELECT (car, boat, plane);
END_TYPE;
TYPE plane = STRING; END_TYPE;
ENTITY car; END_ENTITY;
ENTITY boat; END_ENTITY;
```

the corresponding XML element declaration is

```
<!ELEMENT vehicle (car-ref | boat-ref |
plane)>
<!ATTLIST vehicle
  source_express_concept_type CDATA #FIXED
    "select-type"
  express_name CDATA #FIXED "vehicle">
```

4 For each EXPRESS entity data type declaration:

4.1 Create an ELEMENT declaration where the identifier of the element is the same as the ENTITY identifier.

4.2 Create an ATTLIST declaration for the ENTITY ELEMENT declaration with XML attributes called

4.2.1 “id” with an XML type of ID which is #REQUIRED.

4.2.2 "source_express_concept_type" with an XML type of CDATA and a #FIXED value of "entity data type".

4.2.3 "express_name" with an XML type of CDATA and a #FIXED value of the ENTITY identifier.

4.3 Create an entity data type reference ELEMENT declaration where the identifier is the entity data type reference identifier with a declared content model of EMPTY.

4.3.1 Create an ATTLIST declaration for the entity data type reference element with an XML attribute called “refid” with an XML type IDREF which is #REQUIRED.

4.3.2 Add attribute called "reftype" to the ATTLIST with an XML type of CDATA #FIXED and a value of "refid *xxx*" where xxx is the element type name of thing pointed to.

EXAMPLE For the EXPRESS Declaration:

```
ENTITY car; END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT car EMPTY>
<!ATTLIST car
    id ID #REQUIRED
    source_express_concept_type CDATA #FIXED
          "entity-data-type"
    express_name CDATA #FIXED "car">
<!ELEMENT car-ref EMPTY>
<!ATTLIST car-ref
    refid IDREF #REQUIRED
    reftype CDATA #FIXED "refid car">
```

4.4 If the ENTITY has no explicit attributes, insert the XML keyword EMPTY in place of the content model for the ENTITY ELEMENT declaration and go to 4.4.6; otherwise, for each EXPRESS explicit attribute of the ENTITY:

4.4.1 Create an ELEMENT declaration where the identifier is the ENTITY identifier and the EXPRESS attribute identifier concatenated with a “.” as a separator.

4.4.2 Add the attribute ELEMENT identifier to the content model of the ENTITY ELEMENT declaration. If not the first particle in the content model, separate from previous attribute element identifiers with a comma “,”.

4.4.3 Depending on the data type of the attribute:

4.4.3.1 If the attribute data type is a simple type, add the identifier of the simple type ELEMENT declaration to the content model of the attribute ELEMENT declaration.

EXAMPLE For the EXPRESS declaration:

```
ENTITY car;
  make : STRING;
  model : STRING;
END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT car (car.make, car.model)>
<!ATTLIST car
    id ID #REQUIRED
    source_express_concept_type CDATA #FIXED
      "entity-data-type"
    express_name CDATA #FIXED "car">
<!ELEMENT car.make (string)>
<!ATTLIST car.make
    source_express_concept_type CDATA #FIXED
      "attribute"
    express_name CDATA #FIXED "make">
<!ELEMENT car.model (string)>
<!ATTLIST car.model
    source_express_concept_type CDATA #FIXED
      "attribute"
    express_name CDATA #FIXED "model">
```

4.4.3.2 If the attribute data type is a defined type, enumeration type, or select type, add the identifier of the TYPE ELEMENT declaration to the content model of the attribute ELEMENT declaration.

EXAMPLE For the EXPRESS Declarations:

```
TYPE label = STRING; END_TYPE;
ENTITY car;
  make : label;
  model : label;
END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT label (string)>
<!ATTLIST label
    source_express_concept_type CDATA #FIXED
      "defined-type"
    express_name CDATA #FIXED "label">
<!ELEMENT car (car.make, car.model)>
<!ATTLIST car
    id ID #REQUIRED
    source_express_concept_type CDATA #FIXED
      "entity-data-type"
    express_name CDATA #FIXED "car">
<!ELEMENT car.make (label)>
<!ATTLIST car
    source_express_concept_type CDATA #FIXED
      "attribute"
    express_name CDATA #FIXED "make">
<!ELEMENT car.model (label)>
<!ATTLIST car
```

```
    source_express_concept_type CDATA #FIXED
      "attribute"
    express_name CDATA #FIXED "model">
```

4.4.3.3 If the attribute data type is an entity data type, add the entity data type reference identifier to the content model of the attribute ELEMENT declaration.

EXAMPLE For the EXPRESS declaration:

```
ENTITY person;
  owns : car;
END_ENTITY
ENTITY car;
  make : STRING;
  model : STRING;
END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT person (person.owns)>
<!ATTLIST person
    id ID #REQUIRED
    source_express_concept_type CDATA #FIXED
      "entity-data-type"
    express_name CDATA #FIXED "person">
<!ELEMENT person.owns (car-ref)>
<!ATTLIST person.owns
    source_express_concept_type CDATA #FIXED
      "attribute"
    express_name CDATA #FIXED "owns">
<!ELEMENT car (car.make, car.model)>
<!ATTLIST car
    id ID #REQUIRED
    source_express_concept_type CDATA #FIXED
      "entity-data-type"
    express_name CDATA #FIXED "car">
<!ELEMENT car-ref EMPTY>
<!ATTLIST car-ref
    refid IDREF #REQUIRED
    reftype CDATA #FIXED "refid car">
```

4.4.4 If the attribute data type is an aggregate:

4.4.4.1 Add the appropriate identifier of the aggregate base type (as determined in 4.4.3) to the content model of the attribute ELEMENT declaration.

4.4.4.2 Add the XML attribute aggregrate_type to the ATTLIST declaration for the attribute ELEMENT declaration with the #FIXED value of the aggregate type name.

4.4.4.3 If the aggregate is a SET, BAG, or LIST,

4.4.4.3.1 add the cardinality operator "+" (for lower bound of 1 or more) or "*" (for lower bound of 0) to the identifier

added to the content model of the attribute ELEMENT declaration in 4.4.4.1.

4.4.4.3.2 Add the XML attribute min_bound and max_bound to the ATTLIST declaration for the attribute ELEMENT declaration with the #FIXED values derived from the EXPRESS aggregate bound spec. If the bound specification is not a fixed value (e.g., it is computed by a derived attribute), then use #IMPLIED rather than #FIXED.

4.4.4.3.3 If the aggregate is a LIST, add the XML attribute "list_unique" to the ATTLIST with value choice "(true | false)" and default " 'false' ".

4.4.4.4 If the aggregate is an ARRAY:

4.4.4.4.1 add the cardinality operator "+" to the identifier added to the content model of the attribute ELEMENT declaration in 4.4.4.1.

4.4.4.4.2 Add the XML attributes low_index and high_index to the ATTLIST declaration for the attribute ELEMENT declaration with the #FIXED values derived from the EXPRESS array bound spec.

4.4.4.4.3 Add the XML attributes "array_optional" and "array_unique" to the ATTLIST with value choice "(true | false)" and default " 'false' ".

NOTE - Although the cardinality operator is "+", the EXPRESS language asserts that there shall be a fixed number of aggregate elements. Therefore, in an XML document instance, the aggregate element should explicitly contain high_index-low_index+1 number of sub-elements.

EXAMPLE For the EXPRESS declaration:

```
ENTITY person;
  owns : SET [1:?] OF car;
END_ENTITY
ENTITY car;END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT person (person.owns)>
<!ATTLIST person
id ID #REQUIRED
      source_express_concept_type CDATA #FIXED
        "entity-data-type"
      express_name CDATA #FIXED "person">
```

```
<!ELEMENT person.owns (car-ref+)>
<!ATTLIST person.owns
      source_express_concept_type CDATA #FIXED
            "attribute"
      express_name CDATA #FIXED "owns"
      aggregate_type CDATA #FIXED "set"
      min_bound CDATA #FIXED "1"
      max_bound CDATA #FIXED "?">

<!ELEMENT car EMPTY>
<!ATTLIST car
id ID #REQUIRED
      source_express_concept_type CDATA #FIXED
            "entity-data-type"
      express_name CDATA #FIXED "car">

<!ELEMENT car-ref EMPTY>
<!ATTLIST car-ref
refid IDREF #REQUIRED>
```

4.4.5 If the attribute data type is OPTIONAL, add the XML zero-or-one operator "?" to the attribute ELEMENT identifier in the content model of the ENTITY ELEMENT declaration.

EXAMPLE For the EXPRESS declaration:

```
ENTITY car;
  make : STRING;
  model : LIST [1:3] OF STRING;
  model_year : OPTIONAL INTEGER;
END_ENTITY;
```

the corresponding XML element declarations are:

```
<!ELEMENT car (car.make, car.model,
car.model_year?)>
<!ATTLIST car
id ID #REQUIRED
      source_express_concept_type CDATA #FIXED
            "entity-data-type"
      express_name CDATA #FIXED "car">

<!ELEMENT car-ref EMPTY>
<!ATTLIST car-ref
refid IDREF #REQUIRED
reftype CDATA #FIXED "refid car">

<!ELEMENT car.make (string)>
<!ATTLIST car.make
      source_express_concept_type CDATA #FIXED
            "attribute"
      express_name CDATA #FIXED "make">
<!ELEMENT car.model (string+)>
<!ATTLIST car.model
      source_express_concept_type CDATA #FIXED
            "attribute"
```

```
        express_name CDATA #FIXED "model"
        aggregate_type CDATA #FIXED "list"
        min_bound CDATA #FIXED "1"
        max_bound CDATA #FIXED "3"
        list_unique (true | false) "false">
<!ELEMENT car.model_year (integer)>
<!ATTLIST car.model_year
        source_express_concept_type CDATA #FIXED
              "attribute"
        express_name CDATA #FIXED "model_year">
```

4.4.6 For each EXPRESS UNIQUE rule:

4.4.6.1 For each component of the UNIQUE rule:

4.4.6.1.1 Add the XML attribute called "unique" to the ATTLIST of the attribute ELEMENT corresponding to the EXPRESS attribute to which the UNIQUE rule applies. The type of the XML attribute is CDATA #FIXED. The value of the XML attribute is constructed by appending the UNIQUE rule label (or the serial position of the UNIQUE if no label is present) with a "." and the numeric position of the identifier in the rule.

EXAMPLE For the EXPRESS declaration:

```
ENTITY car;
serial_no : INTEGER;
vin : STRING;
 UNIQUE
UR1: serial_no;
vin;
END_ENTITY;

ENTITY person;
owns : car;
name : STRING;
ssn : INTEGER;
 UNIQUE
UR1: name, ssn;
END_ENTITY;
```

the corresponding XML element declarations are:

(note: the "source_express_concept_type" and "express_name" attributes are omitted from these examples for brevity.)

```
<!ELEMENT car (car.serial_no, car.vin)>
<!ATTLIST car
id ID #IMPLIED>

<!ELEMENT car.serial_no (integer)>
```

```
<!ATTLIST car.serial_no
  unique CDATA #FIXED "UR1.1">
<!ELEMENT car.vin (string)>
<!ATTLIST car.vin
  unique CDATA #FIXED "2.1">

<!ELEMENT car-ref EMPTY>
<!ATTLIST car-ref
refid IDREF #REQUIRED
reftype CDATA #FIXED "refid car">

<!ELEMENT person (person.owns, person.name,
person.ssn)>
<!ATTLIST person
id ID #IMPLIED>

<!ELEMENT person.owns (car-ref)>
<!ELEMENT person.name (string)>
<!ATTLIST person.name
  unique CDATA #FIXED "UR1.1">
<!ELEMENT person.ssn (integer)>
<!ATTLIST person.ssn
  unique CDATA #FIXED "UR1.2">

<!ELEMENT person-ref EMPTY>
<!ATTLIST person-ref
refid IDREF #REQUIRED>
```

4.5 If the ENTITY is a supertype and the subtype/supertype graph contains no entities with multiple supertypes:

Note: for strictly hierarchical subtype/supertype graphs, the following portion of the algorithm takes advantage of the hierarchical nature of XML and produce a natural embedding of subtypes within supertypes.

4.5.1 Create an element with an identifier constructed by appending the string "-subtypes" to the identifier of the ENTITY data type.

4.5.2 Add the identifier created in 4.5.1 to the content model of the ENTITY ELEMENT declaration as the last particle of the content model.

4.5.3 If the ENTITY data type is not an ABSTRACT supertype, add a "?" to the content particle.

4.5.4 For the subgraph consisting of the ENTITY and its immediate subtypes, evaluate the set of complex entity data types according to annex B of ISO 10303-11.

4.5.4.1 Remove from this set the complex entity data corresponding to the ENTITY in isolation (if present);

4.5.4.2 Remove the supertype ENTITY partial complex entity data from each member of the remaining set. (Thus, the remaining partial complex instances in the set are the possible subtype combinations allowed for the supertype.)

4.5.5 If the set in 4.5.4 is empty, insert the XML keyword EMPTY in place of the content model of the ELEMENT created in 4.5.1. Otherwise, in the content model of the ELEMENT created in 4.5.1, add particles separated by the choice operator "|" where a particle is created for each member of the set in 4.5.4 as follows:

4.5.5.1 If the member is a single entity data type, the particle is the entity data type identifier.

4.5.5.2 If the member is comprised of two or more entity data types, the particle is a comma-separated, parenthesized list containing the identifiers of the entity data types.

4.5.6 Factor the content model of the *xxx*-subtypes element created in 4.5.5 according to the factoring algorithm presented in Annex xxx.

4.6 If the ENTITY is a supertype and the subtype/supertype graph contains entities with multiple supertypes, then process the *entire subtype/supertype graph* ***one time*** according to the following steps:

Note - This part of the algorithm for subtype/supertype graphs that contain multiple supertypes results in a single ELEMENT declaration that corresponding to the complex entity data type specified by the entire, *complete* subtype/supertype graph. This ELEMENT declaration is a "synthetic" type with respect to the EXPRESS schema because it does not correspond to any single EXPRESS entity declaration. Each individual entity in the graph is still declared as an ELEMENT, but the only place that the subtype instances of this element can appear is within the context of this large "synthetic" type. The supertypes can be instantiated on their own in the schema element.

4.6.1 Create an ELEMENT declaration with an identifier constructed from the names of the entities in the subtype/supertype graph. The name of the element is constructed by concatenating the names of all the independently instantiable entities in the subtype/supertype graph alphabetically using "CamelCase" for differentiation of name components.

4.6.2 Create an ATTLIST declaration for the ELEMENT.[1]

4.6.3 Evaluate the set of complex entity data types according to annex B of ISO 10303-11.

[1] What should go in this ATTLIST? What should the value of the "express_source_concept_type" attribute be?

4.6.4 Remove from this set the complex entity data types corresponding to supertypes in isolation (i.e., any singleton complex entity data types, any simple entity data types);

4.6.5 If the set in 4.6.4 is empty, insert the XML keyword EMPTY in place of the content model of the ELEMENT created in 4.6.1. Otherwise, in the content model of the ELEMENT created in 4.6.1, add particles separated by the choice operator "|" where a particle is created for each member of the set in 4.6.4 as follows:

4.6.5.1 If the member is a single entity data type, the particle is the entity data type identifier.

4.6.5.2 If the member is comprised of two or more entity data types, the particle is a comma-separated, parenthesized list containing the identifiers of the entity data types in alphabetical order.

EXAMPLE For the EXPRESS declaration:

```
ENTITY truck;
  axel_weight : REAL;
END_ENTITY
ENTITY boat;
  displacement : REAL;
END_ENTITY;
ENTITY amphibious_craft SUBTYPE OF (truck, boat);
  submersible : boolean;
END_ENTITY;
ENTITY commercial_craft SUBTYPE OF (amphibious_craft);
END_ENTITY;
```

```
The complex entity data types of this graph  are
truck&boat&amphibious_craft and
truck&boat&amphibious_craft&commercial_craft.
```

Add to the content model of the synthetic element declaration the identifiers of the ENTITY ELEMENT declarations corresponding to the entities involved in the complex entity data type. The ordering of the elements in the content model shall start with the supertypes in alphabetical order followed by the subtype(s).

4.6.6 The content model produced in 4.6.5 may not be syntactically legal. Therefore the content model should be factored according to the algorithm presented in Annex xxx.

The XML element declarations corresponding to the EXPRESS above are:

```
<!ELEMENT truck (truck.axle_weight)>
<!ATTLIST truck
id ID #REQUIRED
   source_express_concept_type CDATA #FIXED "entity-data-
   type"
```

```
   express_name CDATA #FIXED "truck">
<!ELEMENT truck.axle_weight (real)>
<!ATTLIST truck.axle_weight
   source_express_concept_type CDATA #FIXED "attribute"
   express_name CDATA #FIXED "axle_weight">
<!ELEMENT truck-ref EMPTY>
<!ATTLIST truck-ref
refid IDREF #REQUIRED
    reftype CDATA #FIXED "refid truck">

<!ELEMENT boat (boat.displacement)>
<!ATTLIST boat
id ID #REQUIRED
   source_express_concept_type CDATA #FIXED "entity-data-
   type"
   express_name CDATA #FIXED "boat">
<!ELEMENT boat.displacement (real)>
<!ATTLIST boat.displacement
   source_express_concept_type CDATA #FIXED "attribute"
   express_name CDATA #FIXED "displacement">
<!ELEMENT boat-ref EMPTY>
<!ATTLIST boat-ref
refid IDREF #REQUIRED
reftype CDATA #FIXED "refid boat">

<!ELEMENT amphibious_craft (amphibious_craft.submersible)>
<!ATTLIST amphibious_craft
id ID #REQUIRED
   source_express_concept_type CDATA #FIXED "entity-data-
   type"
   express_name CDATA #FIXED "amphibious_craft">
<!ELEMENT amphibious_craft.submersible (boolean)>
<!ATTLIST amphibious_craft.submersible
   source_express_concept_type CDATA #FIXED "attribute"
   express_name CDATA #FIXED "submersible">
<!ELEMENT amphibious_craft-ref EMPTY>
<!ATTLIST amphibious_craft-ref
refid IDREF #REQUIRED
reftype CDATA #FIXED "refid amphibious_craft">

<!ELEMENT commercial_craft EMPTY>
<!ATTLIST commercial_craft
id ID #REQUIRED
   source_express_concept_type CDATA #FIXED "entity-data-
   type"
   express_name CDATA #FIXED "commercial_craft">
<!ELEMENT commercial_craft-ref EMPTY>
<!ATTLIST commercial_craft-ref
refid IDREF #REQUIRED
reftype CDATA #FIXED "refid commercial_craft">

<!-- the factored synthetic type is -->

<!ELEMENT BoatTruckAmphibious_craft (boat, truck,
amphibious_craft, commercial_craft?)>
```

5 Optimization of DTD

5.1 Remove unused *xxx*-ref element type declarations;

5.2 Remove the ID for entity elements that are not referencable;

5.3 Remove unused primitives;

5.4 Remove any EMPTY *xxx*-subtype elements

5.5 Remove any EMPTY synthetic complex entity data type elements.

NOTE: The following two optimizations are provisionally included in this specification pending a decision whether they would be more appropriate for a future version of this binding.

5.6 Collapse entity attributes with same role name and datatype into a single element declaration that is used in multiple places.

5.7 Offer optional use of xxx-ref; instead of using it, include the referenced element directly in the referencing element's content.

Annex A
(normative)

Late Bound DTD elements for EXPRESS schema

Annex B
(normative)

Information object registration

To provide for unambiguous identification of an information object in an open system, the object identifier

{ iso standard 10303 part(28) version(-1) }

is assigned to this part of ISO 10303. The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

B.1.1 Information identification

To provide for unambiguous identification of the ??? DTD in an open information system, the object identifier

{ iso standard 10303 part(28) version(-1) object(1) ???(1) }

is assigned to the **?? DTD** (see clause ??). The meaning of this value is defined in ISO/IEC 8824-1, and is described in ISO 10303-1.

Assign other information identifiers as appropriate

Annex C
(informative)

Object serialization early binding

This annex specifies an early binding based on viewing data as a series of objects. This binding may be applied to sets of object instances that are specified by either an EXPRESS schema or by other object-based specification approaches.

The text that follows is as supplied to the editors of this part of ISO 10303. It is presented here in order to show the technical approach. The text itself is not yet in a form suitable for publication as a standard and requires considerable further editing.

C.1 Background and context

This paper describes a concept of operations for implementing product data sharing between applications using Web technologies. It includes some background and a description of the functional and non-functional requirements of such a facility. Finally, it concludes with a description of a binding from the EXPRESS information modeling language to XML. This binding supports the requirements described below.

The XML product information Xchange file is one element of an enterprise solution which is described in reference 1. A significant portion of the interoperability challenge entails the integration of components through well specified interfaces, which are designed to hide the details of the underlying implementation. In particular, these interfaces typically hide the specifics of the data managed by components. The Xchange file, however, is specifically charged with enabling the interchange of information between systems and business system domains. Consequently, it addresses interoperability at the implementation (rather than the interface) level. The Xchange file format provides the syntax as well as the minimum semantics required in order to serialize an object instance (or a graph of object instances emanating from a single root object) and to re-materialize the serialized data at a remote site/process. The Xchange file should be computer-interpretable by generally available XML parsing software.

The focus of the Xchange file is on the interchange of product information (and as such may be useful for information with similar characteristics). Typical product information can be characterized as a complex web of a large number of entities and relations, which must be managed over long periods of time. Nevertheless, it is often possible to divide product information into manageable-sized chunks for the purpose of individual interchanges. In fact, the Xchange file is geared toward just such aggregates of information, which are formulated as objects that correspond closely to the user's conceptual model of the product information. This representation may very well be a layer on top of low-level, normalized implementation entities defined using the EXPRESS information modeling language (eg an Application Interpreted Model (AIM)). On the other hand, this representation could be a layer on top of legacy data defined by some proprietary (possibly implicit) schema.

XML promises a wide range of services so it is important to be clear on the functional requirements that are expected to be provided by the Xchange file. The primary focus is the interoperability of systems (possibly across business systems domains). This implies a minimum of human viewing of the data. The human readability of the files through style sheets or other mechanisms is certainly a welcome feature, but it is a secondary requirement driving the design of the Xchange file. The typical exchange scenario is illustrated in Figure 1. A high level overview of the process follows in this section. Technical details of each step in the process are described in the following sections.

This scenario illustrates the interchange of product data from a system in one business system domain to a potentially different system in a second business system domain (possibly using a second middleware technology, component framework, or programming language). The details of the mechanism for moving the Xchange file and the messaging infrastructure in which the product information is contained are both omitted here for simplicity. The goal is to transfer product data from some product data repository so that it can be processed by an application in some other business system domain. The stored product data may be in the form of a STEP Application Integrated Model (AIM) that is modeled in EXPRESS, or it may be stored in a legacy data store in some proprietary schema. The application, which processes the product data, is the client to an interface which exposes an object-oriented representation of the product data. The object model on which the interface is based should be a simple object model that has been widely adopted by Web developers. The interface is implemented by classes which are object-based and which are based on an even simpler object model. Moreover, the classes should represent entities which are close to the user's conceptual model of the product.

In the Xchange scenario, these classes externalize (serialize) a product information instance (including the graph of objects which it references) to an XML file. The XML file is accompanied by a DTD which can be used to validate that the XML instance data conforms to the restrictions of the simple object model. The DTD does not, however, attempt to enforce the semantics of product's information model. The XML instance file may be accompanied by additional XML instance data which does represent the metadata of the product's information model (as defined in the simple object model).

The XML file is internalized (de-serialized) into classes in the target system. These classes support the same simple object model as the source classes and also implement the same product information model with respect to that object model. These classes may, however, be implemented using a different technology or programming language as the source classes. The "class-specific" metadata (which may be exchanged with instance data or established by prior agreement) is used to convert the XML instance back to a object-based programming language materialization of the product information instances. These classes on the target system also implement interfaces, which are fully object-oriented and which provide a data access interface which is substitutable for the source interface. The application in the target business system domain, then, is now able to process the interchanged product information.

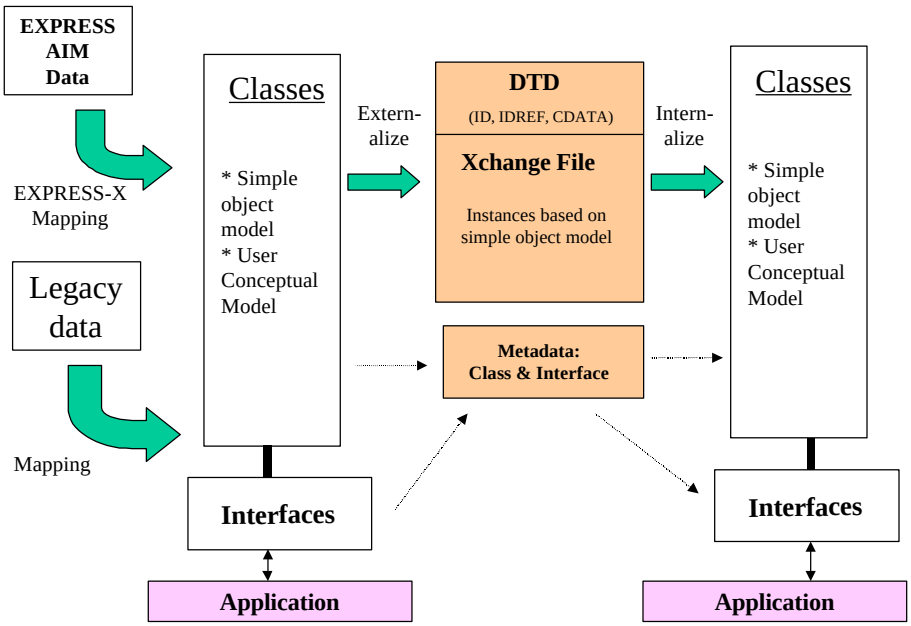

Figure C.1The Xchange scenario

Finally, there are non-functional requirements that need to be satisfied. Foremost is the re-usability requirement. The interchange of product information is a difficult and mutli-faceted process, which only begins with consensus on an information model. One of the reasons that STEP based product data interchange has languished is that users have not had the opportunity to re-use existing information technologies to build product information sharing systems. STEP implementations today are still being built from scratch or using obsolete implementation technologies. A goal of the Xchange process is to maximize the re-use of today's enabling technologies -- including available XML parsers and viewers, Web-based object technology such as Java, distributed component technologies, and the resources available on the Web itself.

C.2 Technical requirements

The functional and non-functional business requirements described above provide the foundation for a set of technical requirements. The XML in the Xchange file should provide the syntax for an object serialization (externalization) mechanism, which is compatible with the frameworks designed in accordance with the Serializer design pattern [Martin '98, p. 291]. The Serializer design pattern is specified and implemented in standard form in the Java Serialization service as well a number of implementations of the CORBA Externalization service. This requirement emphasizes the determination to maximize the re-use of mainstream technologies.

The idea of the Xchange process is that the product information can be represented by instances of classes of an object-oriented programming language in a source system, and the goal is to materialize

the product information in (possibly identical) classes in a target system. The product information is expected to cross system boundaries and may cross business system domain boundaries. Consequently, it is a requirement that the syntax of the Xchange file must be able to *interchange* completely the semantic capabilities supported by the classes of the source and target systems. Since one of the non-functional requirements is that the file be interpretable by generally available Web technologies, there is a technical requirement that the object model be simple and compatible with an object model that is considered a mainstream Web technology. The following section describes this requirement in more detail.

The Xchange file must be capable of containing persistent, global entity identifiers. Product data is characterized by its complexity and volume. Product information has become one of the common denominators of the industrial virtual enterprise. It is, thus, a requirement that any enabling product information management system be able to maintain a *distributed* representation of product information. More specifically, it must be possible to define a product in which different aspects of the product definition reside in different business system domains -- and these domains may exist anywhere in the world. It is not practical to exchange only self-sufficient, stand-alone segments of product information. Even though serialization technology requires that all local references be contained internally, it must be possible to represent an external identifier (in XML form) for references to entities outside the Xchange file.

C.3 Characteristics of the object model(s)

Figure 1 illustrates the point that the Xchange process is enabled not by one, but by four distinct, layered object models. The first is the object model that defines the stored product data: this may be the object model of EXPRESS or some proprietary data model. The second is the object model of the object interfaces that are presented to applications. The third is the object model that underlies the classes which implement the Xchange "wrapper". The last is the object model that underlies the contents of the Xchange file itself. These need not and should not be the same object model, but there are dependencies that must be maintained.

A goal of the Xchange process is to strive for completeness of the exchange. The set of semantics which is used to represent the stored product information drives this requirement. In some sense, this object model is open-ended, ranging from the semantics implied by the EXPRESS information modeling language to any conceivable proprietary or ad hoc schema used to define legacy data.

The object model of the interface must be rich enough to support this open-ended requirement; however, to satisfy the requirement of re-use of (Web) technology, that object model should be restricted in semantics of the interfaces provided by mainstream distributed object technologies. In practical terms this means the object model of Java (RMI) interfaces, or CORBA or DCOM IDL. Effective re-use precludes the specialized extension of these object models for product-related application development, including the production-capable information interchange itself. Therefore, the gap must be filled by a mapping strategy -- such as the use of EXPRESS-X to transform data defined in EXPRESS into a view that is amenable to the object model of the interface.

The next object model is that used to represent the classes which implement the product information "wrapper." Again, this object model need not be equivalent to the object model of the interface. The dependency is that this class object model be capable of *implementing* the interface object model. Today's mainstream object technologies support the clear separation of interface from implementation. Interfaces seek to hide the details of implementation from clients (including the sub-

interfaces that extend the original interface). One of the primary implementation details that is hidden by the interface is the representation of the data contained in the class. The XChange challenge, however, is obviously quite different. It cannot hide the implementation details implied by the data. The requirement is to effect a complete transfer of that data. In this respect the class is the suitable point of departure for the Xchange process.

The Jave language has demonstrated that implementation classes may be based on an object model that differs from the interface object model -- and they can still completely implement those interfaces. Specifically, the object model for the classes may forego multiple inheritance, polymorphism, and encapsulation -- and still implement interfaces that represent those characteristics. In fact, because of this the implementation classes do not have to be object-oriented; it is enough if they are object-based. The requirement is that the classes implement classification and instantiation. That is, every object must have an identity (and be accessible by means of an object reference) and every object must represent an instance of some type.

The requirement for re-use of (Web) technology means that for all practical purposes, the candidates for the Xchange object model are the Java and C++ object models. These two programming languages present fundamentally different object models. The C++ object model favors maximum flexibility and multiple ways to represent a given characteristic. The Java object model favors simplicity at the expense of some flexibility. [Syzperski '98, p.133] says: “Most successful languages [strive] for a balance between enforcement of ‘good things’ and flexibility to allow for the unforeseen. As projects grow in size and complexity, architecture gains importance. There is growing acceptance of the benefits of stringent languages, exemplified by the popular transition from C++ to Java." This is not to say that C++ cannot be used for Xchange process -- only that some of the rules inherent in the Java object model would have be adopted as design patterns when using C++.

The design goal for the Xchange class object model should be simplicity. That is, the number of special cases and alternative representations should be minimized. There should be no special cases based on the type of a specific class. Most important every thing must be representable as an object. Every object has identity and is accessed by means of an object reference. There is no equivalent of a C++ automatic class variable (or expanded class). A class is uniformly represented as a collection of attributes. The one concession to this rule is that there is small set of primitive types that will be represented as values (not objects) in cases where it can be done without a significant increase in complexity. An attribute, then, may be one of a) primitve, b) object reference or c) an array. (A more specific description of the mapping is described in the next section.)

C.4 The Xchange file

The good news is that the Xchange process need only concern itself with the simpler object model of the implementation classes. This means that the complexity of the interchange file can be reduced dramatically. In fact, the object model of the Xchange file can be simplified even further. The dependency is that it must represent an object model that is sufficient to *externalize/internalize* the implementation class. It should be noted that the design goal for the Xchange file is to minimize complexity -- not necessarily to minimize the size of the file. (By adopting XML, there are unavoidable concessions with respect to file size: no binary form, use of tags.) The goal is remove to all information not specifically needed to accomplish the externalization/internalization processes and to eliminate special cases and alternative representations.

One additional requirement on the Xchange file is to de-couple the source and the target classes. As was noted above, there should be no requirement that these classes are implemented in the same programming language. Moreover, the architecture should allow the maximum flexibility possible for the evolution of these classes. (This is the Xchange analog to binary compatibility among software components.) Consequently, the Xchange object model must be self-describing. Moreover, an object's attributes in the file should be order-independent. This sets the stage for one further simplification: the Xchange file need not specify the types of its attributes (within the instance data portion of the file.) The information needed to internalize an Xchange is provided by the name of the attribute. The target and the source class may use a different type for an attribute -- as long as the contract regarding the attribute's semantics is maintained; and this contract is specified outside of the Xchange file. This approach has been used successfully in the Newi Business Object Facility.[Eeles '98, p 152]

In deference to the requirement to the requirement to re-use XML software tools, there is one additional restriction on the nature of an object. An object in the Xchange file is comprised of a set of attribute names and values. An attribute, then, may be one of a) primitve, b) object reference or c) an array *of object references.* Consequently every type, including the primitive types, must be representable by an object type. This is analogous to the Java wrapper classes, java.lang.Integer and java.lang.Double. This requirement actually results in a simplification of some of the semantics required for the modeling of product information.

Finally, since the primary purpose of the Xchange file is to support externalization/internalization, it is not required to perform type checking on its contents. The assumption is that any desired type checking is performed by the target and/or source class. (As mentioned above, the types of attributes are not even captured in the file.) By the same token, the Xchange file is not intended to enforce any other constraints on the data. The checking of constraints should be performed by the target and/or source class and/or specified in the corresponding interface.

C.5 Role of the Xchange file DTD

The role of the Xchange file's DTD is to provide the metadata for the Xchange file's object model -- and nothing more. This DTD should not be used to attempt to describe the metadata of any of the other object models: neither the class object model, the interface object model, nor the product information object model itself. When parsing the Xchange file, it is necessary to determine for each value only whether it is an object's identifier, an object reference, a primitive value or an array of object references. This corresponds with capabilities natural to a DTD. The DTD specifies only whether a value is an ID, IDREF, CDATA or IDREFS. (Because the file does not specify the type of primitive valued attributes, there is only one primitive type required in the file: CDATA. The Xchange file DTD contains a subset of the information defined in the Xchange class metadata; and , thus, may be automatically generated from the class metadata.

Product information is also characterized by the possibility of unset attribute values. Often product information needs to be interchanged before it is fully defined. The product model schema typically designates which attributes are optional. This information could be represented in the DTD by specifying #REQUIRED or #IMPLIED attributes. However, this information is not necessarily accessible to the Xchange class itself, and, in fact, represents another constraint on the data that is more properly enforced elsewhere. Consequently, the guideline for the Xchange file is that all attributes shall be #IMPLIED. The machinery of the XML DTD are, thus, not used to enforce the constraints represented by optional information fields.

C.6 Instance data in the Xchange file

This sections describes how each feature of product information modeling semantic shall be represented in the instance data portion of the Xchange file. (The range of supported semantics corresponds to that supported by EXPRESS.)

One of the principal rules of the Xchange file is that every type may be representable as an object. This means that IDFEF(S) are untyped object references. Some types may be represented as primitive (CDATA) values -- but only in contexts that are natural for XML.

The following sections outline the (informal) guidelines for the specific modeling features supported by the Xchange file.

C.6.1 Objects

Each object is represented by an XML element with the name of the type of which it is a direct instance. (Need to agree on global naming for type names.)

Each object contains an attribute list that is the union of the attributes of all its ancestor types. The order of the attributes in the list shall not affect the internalization of the object.

Each object has an attribute "id" of type ID, which is its unique identifier within the file.

All primitive attributes shall be of type CDATA.

All object reference attributes shall be of type IDREF.

All collection attributes shall be of type IDREFS.

C.6.2 Primitives

Primitive types (float, double, int, long) shall be represented as attribute values of type CDATA.

Primitive types shall be represented by wrapper elements of the following form, where necessary:

```
<!ELEMENT double EMPTY>
<!ATTLIST double
  id  ID  #REQUIRED
  val CDATA  #REQUIRED>
```

C.6.3 Collections

Collection-valued attributes shall be represented by IDREFS. Each element in the collection must be an object (or object wrapper for a primitive or collection).

The object wrapper for collection shall be of the form:

```
<!ELEMENT collection EMPTY>
<!ATTLIST collection
 id    ID  #REQUIRED
 contents  IDREFS   #IMPLIED>
```

C.6.4 Boolean (and Logical)

Attributes of type bool or logical shall be represented by the appropriate enumerated values.

The wrapper elements for Boolean and Logical shall be:

```
<!ELEMENT bool EMPTY>
<!ATTLIST bool
 id    ID  #REQUIRED
 val   (true | false)#REQUIRED>

<!ELEMENT logical EMPTY>
<!ATTLIST logical
 id    ID    #REQUIRED
  val   (true | false | unknown)#REQUIRED>
```

C.6.5 Enumeration

Attributes of enumeration type shall be represented by the appropriate enumerated values.

The wrapper elements for an enumeration shall be of the form:

```
<!ELEMENT enum_name EMPTY>
<!ATTLIST enum_name
 id    ID  #REQUIRED
 val   (enumerator1 | enumerator2 | …)#REQUIRED>
```

C.6.6 Type definitions

- Elements of type definitions shall be designated by the underlying type of the type definition.

C.6.7 Binary

- Binary data shall be represented by the binary element:

```
<!ELEMENT binary (PCDATA)?>
<!ATTLIST binary
  id ID  #REQUIRED >
```

- [Binary mapping needs to be specified]

C.6.8 String

The wrapper class for string data shall be:

```
<!ELEMENT string  (#PCDATA)?>
<!ATTLIST string
  id ID  #REQUIRED
```

C.6.9 Discriminated union (EXPRESS SELECT)

Each discriminated union shall be of the form:

```
<!ELEMENT select_type EMPTY>
<!ATTLIST select_type
  id ID  #REQUIRED
  utype CDATA  #REQUIRED
  val IDREF  #IMPLIED>
```

The attribute utype shall contain the type name of the underlying type, which may be a type definition or an abstract class for which there is no corresponding element in the file.

C.6.10 Optionals (unset)

An unset entity (of any type) in a sparse collection shall be indicated by the null element, which shall be of the form:

```
<!ELEMENT null EMPTY>
<!ATTLIST null
id ID  #REQUIRED >
```

All other unset attributes shall be indicated by the absence of the attribute from the file.

C.6.11 Root element

The root element shall be used to enclose each single unit of externalization. Local pointers to objects within a root element are permitted, but there shall be no idrefs pointing to objects within a second root element. A root element may be used to represent an object that has a persistent, globally unique identifier.

The eckey element shall be used to represent such a persistent, globally unique identifier. Each root element is identified by its eckey element. Moreover, objects within a root element may refer to other root objects, which may be absent from the file, by using that root object's eckey. Within the eckey the optional home element shall designate the organization and repository which owns the root object; the

displayname element is an optional element, which may be used for an informative name such as may be associated with an icon on a desktop. The tech attribute shall designate the technology used to access the object. Finally, the root element shall contain the primary key of the root object, which is unique within the object's home. The primary key shall consist of one or more key attribute name-value pairs.

The root and eckey elements shall be of the form:

```
<!ELEMENT root(eckey, ANY*)>

<!ATTLIST root

  id      ID     #required

  eckeyid      IDREF    #required>

<!ELEMENT home #PCDATA>

<!ELEMENT displayname #PCDATA>

<!ELEMENT eckey(n*, home?, displayname?)>

<!ATTLIST eckey

  id      ID     #required

  tech      CDATA     #implied>

<!ELEMENT n(#PCDATA,v)>

<!ELEMENT v(#PCDATA)>
```

C.6.12 Out of scope

- The following are not contained in the Xchange file: derived attributes, functions , constraints, constant names.

C.6.13 Class and Interface Metadata

The Xchange file format may also be used to exchange metadata between source and target systems. That metadata may be the metadata for the source or target Xchange classes, the metadata for the source or target Xchange interfaces, or the metadata for the product information.

Annex D
(informative)

Computer interpretable listings

This annex references a listing of the DTD's specified in this part of ISO 10303. These listings are available in computer-interpretable form and can be found at the following URL:

http://www.mel.nist.gov/step/parts/part28e1/cd/

If there is difficulty accessing these sites contact ISO Central Secretariat or contact the ISO TC 184/SC4 Secretariat directly at: sc4sec@cme.nist.gov.

NOTE – The information provided in computer-interpretable form at the above URLs is informative. The information that is contained in the body of this part of ISO 10303 is normative.

Annex E
(informative)

Deterministic Content Models

XML requires that the content particles be deterministic based on the first item. This is an extract from the XML specification:

> “ For compatibility, it is required that content models in element type declarations be deterministic.
>
> SGML requires deterministic content models (it calls them "unambiguous"); XML processors built using SGML systems may flag non-deterministic content models as errors.
>
> > For example, the content model `((b, c) | (b, d))` is non-deterministic, because given an initial `b` the parser cannot know which `b` in the model is being matched without looking ahead to see which element follows the `b`. In this case, the two references to `b` can be collapsed into a single reference, making the model read `(b, (c | d))`. An initial `b` now clearly matches only a single name in the content model. The parser doesn't need to look ahead to see what follows; either `c` or `d` would be accepted.”

The ?? Early Binding combines elements corresponding to the elements in an EXPRESS subtype/supertype graph. The content models for such elements should be deterministic.

This annex presents one algorthm for defining such a deterministic content model corresponding to a given subtype/supertype graph.

Step 1 Evaluate the set S of allowed entity instances for the subtype/supertype graph according to the algorithm specified in ISO 10303-11, Annex B. This set may comprise both simple and complex entity instances.

Step 2 Identify the set E of entities that occur within S.

> Note: This will be all of or a subset of the entities declared within the subtype/supertype graph.

Step 3 Count the occurrences in S of each entity in E.

Step 4 Identify the entity that occurs most, M. In the event that two or more entities occur most often, select to be M the first entity according to the collating sequence of ISO 646).

Step 5 Divide the set S into two sub-sets: those entity instances that include M, refered to as with-M, and those that do not, without-M.

Step 6 Process the set with-M by removing M from each of the members.

Step 7 Compare with-M and without-M.

a) If they are the same and not empty, generate the content particle (M?, x) where x is the result of applying steps 2 to 6 of this algorithm to with-M.

b) If they are both empty: if this is the first pass through step 7 generate the content particle M, otherwise generate M?

c) If they are different and without-M is empty, generate the content particle (M, y) where y is the result of applying steps 2 to 6 of this algorithm to with-M.

d) If they are different and with-M is empty, generate the content particle (M | z) where z is the result of applying steps 2 to 6 of this algorithm to without-M.

e) If they are different and neither with-M nor without-M is empty, generate the content particle ((M, y) | (z)) where y is the result of applying steps 2 to 6 of this algorithm to with-M and z is the the result of applying steps 2 to 6 to without-M.

The result of the process will be a content particle which allows only the valid combinations in S:

((M, xxx) | (N, yyy) | (O, zzz) …)

where xxx, yyy and zzz are in turn complex or simple content particles and M, N and O are entity identifiers.

Annex F
(informative)

Late Bound DTD correspondence with EXPRESS syntax

Table D.1 shows the correspondence between the syntax of EXPRESS (as specified in ISO 10303-11) to elements used in the late bound DTD.

In table D.1 the EXPRESS syntax is reproduced from ISO10303-11 and the XML DTD element definitions are reproduced from clause 6 of this part of ISO 10303. In both cases table D.1 is not the primary definition of the respective syntax.

Table D.1 – Late Bound DTD correspondence with EXPRESS syntax

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ELEMENT express_driven_data (%remark;, (schema_decl* | data)*) > <!ATTLIST express_driven_data %ex-prod; "#NONE"> <!ELEMENT data (TBA)> <!ATTLIST data %ex-prod; "#NONE"> <!ELEMENT TBA EMPTY> <!ATTLIST TBA %ex-prod; "#NONE">
		<!ENTITY % ex-prod "express-production CDATA #FIXED">
		<!ENTITY % def_id "id ID #IMPLIED"> <!ENTITY % def_id_ref "refid IDREF #IMPLIED reftype CDATA ">
0	ABS = 'abs' !	<!ELEMENT abs EMPTY>
1	ABSTRACT = 'abstract' !	
2	ACOS = 'acos' !	<!ELEMENT acos EMPTY>
3	AGGREGATE = 'aggregate' !	

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
4	ALIAS = 'alias' !	
5	AND = 'and' !	<!ELEMENT and EMPTY>
6	ANDOR = 'andor' !	
7	ARRAY = 'array' !	
8	AS = 'as' !	
9	ASIN = 'asin' !	<!ELEMENT asin EMPTY>
10	ATAN = 'atan' !	<!ELEMENT atan EMPTY>
11	BAG = 'bag' !	
12	BEGIN = 'begin' !	
13	BINARY = 'binary' !	
14	BLENGTH = 'blength' !	<!ELEMENT blength EMPTY>
15	BOOLEAN = 'boolean' !	
16	BY = 'by' !	
17	CASE = 'case' !	
18	CONSTANT = 'constant' !	
19	CONST_E = 'const_e' !	<!ELEMENT const_e EMPTY>
20	CONTEXT = 'context' !	
21	COS = 'cos' !	<!ELEMENT cos EMPTY>
22	DERIVE = 'derive' !	
23	DIV = 'div' !	<!ELEMENT integer_divide EMPTY> <!ATTLIST integer_divide %ex-prod; "DIV">
24	ELSE = 'else' !	
25	END = 'end' !	
26	END_ALIAS = 'end_alias' !	

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
27	END_CASE = 'end_case' !	
28	END_CONSTANT = 'end_constant' !	
29	END_CONTEXT = 'end_context' !	
30	END_ENTITY = 'end_entity' !	
31	END_FUNCTION = 'end_function' !	
32	END_IF = 'end_if' !	
33	END_LOCAL = 'end_local' !	
34	END_MODEL = 'end_model' !	
35	END_PROCEDURE = 'end_procedure' !	
36	END_REPEAT = 'end_repeat' !	
37	END_RULE = 'end_rule' !	
38	END_SCHEMA = 'end_schema' !	
39	END_TYPE = 'end_type' !	
40	ENTITY = 'entity' !	
41	ENUMERATION = 'enumeration' !	
42	ESCAPE = 'escape' !	
43	EXISTS = 'exists' !	<!ELEMENT exists EMPTY>
44	EXP = 'exp' !	<!ELEMENT exp EMPTY>
45	FALSE = 'false' !	<!ELEMENT false EMPTY>
46	FIXED = 'fixed' !	<!ELEMENT fixed EMPTY>
47	FOR = 'for' !	
48	FORMAT = 'format' !	<!ELEMENT format EMPTY>
49	FROM = 'from' !	

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
50	FUNCTION = 'function' !	
51	GENERIC = 'generic' !	
52	HIBOUND = 'hibound' !	<!ELEMENT hibound EMPTY>
53	HIINDEX = 'hiindex' !	<!ELEMENT hiindex EMPTY>
54	IF = 'if' !	
55	IN = 'in' !	<!ELEMENT in EMPTY>
56	INSERT = 'insert' !	<!ELEMENT insert EMPTY>
57	INTEGER = 'integer' !	
58	INVERSE = 'inverse' !	
59	LENGTH = 'length' !	<!ELEMENT length EMPTY>
60	LIKE = 'like' !	<!ELEMENT like EMPTY>
61	LIST = 'list' !	
62	LOBOUND = 'lobound' !	<!ELEMENT lobound EMPTY>
63	LOCAL = 'local' !	
64	LOG = 'log' !	<!ELEMENT log EMPTY>
65	LOG10 = 'log10' !	<!ELEMENT log10 EMPTY>
66	LOG2 = 'log2' !	<!ELEMENT log2 EMPTY>
67	LOGICAL = 'logical' !	
68	LOINDEX = 'loindex' !	<!ELEMENT loindex EMPTY>
69	MOD = 'mod' !	<!ELEMENT mod EMPTY>
70	MODEL = 'model' !	
71	NOT = 'not' !	<!ELEMENT not EMPTY>
72	NUMBER = 'number' !	

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
73	NVL = 'nvl' !	<!ELEMENT nvl EMPTY>
74	ODD = 'odd' !	<!ELEMENT odd EMPTY>
75	OF = 'of' !	
76	ONEOF = 'oneof' !	
77	OPTIONAL = 'optional' !	<!ELEMENT optional EMPTY>
78	OR = 'or' !	<!ELEMENT or EMPTY>
79	OTHERWISE = 'otherwise' !	<!ELEMENT otherwise (%remark;, (%stmt;)) >
80	PI = 'pi' !	<!ELEMENT pi EMPTY>
81	PROCEDURE = 'procedure' !	
82	QUERY = 'query' !	
83	REAL = 'real' !	
84	REFERENCE = 'reference' !	
85	REMOVE = 'remove' !	<!ELEMENT remove EMPTY>
86	REPEAT = 'repeat' !	
87	RETURN = 'return' !	
88	ROLESOF = 'rolesof' !	<!ELEMENT rolesof EMPTY>
89	RULE = 'rule' !	
90	SCHEMA = 'schema' !	
91	SELECT = 'select' !	
92	SELF = 'self' !	<!ELEMENT self EMPTY>
93	SET = 'set' !	
94	SIN = 'sin' !	<!ELEMENT sin EMPTY>
95	SIZEOF = 'sizeof' !	<!ELEMENT sizeof EMPTY>

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
96	SKIP = 'skip' !	
97	SQRT = 'sqrt' !	<!ELEMENT sqrt EMPTY>
98	STRING = 'string' !	
99	SUBTYPE = 'subtype' !	
100	SUPERTYPE = 'supertype' !	
101	TAN = 'tan' !	<!ELEMENT tan EMPTY>
102	THEN = 'then' !	
103	TO = 'to' !	
104	TRUE = 'true' !	<!ELEMENT true EMPTY>
105	TYPE = 'type' !	
106	TYPEOF = 'typeof' !	<!ELEMENT typeof EMPTY>
107	UNIQUE = 'unique' !	<!ELEMENT unique EMPTY>
108	UNKNOWN = 'unknown' !	<!ELEMENT unknown EMPTY>
109	UNTIL = 'until' !	
110	USE = 'use' !	
111	USEDIN = 'usedin' !	<!ELEMENT usedin EMPTY>
112	VALUE = 'value' !	<!ELEMENT value EMPTY>
113	VALUE_IN = 'value_in' !	<!ELEMENT value_in EMPTY>
114	VALUE_UNIQUE = 'value_unique' !	<!ELEMENT value_unique EMPTY>
115	VAR = 'var' !	
116	WHERE = 'where' !	
117	WHILE = 'while' !	
118	XOR = 'xor' !	<!ELEMENT xor EMPTY>

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
119	bit = '0' | '1' .	
120	digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .	
121	digits = digit { digit } .	
122	encoded_character = octet octet octet octet .	
123	hex_digit = digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .	
124	letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' .	
125	lparen_not_star = '(' not_star .	
126	not_lparen_star = not_paren_star | ')' .	
127	not_paren_star = letter | digit | not_paren_star_special .	
128	not_paren_star_quote_special = '!' | '"' | '#' | '$' | '%' | '&' | '+' | ',' | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | '[' | '\' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~' .	
129	not_paren_star_special = not_paren_star_quote_special | '''' .	
130	not_quote = not_paren_star_quote_special | letter | digit | '(' | ')' | '*' .	
131	not_rparen = not_paren_star | '*' | '(' .	
132	not_star = not_paren_star | '(' | ')' .	
133	octet = hex_digit hex_digit .	
134	special = not_paren_star_quote_special	

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
	| '(' | ')' | '*' | '''' !	
135	star_not_rparen = '*' not_rparen .	
136	binary_literal = '%' bit { bit } !	<!ELEMENT binary_literal (#PCDATA) >
137	encoded_string_literal = '"' encoded_character { encoded_character } '"' .	
138	integer_literal = digits !	<!ELEMENT integer_literal (#PCDATA) >
139	real_literal = digits '.' [digits] ['e' [sign] digits] !	<!ELEMENT real_literal (#PCDATA) >
140	simple_id = letter { letter | digit | '_' } !	
141	simple_string_literal = \q { (\q \q) | not_quote | \s | \o } \q .	
142	embedded_remark = '(*' { not_lparen_star | lparen_not_star | star_not_rparen | embedded_remark } '*)' .	<!ELEMENT embedded_remark (#PCDATA | embedded_remark)* >
143	remark = embedded_remark | tail_remark .	<!ENTITY % remark “(embedded_remark | tail_remark)?”>
144	tail_remark = '--' { \a | \s | \o } \n .	<!ELEMENT tail_remark (#PCDATA)* >
145	attribute_ref = attribute_id !	<!ELEMENT attribute_ref (#PCDATA) > <!ATTLIST attribute_ref %def_id_ref; “attribute_id”>
146	constant_ref = constant_id !	<!ELEMENT constant_ref (#PCDATA) > <!ATTLIST constant_ref %def_id_ref; “constant_id”>
147	entity_ref = entity_id !	<!ELEMENT entity_ref (#PCDATA) > <!ATTLIST entity_ref %def_id_ref; “entity_id”>
148	enumeration_ref = enumeration_id !	<!ELEMENT enumeration_ref (#PCDATA) > <!ATTLIST enumeration_ref %def_id_ref; “enumeration_id”>

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
149	function_ref = function_id !	<!ELEMENT function_ref (#PCDATA) > <!ATTLIST function_ref %def_id_ref; “function_id”>
150	parameter_ref = parameter_id !	<!ELEMENT parameter_ref (#PCDATA) > <!ATTLIST parameter_ref %def_id_ref; “parameter_id”>
151	procedure_ref = procedure_id !	<!ELEMENT procedure_ref (#PCDATA) > <!ATTLIST procedure_ref %def_id_ref; “procedure_id”>
152	schema_ref = schema_id !	<!ELEMENT schema_ref (#PCDATA) > <!ATTLIST schema_ref %def_id_ref; “schema_id”>
153	type_label_ref = type_label_id !	<!ELEMENT type_label_ref (#PCDATA) > <!ATTLIST type_label_ref %def_id_ref; “type_label_id”>
154	type_ref = type_id !	<!ELEMENT type_ref (#PCDATA) > <!ATTLIST type_ref %def_id_ref; “type_id”>
155	variable_ref = variable_id !	<!ELEMENT variable_ref (#PCDATA) > <!ATTLIST variable_ref %def_id_ref; “variable_id”>
156	abstract_supertype_declaration = ABSTRACT SUPERTYPE [subtype_constraint] .	<!ELEMENT abstract_supertype (%remark;, (%supertype_expression;)?) > <!ATTLIST abstract_supertype %ex-prod; "abstract_supertype_declaration">
157	actual_parameter_list = '(' parameter { ',' parameter } ')' .	<!ENTITY % actual_parameter_list “(%parameter;)*” >
158	add_like_op = '+' | '-' | OR | XOR .	<!ENTITY % add_like_op “add | subtract | or | xor “ >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ELEMENT add EMPTY> <!ATTLIST add %ex-prod; "+"> <!ELEMENT subtract EMPTY> <!ATTLIST subtract %ex-prod; "-">
159	aggregate_initializer = '[' [element { ',' element }] ']' !	<!ELEMENT aggregate_initializer (%remark;, element_list) >
160	aggregate_source = simple_expression .	<!ELEMENT aggregate_source (%remark;, (%SIMPLE_EXPRESSION;)) >
161	aggregate_type = AGGREGATE [':' type_label] OF parameter_type !	<!ELEMENT aggregate_type (%remark;, (%parameter_type;), (%type_label;)?) >
162	aggregation_types = array_type | bag_type | list_type | set_type !	<!ENTITY % aggregation_types “ array_type | bag_type | list_type | set_type“>
163	algorithm_head = { declaration } [constant_decl] [local_decl] .	<!ELEMENT algorithm_head (%remark;, declaration_block?, constant_block?, local_variable_block?) >
164	alias_stmt = ALIAS variable_id FOR general_ref { qualifier } ';' stmt { stmt } END_ALIAS ';' !	<!ELEMENT alias_stmt (%remark;, variable_id, (%general_ref;), qualifier?, statement_block) >
165	array_type = ARRAY bound_spec OF [OPTIONAL] [UNIQUE] base_type !	<!ELEMENT array_type (%remark;, index_spec, base_type, optional?, unique?) >
166	assignment_stmt = general_ref { qualifier } ':=' expression ';' !	<!ELEMENT assignment_stmt (%remark;, (%general_ref;), qualifier?, (%expression;)) >
167	attribute_decl = attribute_id | qualified_attribute .	<!ENTITY % attribute_decl “attribute_id | qualified_attribute” >
168	attribute_id = simple_id !	<!ELEMENT attribute_id (#PCDATA) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ATTLIST attribute_id %def_id;>
169	attribute_qualifier = '.' attribute_ref .	<!ENTITY % attribute_qualifier “attribute_ref” >
170	bag_type = BAG [bound_spec] OF base_type !	<!ELEMENT bag_type (%remark;, bound_spec?, base_type) >
171	base_type = aggregation_types | simple_types | named_types .	<!ELEMENT base_type (%remark;, (%aggregation_types; | %simple_types; | %named_types;)) >
172	binary_type = BINARY [width_spec] !	<!ELEMENT binary (%remark;, width_spec?) > <!ATTLIST binary %ex-prod; "binary_type">
173	boolean_type = BOOLEAN !	<!ELEMENT boolean EMPTY > <!ATTLIST boolean %ex-prod; " boolean_type">
174	bound_1 = numeric_expression .	<!ELEMENT lower_bound (%remark;, (%numeric_expression_top;)) > <!ATTLIST lower_bound %ex-prod; " bound_1">
175	bound_2 = numeric_expression .	<!ELEMENT upper_bound (%remark;, (indeterminate | %numeric_expression_top;)) > <!ATTLIST upper_bound %ex-prod; "bound_2"> <!ELEMENT indeterminate EMPTY > <!ATTLIST indeterminate %ex-prod; "?">
176	bound_spec = '[' bound_1 ':' bound_2 ']' .	<!ELEMENT bound_spec (%remark;, lower_bound, upper_bound) > <!ELEMENT index_spec (%remark;, low_index, high_index?) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ATTLIST index_spec %ex-prod; "bound_spec">
177	built_in_constant = CONST_E | PI | SELF | '?' !	<!ENTITY % built_in_constant “const_e | pi | self | unknown” >
178	built_in_function = ABS | ACOS | ASIN | ATAN | BLENGTH | COS | EXISTS | EXP | FORMAT | HIBOUND | HIINDEX | LENGTH | LOBOUND | LOINDEX | LOG | LOG2 | LOG10 | NVL | ODD | ROLESOF | SIN | SIZEOF | SQRT | TAN | TYPEOF | USEDIN | VALUE | VALUE_IN | VALUE_UNIQUE !	<!ENTITY % built_in_function "abs | acos | asin | atan | blength | cos | exists | exp | format | hibound | hiindex | length | lobound | loindex | log | log2 | log10 | nvl | odd | rolesof | sin | sizeof | sqrt | tan | typeof | usedin | value | value_in | value_unique" >
179	built_in_procedure = INSERT | REMOVE !	<!ENTITY % built_in_procedure “insert | remove” >
180	case_action = case_label { ',' case_label } ':' stmt .	<!ELEMENT case_action (%remark;, case_label, (%stmt;)) >
181	case_label = expression .	<!ELEMENT case_label (%remark;, (%expression;)+) >
182	case_stmt = CASE selector OF { case_action } [OTHERWISE ':' stmt] END_CASE ';' !	<!ELEMENT case_stmt (%remark;, (%selector;), case_action*, otherwise?) >
183	compound_stmt = BEGIN stmt { stmt } END ';' !	<!ENTITY % compound_stmt "statement_block" >
184	constant_body = constant_id ':' base_type ':=' expression ';' .	<!ELEMENT constant_decl (%remark;, constant_id, base_type, (%expression;)) > <!ATTLIST constant_decl %ex-prod; "constant_body">
185	constant_decl = CONSTANT constant_body { constant_body } END_CONSTANT ';' !	<!ELEMENT constant_block (%remark;, constant_decl*) > <!ATTLIST constant_block %ex-prod; "constant_decl">

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
186	constant_factor = built_in_constant | constant_ref .	<!ENTITY % constant_factor "%built_in_constant; | constant_ref" >
187	constant_id = simple_id !	<!ELEMENT constant_id (#PCDATA) > <!ATTLIST constant_id %def_id;>
188	constructed_types = enumeration_type | select_type !	<!ENTITY % constructed_types "enumeration | select ">
189	declaration = entity_decl | function_decl | procedure_decl | type_decl .	<!ENTITY % declaration "entity_decl | function_decl | procedure_decl | type_decl" > <!ELEMENT declaration_block (%remark;, (%declaration;)*) > <!ATTLIST declaration_block %ex-prod; "declaration">
190	derived_attr = attribute_decl ':' base_type ':=' expression ';' .	<!ELEMENT derived_attr (%remark;, (%attribute_decl;), base_type, (%expression;)) >
191	derive_clause = DERIVE derived_attr { derived_attr } !	<!ELEMENT derive_clause (%remark;, derived_attr+) >
192	domain_rule = [label ':'] logical_expression !	<!ELEMENT domain_rule (%remark;, label?, logical_expression) >
193	element = expression [':' repetition] .	<!ELEMENT element_list (%remark;, element_item*) > <!ATTLIST element_list %ex-prod; "element"> <!ELEMENT element_item (%remark;, (%expression;), repetition?) > <!ATTLIST element_item %ex-prod; "element">
194	entity_body = { explicit_attr } [derive_clause] [inverse_clause] [unique_clause] [	<!ENTITY % entity_body "explicit_attr_block?, derive_clause?, inverse_clause?, unique_clause?,

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
	where_clause] .	where_clause?” >
195	entity_constructor = entity_ref '(' [expression { ',' expression }] ')' !	<!ELEMENT entity_constructor (%remark;, entity_ref, (%expression;)*) >
196	entity_decl = entity_head entity_body END_ENTITY ';' !	<!ELEMENT entity_decl (%remark;, %entity_head;, %entity_body;) >
197	entity_head = ENTITY entity_id [subsuper] ';' .	<!ENTITY % entity_head “entity_id, %subsuper;” >
198	entity_id = simple_id !	<!ELEMENT entity_id (#PCDATA) > <!ATTLIST entity_id %def_id;>
199	enumeration_id = simple_id !	<!ELEMENT enumeration_id (#PCDATA) > <!ATTLIST enumeration_id %def_id;>
200	enumeration_reference = [type_ref '.'] enumeration_ref !	<!ELEMENT enumeration_reference (%remark;, type_ref?, enumeration_ref) >
201	enumeration_type = ENUMERATION OF '(' enumeration_id { ',' enumeration_id } ')' !	<!ELEMENT enumeration (%remark;, enumeration_id+) > <!ATTLIST enumeration %ex-prod; "enumeration_type">
202	escape_stmt = ESCAPE ';' !	<!ELEMENT escape_stmt EMPTY >
203	explicit_attr = attribute_decl { ',' attribute_decl } ':' [OPTIONAL] base_type ';' !	<!ELEMENT explicit_attr_block (%remark;, explicit_attr+) > <!ATTLIST explicit_attr_block %ex-prod; "explicit_attr"> <!ELEMENT explicit_attr (%remark;, (%attribute_decl;), optional?, base_type) >
204	expression = simple_expression [rel_op_extended	<!ENTITY % expression "%SIMPLE_EXPRESSION; | relation_expression"

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
	simple_expression] !	>\<!ELEMENT relation_expression (%remark;, (%rel_op_extended;), (%SIMPLE_EXPRESSION;), (%SIMPLE_EXPRESSION;)) >\<!ATTLIST relation_expression %ex-prod; "expression">
205	factor = simple_factor ['**' simple_factor] .	<!ENTITY % FACTOR “%simple_factor; | factor” > <!ELEMENT factor (raise_to_power, (%simple_factor;), (%simple_factor;)) > <!ELEMENT raise_to_power EMPTY> <!ATTLIST raise_to_power %ex-prod; "**">
206	formal_parameter = parameter_id { ',' parameter_id } ':' parameter_type .	<!ELEMENT formal_parameter_block (%remark;, formal_parameter*) > <!ATTLIST formal_parameter_block %ex-prod; "formal_parameter">> <!ELEMENT formal_parameter (%remark;, parameter_id, (%parameter_type;)) > <!ELEMENT procedure_formal_parameter_block (%remark;, (formal_parameter | var_formal_parameter)*) > <!ATTLIST procedure_formal_parameter_block %ex-prod; "formal_parameter">> <!ELEMENT var_formal_parameter (%remark;, parameter_id, (%parameter_type;)) > <!ATTLIST var_formal_parameter %ex-prod; "formal_parameter">>
207	function_call = (built_in_function | function_ref) [actual_parameter_list] !	<!ELEMENT function_call (%remark;, (%built_in_function; | function_ref), %actual_parameter_list;) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
208	function_decl = function_head [algorithm_head] stmt { stmt } END_FUNCTION ';' !	<!ELEMENT function_decl (%remark;, %function_head;, algorithm_head?, statement_block) >
209	function_head = FUNCTION function_id ['(' formal_parameter { ';' formal_parameter } ')'] ':' parameter_type ';' .	<!ENTITY % function_head "function_id, formal_parameter_block?, function_return_type" >
210	function_id = simple_id !	<!ELEMENT function_id (#PCDATA) > <!ATTLIST function_id %def_id;>
211	generalized_types = aggregate_type | general_aggregation_types | generic_type !	<!ENTITY % generalized_types "aggregate_type | %general_aggregation_types; | generic_type" >
212	general_aggregation_types = general_array_type | general_bag_type | general_list_type | general_set_type !	<!ENTITY % general_aggregation_types "general_array_type | general_bag_type | general_list_type | general_set_type" >
213	general_array_type = ARRAY [bound_spec] OF [OPTIONAL] [UNIQUE] parameter_type .	<!ELEMENT general_array_type (%remark;, (%parameter_type;), bound_spec?, optional?, unique?) >
214	general_bag_type = BAG [bound_spec] OF parameter_type .	<!ELEMENT general_bag_type (%remark;, (%parameter_type;), bound_spec?) >
215	general_list_type = LIST [bound_spec] OF [UNIQUE] parameter_type .	<!ELEMENT general_list_type (%remark;, (%parameter_type;), bound_spec?, unique?) >
216	general_ref = parameter_ref | variable_ref .	<!ENTITY % general_ref "parameter_ref | variable_ref" >
217	general_set_type = SET [bound_spec] OF parameter_type .	<!ELEMENT general_set_type (%remark;, (%parameter_type;), bound_spec?) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
218	generic_type = GENERIC [':' type_label] !	<!ELEMENT generic_type (%remark;, (%type_label;)?) >
219	group_qualifier = '\' entity_ref .	<!ENTITY % group_qualifier “entity_ref” >
220	if_stmt = IF logical_expression THEN stmt { stmt } [ELSE stmt { stmt }] END_IF ';' !	<!ELEMENT if_stmt (%remark;, logical_expression, statement_block, statement_block?) >
221	increment = numeric_expression .	<!ELEMENT increment (%remark;, (%numeric_expression_top;)) >
222	increment_control = variable_id ':=' bound_1 TO bound_2 [BY increment] .	<!ELEMENT increment_control (%remark;, variable_id, lower_bound, upper_bound, increment?) >
223	index = numeric_expression .	<!ENTITY % index “%numeric_expression_top;” >
224	index_1 = index .	<!ELEMENT low_index (%remark;, (%index;)) > <!ATTLIST low_index %ex-prod; "index_1">
225	index_2 = index .	<!ELEMENT high_index (%remark;, (%index;)) > <!ATTLIST high_index %ex-prod; "index_2">
226	index_qualifier = '[' index_1 [':' index_2] ']' .	<!ELEMENT index_qualifier (%remark;, low_index, high_index?) >
227	integer_type = INTEGER !	<!ELEMENT integer EMPTY > <!ATTLIST integer %ex-prod; "integer_type">
228	interface_specification = reference_clause | use_clause .	<!ELEMENT interface_specification_block (%remark;, (reference_from | use_from)+) > <!ATTLIST

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		interface_specification_block %ex-prod; "interface_specification">
229	interval = '{' interval_low interval_op interval_item interval_op interval_high '}' !	<!ELEMENT interval (%remark;, (interval_low_inclusive | interval_low_exclusive), interval_item, (interval_high_inclusive | interval_high_exclusive)) >
230	interval_high = simple_expression .	<!ELEMENT interval_high_inclusive (%remark;, (%SIMPLE_EXPRESSION;)) > <!ATTLIST interval_high_inclusive %ex-prod; "interval_high">> <!ELEMENT interval_high_exclusive (%remark;, (%SIMPLE_EXPRESSION;)) > <!ATTLIST interval_high_exclusive %ex-prod; "interval_high">>
231	interval_item = simple_expression .	<!ELEMENT interval_item (%remark;, (%SIMPLE_EXPRESSION;)) >
232	interval_low = simple_expression .	<!ELEMENT interval_low_inclusive (%remark;, (%SIMPLE_EXPRESSION;)) > <!ATTLIST interval_low_inclusive %ex-prod; "interval_low">> <!ELEMENT interval_low_exclusive (%remark;, (%SIMPLE_EXPRESSION;)) > <!ATTLIST interval_low_exclusive %ex-prod; "interval_low">>
233	interval_op = '<' | '<=' .	
234	inverse_attr = attribute_decl ':' [(SET | BAG) [bound_spec] OF] entity_ref FOR attribute_ref ';' .	<!ELEMENT inverse_attr (%remark;, (%attribute_decl;), entity_ref, attribute_ref, (inverse_set | inverse_bag)?) > <!ELEMENT inverse_set (%remark;, bound_spec?) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ATTLIST inverse_set %ex-prod; "SET"> <!ELEMENT inverse_bag (%remark;, bound_spec?) > <!ATTLIST inverse_bag %ex-prod; "BAG">
235	inverse_clause = INVERSE inverse_attr { inverse_attr } !	<!ELEMENT inverse_clause (%remark;, inverse_attr+) >
236	label = simple_id !	<!ELEMENT label (#PCDATA) >
237	list_type = LIST [bound_spec] OF [UNIQUE] base_type !	<!ELEMENT list_type (%remark;, bound_spec?, base_type, unique?) >
238	literal = binary_literal | integer_literal | logical_literal | real_literal | string_literal !	<!ENTITY % literal “binary_literal | integer_literal | logical_literal | real_literal | string_literal” >
239	local_decl = LOCAL local_variable { local_variable } END_LOCAL ';' !	<!ELEMENT local_variable_block (%remark;, local_variable_decl*) > <!ATTLIST local_variable_block %ex-prod; "local_decl">
240	local_variable = variable_id { ',' variable_id } ':' parameter_type [':=' expression] ';' .	<!ELEMENT local_variable_decl (%remark;, variable_id, (%parameter_type;), (%expression;)?) > <!ATTLIST local_variable_decl %ex-prod; "local_variable">
241	logical_expression = expression .	<!ELEMENT logical_expression (%remark;, (%expression;)) >
242	logical_literal = FALSE | TRUE | UNKNOWN !	<!ELEMENT logical_literal (%remark;, (false | true | unknown)) >
243	logical_type = LOGICAL !	<!ELEMENT logical EMPTY > <!ATTLIST logical %ex-prod; "logical_type">

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
244	multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .	<!ENTITY % multiplication_like_op “multiply | real_divide | integer_divide | mod | and | complex_entity_constructor“ > <!ELEMENT multiply EMPTY> <!ATTLIST multiply %ex-prod; "*">> <!ELEMENT real_divide EMPTY> <!ATTLIST real_divide %ex-prod; "/">> <!ELEMENT complex_entity_constructor EMPTY> <!ATTLIST complex_entity_constructor %ex-prod; "||">
245	named_types = entity_ref | type_ref !	<!ENTITY % named_types “entity_ref | type_ref “ >
246	named_type_or_rename = named_types [AS (entity_id | type_id)] .	
247	null_stmt = ';' !	<!ELEMENT null_stmt EMPTY >
248	number_type = NUMBER !	<!ELEMENT number EMPTY >
249	numeric_expression = simple_expression !	<!ENTITY % numeric_expression_top “integer_literal | numeric_expression” > <!ELEMENT numeric_expression (%remark;, (%SIMPLE_EXPRESSION;)) >
250	one_of = ONEOF '(' supertype_expression { ',' supertype_expression } ')' .	<!ELEMENT supertype_one_of (%remark;, (%supertype_expression;)+) > <!ATTLIST supertype_one_of %ex-prod; "one_of">>
251	parameter = expression .	<!ENTITY % parameter "%expression;" >
252	parameter_id = simple_id !	<!ELEMENT parameter_id (#PCDATA)

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		>\<!ATTLIST parameter_id %def_id;>
253	parameter_type = generalized_types | named_types | simple_types .	<!ENTITY % parameter_type “%generalized_types; | %named_types; | %simple_types;” > <!ELEMENT function_return_type (%parameter_type;) > <!ATTLIST function_return_type %ex-prod; "parameter_type">
254	population = entity_ref .	<!ELEMENT population (%remark;, entity_ref) >
255	precision_spec = numeric_expression .	<!ELEMENT precision_spec (%remark;, (%numeric_expression_top;)) >
256	primary = literal | (qualifiable_factor { qualifier }) .	<!ENTITY % primary “%literal; | %qualifiable_factor; | qualified_factor” > <!ELEMENT qualified_factor (%remark;, (%qualifiable_factor;), qualifier) > <!ATTLIST qualified_factor %ex-prod; "qualifiable_factor">
257	procedure_call_stmt = (built_in_procedure | procedure_ref) [actual_parameter_list] ';' !	<!ELEMENT procedure_call_stmt (%remark;, ((%built_in_procedure; | procedure_ref), %actual_parameter_list;)) >
258	procedure_decl = procedure_head [algorithm_head] { stmt } END_PROCEDURE ';' !	<!ELEMENT procedure_decl (%remark;, %procedure_head;, algorithm_head?, statement_block?) >
259	procedure_head = PROCEDURE procedure_id ['(' [VAR] formal_parameter { ';' [VAR] formal_parameter } ')'] ';' .	<!ENTITY % procedure_head “procedure_id, procedure_formal_parameter_block?” >
260	procedure_id = simple_id !	<!ELEMENT procedure_id (#PCDATA) >

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		<!ATTLIST procedure_id %def_id;>
261	qualifiable_factor = attribute_ref | constant_factor | function_call | general_ref | population !	<!ENTITY % qualifiable_factor “attribute_ref | %constant_factor; | function_call | %general_ref; | population” >
262	qualified_attribute = SELF group_qualifier attribute_qualifier !	<!ELEMENT qualified_attribute (%remark;, %group_qualifier;, %attribute_qualifier;) >
263	qualifier = attribute_qualifier | group_qualifier | index_qualifier !	<!ELEMENT qualifier (%remark;, (%attribute_qualifier; | %group_qualifier; | index_qualifier)*) >
264	query_expression = QUERY '(' variable_id '<*' aggregate_source '|' logical_expression ')' !	<!ELEMENT query (%remark;, variable_id, (aggregate_source, logical_expression)) > <!ATTLIST query %ex-prod; "query_expression">
265	real_type = REAL ['(' precision_spec ')'] !	<!ELEMENT real (%remark;, precision_spec?) > <!ATTLIST real %ex-prod; "real_type">
266	referenced_attribute = attribute_ref | qualified_attribute .	<!ENTITY % referenced_attribute “attribute_ref | qualified_attribute” >
267	reference_clause = REFERENCE FROM schema_ref ['(' resource_or_rename { ',' resource_or_rename } ')'] ';' !	<!ELEMENT reference_from (%remark;, schema_ref, (import_all | (constant_import | entity_import | function_import | procedure_import | type_import)+)) > <!ATTLIST reference_from %ex-prod; "reference_clause">
268	rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .	<!ENTITY % rel_op “less_than | greater_than | less_than_or_equal | greater_than_or_equal |

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		not_equal | equal | instance_not_equal | instance_equal” > <!ELEMENT less_than EMPTY > <!ELEMENT greater_than EMPTY> <!ELEMENT less_than_or_equal EMPTY> <!ELEMENT greater_than_or_equal EMPTY> <!ELEMENT not_equal EMPTY> <!ELEMENT equal EMPTY> <!ELEMENT instance_not_equal EMPTY> <!ELEMENT instance_equal EMPTY>
269	rel_op_extended = rel_op | IN | LIKE .	<!ENTITY % rel_op_extended “%rel_op; | in | like” >
270	rename_id = constant_id | entity_id | function_id | procedure_id | type_id .	
271	repeat_control = [increment_control] [while_control] [until_control] .	<!ELEMENT repeat_control (%remark;, increment_control, while?, until?) >
272	repeat_stmt = REPEAT repeat_control ';' stmt { stmt } END_REPEAT ';' !	<!ELEMENT repeat_stmt (%remark;, repeat_control, statement_block) >
273	repetition = numeric_expression .	<!ELEMENT repetition (%remark;, (%numeric_expression_top;)) >
274	resource_or_rename = resource_ref [AS rename_id] .	<!ELEMENT constant_import (%remark;, constant_id, constant_ref) > <!ATTLIST constant_import %ex-prod; "resource_or_rename"> <!ELEMENT entity_import (%remark;, entity_id, entity_ref) > <!ATTLIST entity_import %ex-prod; "resource_or_rename"> <!ELEMENT function_import (%remark;, function_id,

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		function_ref) > <!ATTLIST function_import %ex-prod; "resource_or_rename"> <!ELEMENT procedure_import (%remark;, procedure_id, procedure_ref) > <!ATTLIST procedure_import %ex-prod; "resource_or_rename"> <!ELEMENT type_import (%remark;, type_id, type_ref) > <!ATTLIST type_import %ex-prod; "resource_or_rename"> <!ELEMENT import_all EMPTY> <!ATTLIST import_all %ex-prod; "#NONE">>
275	resource_ref = constant_ref | entity_ref | function_ref | procedure_ref | type_ref .	
276	return_stmt = RETURN ['(' expression ')'] ';' !	<!ELEMENT return_stmt (%remark;, (%expression;)?) >
277	rule_decl = rule_head [algorithm_head] { stmt } where_clause END_RULE ';' !	<!ELEMENT rule_decl (%remark;, %rule_head;, algorithm_head?, statement_block?, where_clause) >
278	rule_head = RULE rule_id FOR '(' entity_ref { ',' entity_ref } ')' ';' .	<!ENTITY % rule_head “rule_id, applies_to_entities” > <!ELEMENT applies_to_entities (%remark;, entity_ref+) > <!ATTLIST applies_to_entities %ex-prod; "rule_head">>
279	rule_id = simple_id !	<!ELEMENT rule_id (#PCDATA) >

280	schema_body = {
	interface_specification } [
	constant_decl] {
	declaration | rule_decl } .
		<!ENTITY % schema_body
		“interface_specification_block?,
		constant_block?,
		(%declaration; |

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		rule_decl)*” >
281	schema_decl = SCHEMA schema_id ';' schema_body END_SCHEMA ';' !	<!ELEMENT schema_decl (%remark;, schema_id, %schema_body;) >
282	schema_id = simple_id !	<!ELEMENT schema_id (#PCDATA) > <!ATTLIST schema_id %def_id;>
283	selector = expression .	<!ENTITY % selector “%expression;” >
284	select_type = SELECT '(' named_types { ',' named_types } ')' !	<!ELEMENT select (%remark;, (%named_types;)+) > <!ATTLIST select %ex-prod; "select_type">
285	set_type = SET [bound_spec] OF base_type !	<!ELEMENT set_type (%remark;, bound_spec?, base_type) >
286	sign = '+' | '-' .	
287	simple_expression = term { add_like_op term } !	<!ENTITY % SIMPLE_EXPRESSION “%TERM; | simple_expression" > <!ELEMENT simple_expression (%remark;, (%add_like_op;), (%TERM;), (%TERM;)) >
288	simple_factor = aggregate_initializer | entity_constructor | enumeration_reference | interval | query_expression | ([unary_op] ('(' expression')' | primary)) .	<!ENTITY % simple_factor “aggregate_initializer | entity_constructor | enumeration_reference | interval | query | %primary; | bracketed_expression | unary_op” > <!ELEMENT bracketed_expression (%remark;, (%expression;)) > <!ATTLIST bracketed_expression

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		%ex-prod; "()">
289	simple_types = binary_type | boolean_type | integer_type | logical_type | number_type | real_type | string_type !	<!ENTITY % simple_types "binary | boolean | integer | logical | number | real | string " >
290	skip_stmt = SKIP ';' !	<!ELEMENT skip_stmt EMPTY >
291	stmt = alias_stmt | assignment_stmt | case_stmt | compound_stmt | escape_stmt | if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt | skip_stmt !	<!ENTITY % stmt "alias_stmt | assignment_stmt | case_stmt | %compound_stmt; | escape_stmt | if_stmt | null_stmt | procedure_call_stmt | repeat_stmt | return_stmt | skip_stmt" > <!ELEMENT statement_block (%remark;, (%stmt;)+) > <!ATTLIST statement_block %ex-prod; "stmt">
292	string_literal = simple_string_literal | encoded_string_literal !	<!ELEMENT string_literal (#PCDATA) >
293	string_type = STRING [width_spec] !	<!ELEMENT string (%remark;, width_spec?) > <!ATTLIST string %ex-prod; "string_type">
294	subsuper = [supertype_constraint] [subtype_declaration] !	<!ENTITY % subsuper "(%supertype_constraint;)?, subtype_of?" >
295	subtype_constraint = OF '(' supertype_expression ')' .	
296	subtype_declaration = SUBTYPE OF '(' entity_ref { ',' entity_ref } ')' !	<!ELEMENT subtype_of (%remark;, entity_ref+) > <!ATTLIST subtype_of %ex-prod; "subtype_declaration">
297	supertype_constraint = abstract_supertype_declaration | supertype_rule .	<!ENTITY % supertype_constraint "abstract_supertype | supertype_of " >
298	supertype_expression = supertype_factor { ANDOR	<!ENTITY % supertype_expression "entity_ref |

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
	supertype_factor } .	supertype_one_of | supertype_and_or | supertype_and “ >
299	supertype_factor = supertype_term { AND supertype_term } .	
300	supertype_rule = SUPERTYPE subtype_constraint .	<!ELEMENT supertype_of (%remark;, (%supertype_expression;)) > <!ATTLIST supertype_of %ex-prod; "supertype_rule"> <!ELEMENT supertype_and_or (%remark;, (%supertype_expression;)+) > <!ATTLIST supertype_and_or %ex-prod; "supertype_rule"> <!ELEMENT supertype_and (%remark;, (%supertype_expression;)+) > <!ATTLIST supertype_and %ex-prod; "supertype_rule">
301	supertype_term = entity_ref | one_of | '(' supertype_expression ')' .	
302	syntax = schema_decl { schema_decl } .	
303	term = factor { multiplication_like_op factor } .	<!ENTITY % TERM “%FACTOR; | term" > <!ELEMENT term ((%multiplication_like_op;), (%TERM;), (%TERM;)) >
304	type_decl = TYPE type_id '=' underlying_type ';' [where_clause] END_TYPE ';' !	<!ELEMENT type_decl (%remark;, type_id, underlying_type, where_clause?) >
305	type_id = simple_id !	<!ELEMENT type_id (#PCDATA) > <!ATTLIST type_id %def_id;>
306	type_label = type_label_id | type_label_ref .	<!ENTITY % type_label “type_label_id |

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
		type_label_ref” >
307	type_label_id = simple_id !	<!ELEMENT type_label_id (#PCDATA) > <!ATTLIST type_label_id %def_id;>
308	unary_op = '+' | '-' | NOT .	<!ENTITY % UNARY_OP "plus | negate | not" > <!ELEMENT unary_op (%remark;, (%UNARY_OP;), (%primary; | bracketed_expression)) > <!ELEMENT plus EMPTY> <!ATTLIST plus %ex-prod; "+"> <!ELEMENT negate EMPTY> <!ATTLIST negate %ex-prod; "-">
309	underlying_type = constructed_types | aggregation_types | simple_types | type_ref .	<!ELEMENT underlying_type (%remark;, (%constructed_types; | %aggregation_types; | %simple_types; | type_ref)) >
310	unique_clause = UNIQUE unique_rule ';' { unique_rule ';' } !	<!ELEMENT unique_clause (%remark;, unique_rule+) >
311	unique_rule = [label ':'] referenced_attribute { ',' referenced_attribute } .	<!ELEMENT unique_rule (%remark;, label?, (%referenced_attribute;)+) >
312	until_control = UNTIL logical_expression !	<!ELEMENT until (%remark;, logical_expression) > <!ATTLIST until %ex-prod; "until_control">
313	use_clause = USE FROM schema_ref ['(' named_type_or_rename { ',' named_type_or_rename } ')'] ';' !	<!ELEMENT use_from (%remark;, schema_ref, (import_all | (entity_import | type_import)+)) > <!ATTLIST use_from %ex-prod; "use_clause">

Express LRM Ref.	Express LRM Syntax	XML DTD element definitions
314	variable_id = simple_id !	<!ELEMENT variable_id (#PCDATA) > <!ATTLIST variable_id %def_id;>
315	where_clause = WHERE domain_rule ';' { domain_rule ';' } !	<!ELEMENT where_clause (%remark;, domain_rule+) >
316	while_control = WHILE logical_expression !	<!ELEMENT while (%remark;, logical_expression) > <!ATTLIST while %ex-prod; "while_control">
317	width = numeric_expression .	<!ENTITY % width “%numeric_expression_top;” >
318	width_spec = '(' width ')' [FIXED] .	<!ELEMENT width_spec (%remark;, (%width;), fixed?) >

Annex G
(informative)

Examples

G.1 Late Bound DTD examples

An example schema that illustrates many features of the EXPRESS language is presented, first in EXPRESS (ISO 10303-11) and then in XML according to the late bound DTD.

G.1.1 Example schema in EXPRESS

```
(* This is an example model created by Alan Williams (The University of
Manchester) to illustrate different aspects of EXPRESS. It has been
modified to have examples of multiple inheritance, type hierarchy and
multiple schemas. *)

SCHEMA mr_jones_garden;

REFERENCE FROM support_items;

CONSTANT
   water_volume_per_fish: INTEGER := 1;
   water_volume_per_amphibian: INTEGER := 2;
   water_volume_per_ornament: INTEGER := 3;
END_CONSTANT;

(*
** The description of the domain states that all the flowers in Mr
** Jones's garden are either coloured red, yellow or white.  For this
** reason, "flower_colour" has been modelled as an enumerated type.
*)
TYPE flower_colour
   = ENUMERATION OF (red, yellow, white);
END_TYPE;

TYPE pond_ornament
   = ENUMERATION OF (waterfall, fountain, bridge);
END_TYPE;

TYPE fish_type
   = ENUMERATION OF (koi, goldfish);
END_TYPE;

TYPE aquatic_plant_types
```

```
   = ENUMERATION OF (lilly, weed, lotus);
END_TYPE;

TYPE amphibian
   = ENUMERATION OF (frog, newt);
END_TYPE;


ENTITY water_treatment_system
   ABSTRACT SUPERTYPE OF (pumping_system AND filtration_system);
   capacity: positive_integer;
END_ENTITY;

ENTITY filtration_system
   SUBTYPE OF (water_treatment_system);
   filtration_efficiency: efficiency;
   number_of_filters: positive_integer;
END_ENTITY;

ENTITY pumping_system
   SUBTYPE OF (water_treatment_system);
   pumping_efficiency: efficiency;
   number_of_pumps: positive_integer;
END_ENTITY;

ENTITY pond
   ABSTRACT SUPERTYPE OF
      (ONEOF (fish_pond, amphibian_pond) ANDOR ornamental_pond);
   maintained_by: OPTIONAL water_treatment_system;
   plants: SET[1:?] OF aquatic_plant;
   water_volume: positive_integer;
   water_ph: ph;
END_ENTITY;

ENTITY fish_pond
   SUBTYPE OF (pond);
   fish: SET [1:?] OF fish_type;
DERIVE
   SELF\pond.water_volume: positive_integer := water_volume_per_fish *
   SIZEOF(fish);
END_ENTITY;

ENTITY amphibian_pond
   SUBTYPE OF (pond);
   amphibians: SET [1:?] OF amphibian;
DERIVE
   SELF\pond.water_volume: positive_integer :=
   water_volume_per_amphibian * SIZEOF(amphibians);
END_ENTITY;

ENTITY ornamental_pond
   SUBTYPE OF (pond);
   ornaments: SET [1:?] OF pond_ornament;
DERIVE
   SELF\pond.water_volume: positive_integer :=
```

```
   water_volume_per_ornament * SIZEOF(ornaments);
END_ENTITY;


(*
** The "garden" is the main entity of the model.  All information
within
** the description which is not related to the garden and its contents
** has been ignored e.g. Mr Jones himself and the local horticultural
** society.

** The set aggregate has been used for the relationship between a
** "garden" and the "beds" in it.  List and array were rejected as
** there
** is no indication within the description of ordering.  Bag was
** rejected
** as a "bed" cannot occur in the "garden" more than once.
*)
ENTITY garden;
   has_pond                    : fish_pond;
   has_greenhouse              : greenhouse;
   climatic_temperature_range  : temperature_range;
   has_beds                    : SET [5 : 5] OF bed;
END_ENTITY;

(*
** The "only_one_garden" rule ensures that the model only allows one
** garden to exist within the domain.
*)
RULE only_one_garden FOR (garden);
WHERE
   mr_jones_has_one_garden :
      SIZEOF (garden) = 1;
END_RULE;

(*
** A "greenhouse" is a greenhouse within Mr Jones's "garden".  It
** contains a set of at least one "greenhouse_plants".  The
"greenhouse"
** also has an associated "temperature".

** The "greenhouse" is constrained to associated with one and only one
** "garden".  Since there is "only_one_garden" in the domain, there is
** no
** need to explicitly constrain the model to only allow one
** "greenhouse".
*)
ENTITY greenhouse;
   enforced_temperature_range  : temperature_range;
   holds_plants                : SET [1 : ?] OF greenhouse_plant;
INVERSE
   the_garden                  : garden FOR has_greenhouse;
END_ENTITY;

(*
```

```
** A "bed" is a flower bed within Mr Jones's "garden".  It contains a
set
** of at least one "outdoors_plant".  It also has an associated "ph"
** value.

** Each "bed" is constrained to associated with one and only one
** "garden".  Since there is "only_one_garden" in the domain, and the
** "garden" is related to exactly five flower "beds" there is no
** need to explicitly constrain the model to only allow only five
** "beds".

** The acidity of each "bed" is different i.e. unique.
*)
ENTITY bed;
   acidity                     : ph;
   holds_plants                : SET [1 : ?] OF outdoors_plant;
INVERSE
   the_garden                  : garden FOR has_beds;
UNIQUE
   every_bed_has_a_different_acidity :
      acidity;
END_ENTITY;

(*
** A "plant" is a flowering plant within either the "greenhouse" or a
** "bed".  These kinds of "plant" are distinct and are represented by
the
** two subtypes of "plant", namely "greenhouse_plant" and
** "outdoors_plant".

** All "plants" have an associated "flower_colour".  They must have a
** unique Latin "plant_name".  In addition they may have one or more
** English "plant_names".  The English "plant_names" are not unique
** since
** several different species of "plants" may have the same English
** "plant_name".

** The description states that none of the "plants" inside the
** "greenhouse" can survive outside it.  This, together with the fact
** that Mr Jones is a keen gardener, has been taken to imply that the
** "plants" he places inside the "greenhouse" can survive there.
** Similarly, the "outdoors_plants" must be able to survive in the
** "garden".  Therefore, the "temperature_range" which a "plant" can
** survive in is relevant to all "plants".
*)
ENTITY plant
   ABSTRACT SUPERTYPE OF
      (ONEOF (greenhouse_plant, outdoors_plant, aquatic_plant));
   colour                      : flower_colour;
   latin_name                  : plant_name;
   english_names               : OPTIONAL SET [1 : ?] OF plant_name;
   survival_temperature_range  : temperature_range;
UNIQUE
   the_latin_name_of_a_plant_species_is_unique :
      latin_name;
```

```
END_ENTITY;

(*
** A "greenhouse_plant" is a "plant" which can only survive within the
** "greenhouse".  In order to ensure that the "greenhouse_plant" may
grow
** within the "greenhouse" it has an associated "temperature" range.

** A "greenhouse_plant" must be inside the "greenhouse".

** A "greenhouse_plant" must be able to survive in the
** "temperature_range" of the "greenhouse".

** A "greenhouse_plant" cannot survive in the "temperature_range" of
** the
** "garden".  Note that this does not prevent the "temperature_ranges"
** of
** the "garden" and "greenhouse" from overlapping.
*)
ENTITY greenhouse_plant
   SUBTYPE OF (plant);
INVERSE
   the_greenhouse              : greenhouse FOR holds_plants;
WHERE
   r1 :
      (* A greenhouse plant can survive in the greenhouse temperature
** *)
       is_sub_range (the_greenhouse.enforced_temperature_range,
                   SELF\plant.survival_temperature_range);

   r2 :
      (* A greenhouse plan cannot survive in the garden temperature *)
      NOT is_sub_range
      (the_greenhouse.the_garden.climatic_temperature_range,
                       SELF\plant.survival_temperature_range);
END_ENTITY;

(*
** An "outdoors_plant" is a "plant" which can survive outside of the
** "greenhouse".  It must be grown in a flower "bed".  In order to
ensure
** that the "outdoors_plant" can cope with the acidity of the "bed" it
** has an associated "ph_range".

** The "ph_range" of the "outdoors_plant" must include the "ph" of the
** "bed".
*)
ENTITY outdoors_plant
   SUBTYPE OF (plant);
   survival_ph_range           : ph_range;
INVERSE
   the_beds                    : SET [1 : ?] OF bed FOR holds_plants;
WHERE
   r1 :
      (* The ph range of the outdoors plant must include the ph value
```

```
** of the
      bed *)
    QUERY (b <* the_beds |
        value_is_within_range (b.acidity, survival_ph_range))
        = the_beds;

   r2 :
      (* An outdoors plant can survive in the garden temperature *)
      is_sub_range (the_beds
      [1].the_garden.climatic_temperature_range,
                  SELF\plant.survival_temperature_range);
END_ENTITY;

ENTITY aquatic_plant
   SUBTYPE OF (plant);
   aquatic_plant_type: aquatic_plant_types;
   oxygen_volumetric_requirement: positive_integer;
   aquatic_plant_size: positive_integer;
END_ENTITY;


(*
** This function checks that a real value lies within the specified
** "real_value_range".
*)
FUNCTION value_is_within_range (v : REAL;
                                r : real_value_range) : BOOLEAN;
   RETURN ( (v >= r.minimum_value) AND (v <= r.maximum_value));
END_FUNCTION;

(*
** This function checks that one "real_value_range" is completely
** contained within another.
*)
FUNCTION is_sub_range (r1,
                       r2 : real_value_range) : BOOLEAN;
   RETURN (value_is_within_range (r1.minimum_value, r2) AND
           value_is_within_range (r1.maximum_value, r2));
END_FUNCTION;

END_SCHEMA;


(* ============================ S C H E M A  ====================== *)

SCHEMA support_items;

(*
** The appropriate base type to use for "temperature" is not clear
from
** the description.  However, given that Mr Jones takes great care of
** "temperature" values it was decided to model it as a real value.
*)
TYPE temperature
    = REAL;
```

```
END_TYPE;

(*
** Both "temperatures" and "ph" values for a plant are within a
** "real_value_range".
*)
ENTITY real_value_range
   ABSTRACT SUPERTYPE OF
      (ONEOF (temperature_range, ph_range));
   minimum_value               : REAL;
   maximum_value               : REAL;
WHERE
   the_values_must_be_sensible :
      maximum_value >= minimum_value;
END_ENTITY;

(*
** A "plant" is able to survive within a "temperature_range".  The
** "garden" will throughout the year have a "temperature_range" which
the
** "outdoors_plants" must be able to survive in.  Similarly, the
** "greenhouse" has a "temperature_range" which the
"greenhouse_plants"
** can survive in.
*)
ENTITY temperature_range
   SUBTYPE OF (real_value_range);
   SELF\real_value_range.minimum_value : temperature;
   SELF\real_value_range.maximum_value : temperature;
END_ENTITY;

(*
** The description mentions both Latin names and English names for
** "plants".  These two `types' of "plant_name" were considered to
have
** sufficient in common to justify their modelling by a single defined
** type.

** It is not clear if some constraints should be specified for
** "plant_name".  For example, it may sensible to ensure that it is
** not
** empty and that it is alphabetic.
*)
TYPE plant_name
    = STRING;
END_TYPE;

(*
** The acidity of the "beds" and the range acceptable to
** "outdoors_plants" has been modelled by the defined type "ph".  It
is
** not clear which base type should be used for "ph"; however, given
that
** Mr Jones takes pride in compatibility of values, it was decided to
use
```

```
** the real type.

** A "ph" value is constrained to being between zero and fourteen.  In
** reality, it is highly unlikely that Mr Jones would ever use "ph"
** values near the extremes.  However, there is no indication of more
** realistic limits within the domain description.
*)
TYPE ph
   = REAL;
WHERE
   the_ph_is_between_0_and_14 :
     {0 <= SELF <= 14};
END_TYPE;

(*
** An "outdoors_plant", indeed any "plant", can survive a range of
"ph"
** values.
*)
ENTITY ph_range
   SUBTYPE OF (real_value_range);
   SELF\real_value_range.minimum_value : ph;
   SELF\real_value_range.maximum_value : ph;
END_ENTITY;

TYPE positive_integer = INTEGER;
WHERE
   positive_only: SELF > 0;
END_TYPE;

TYPE percentage = REAL;
WHERE
   between_0_and_100: {0.0 <= SELF <= 100.0};
END_TYPE;

TYPE fractional_value = REAL;
WHERE
   between_0_and_1: {0.0 <= SELF <= 1.0};
END_TYPE;

TYPE efficiency = SELECT (percentage, fractional_value);
END_TYPE;

END_SCHEMA;
```

G.1.2 Example schema in XML

This schema presented in the previous sub-clause is encoded in XML as follows:

```
<?xml version="1.0"?>
<!DOCTYPE express_driven_data SYSTEM "express-dtd-v6.dtd">
<express_driven_data>
  <schema_decl>
    <embedded_remark>This is an example model created by Alan
```

```
Williams (The
      University of Manchester) to illustrate different
aspects of EXPRESS. It has
      been modified to have examples of multiple
inheritance, type hierarchy and
      multiple schemas.</embedded_remark>
    <schema_id>mr_jones_garden</schema_id>
    <interface_specification_block>
      <reference_from>
        <schema_ref>support_items</schema_ref><import_all/>
      </reference_from>
    </interface_specification_block>
    <constant_block>
      <constant_decl>
  <constant_id>water_volume_per_fish</constant_id>
        <base_type><integer/>
        </base_type>
        <integer_literal>1</integer_literal>
      </constant_decl>
      <constant_decl>
  <constant_id>water_volume_per_amphibian</constant_id>
        <base_type><integer/>
        </base_type>
        <integer_literal>2</integer_literal>
      </constant_decl>
      <constant_decl>
        <constant_id>water_volume_per_ornament</constant_id>
        <base_type><integer/>
        </base_type>
        <integer_literal>3</integer_literal>
      </constant_decl>
    </constant_block>
    <type_decl>
      <embedded_remark>** The description of the domain
states that all the
        flowers in Mr ** Jones's garden are either
coloured red, yellow or white. For
        this ** reason, "flower_colour" has been
modelled as an enumerated type.
        </embedded_remark>
      <type_id>flower_colour</type_id>
      <underlying_type>
        <enumeration>
          <enumeration_id>red</enumeration_id>
  <enumeration_id>yellow</enumeration_id>
          <enumeration_id>white</enumeration_id>
        </enumeration>
      </underlying_type>
    </type_decl>
    <type_decl>
      <type_id>pond_ornament</type_id>
      <underlying_type>
        <enumeration>
  <enumeration_id>waterfall</enumeration_id>
  <enumeration_id>fountain</enumeration_id>
```

```
<enumeration_id>bridge</enumeration_id>
      </enumeration>
    </underlying_type>
  </type_decl>
  <type_decl>
    <type_id>fish_type</type_id>
    <underlying_type>
      <enumeration>
        <enumeration_id>lilly</enumeration_id>
        <enumeration_id>weed</enumeration_id>
        <enumeration_id>lotus</enumeration_id>
      </enumeration>
    </underlying_type>
  </type_decl>
  <type_decl>
    <type_id>amphibian</type_id>
    <underlying_type>
      <enumeration>
        <enumeration_id>frog</enumeration_id>
        <enumeration_id>newt</enumeration_id>
      </enumeration>
    </underlying_type>
  </type_decl>
  <entity_decl>
    <entity_id>water_treatment_system</entity_id>
    <abstract_supertype>
      <supertype_and>
<entity_ref>pumping_system</entity_ref>
<entity_ref>filtration_system</entity_ref>
      </supertype_and>
    </abstract_supertype>
    <explicit_attr_block>
      <explicit_attr>
        <attribute_id>capacity</attribute_id>
        <base_type>
<type_ref>positive_integer</type_ref>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
  </entity_decl>
  <entity_decl>
    <entity_id>filtration_system</entity_id>
    <subtype_of>
<entity_ref>water_treatment_system</entity_ref>
    </subtype_of>
    <explicit_attr_block>
      <explicit_attr>
<attribute_id>filtration_efficiency</attribute_id>
        <base_type>
<type_ref>efficiency</type_ref>
        </base_type>
      </explicit_attr>
      <explicit_attr>
<attribute_id>number_of_filters</attribute_id>
        <base_type>
```

```
<type_ref>positive_integer</type_ref>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
  </entity_decl>
  <entity_decl>
    <entity_id>pumping_system</entity_id>
    <subtype_of>
<entity_ref>water_treatment_system</entity_ref>
    </subtype_of>
    <explicit_attr_block>
      <explicit_attr>
<attribute_id>pumping_efficiency</attribute_id>
        <base_type>
<type_ref>efficiency</type_ref>
        </base_type>
      </explicit_attr>
      <explicit_attr>
<attribute_id>number_of_pumps</attribute_id>
        <base_type>
<type_ref>positive_integer</type_ref>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
  </entity_decl>
  <entity_decl>
    <entity_id>pond</entity_id>
    <abstract_supertype>
      <supertype_and_or>
        <supertype_one_of>
<entity_ref>fish_pond</entity_ref>
<entity_ref>amphibian_pond</entity_ref>
        </supertype_one_of>
<entity_ref>ornamental_pond</entity_ref>
      </supertype_and_or>
    </abstract_supertype>
    <explicit_attr_block>
      <explicit_attr>
        <attribute_id>maintained_by</attribute_id><optional/>
        <base_type>
<entity_ref>water_treatment_system</entity_ref>
        </base_type>
      </explicit_attr>
      <explicit_attr>
        <attribute_id>plants</attribute_id>
        <base_type>
          <set_type>
            <bound_spec>
              <lower_bound>
<integer_literal>1</integer_literal>
              </lower_bound>
<upper_bound><indeterminate/>
              </upper_bound>
            </bound_spec>
            <base_type>
```

```
<entity_ref>aquatic_plant</entity_ref>
            </base_type>
          </set_type>
        </base_type>
      </explicit_attr>
      <explicit_attr>
<attribute_id>water_volume</attribute_id>
        <base_type>
<type_ref>positive_integer</type_ref>
        </base_type>
      </explicit_attr>
      <explicit_attr>
        <attribute_id>water_ph</attribute_id>
        <base_type>
          <type_ref>ph</type_ref>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
  </entity_decl>
  <entity_decl>
    <entity_id>fish_pond</entity_id>
    <subtype_of>
      <entity_ref>pond</entity_ref>
    </subtype_of>
    <explicit_attr_block>
      <explicit_attr>
        <attribute_id>fish</attribute_id>
        <base_type>
          <set_type>
            <bound_spec>
              <lower_bound>
                <integer_literal>1</integer_literal>
              </lower_bound>
                <upper_bound><indeterminate/>
              </upper_bound>
            </bound_spec>
            <base_type>
              <type_ref>fish_type</type_ref>
            </base_type>
          </set_type>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
    <derive_clause>
      <derived_attr>
        <qualified_attribute>
          <entity_ref>pond</entity_ref>
            <attribute_ref>water_volume</attribute_ref>
        </qualified_attribute>
        <base_type>
          <type_ref>positive_integer</type_ref>
        </base_type>
        <term><multiply/>
          <constant_ref>water_volume_per_fish</constant_ref>
          <function_call><sizeof/>
```

```
            <attribute_ref>fish</attribute_ref>
          </function_call>
        </term>
      </derived_attr>
    </derive_clause>
  </entity_decl>
  <entity_decl>
    <entity_id>amphibian_pond</entity_id>
    <subtype_of>
      <entity_ref>pond</entity_ref>
    </subtype_of>
    <explicit_attr_block>
      <explicit_attr>
        <attribute_id>amphibians</attribute_id>
        <base_type>
          <set_type>
            <bound_spec>
              <lower_bound>
                <integer_literal>1</integer_literal>
              </lower_bound>
                <upper_bound><indeterminate/>
              </upper_bound>
            </bound_spec>
            <base_type>
              <type_ref>amphibian</type_ref>
            </base_type>
          </set_type>
        </base_type>
      </explicit_attr>
    </explicit_attr_block>
    <derive_clause>
      <derived_attr>
        <qualified_attribute>
          <entity_ref>pond</entity_ref>
            <attribute_ref>water_volume</attribute_ref>
        </qualified_attribute>
        <base_type>
          <type_ref>positive_integer</type_ref>
        </base_type>
        <term><multiply/>
        <constant_ref>water_volume_per_amphibian</constant_ref>
          <function_call><sizeof/>
            <attribute_ref>amphibians</attribute_ref>
          </function_call>
        </term>
      </derived_attr>
    </derive_clause>
  </entity_decl>
  <entity_decl>
    <entity_id>ornamental_pond</entity_id>
    <subtype_of>
      <entity_ref>pond</entity_ref>
    </subtype_of>
    <explicit_attr_block>
      <explicit_attr>
```

```
          <attribute_id>ornaments</attribute_id>
          <base_type>
            <set_type>
              <bound_spec>
                <lower_bound>
                  <integer_literal>1</integer_literal>
                </lower_bound>
                  <upper_bound><indeterminate/>
                </upper_bound>
              </bound_spec>
              <base_type>
                <type_ref>pond_ornament</type_ref>
              </base_type>
            </set_type>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <derive_clause>
        <derived_attr>
          <qualified_attribute>
            <entity_ref>pond</entity_ref>
            <attribute_ref>water_volume</attribute_ref>
          </qualified_attribute>
          <base_type>
            <type_ref>positive_integer</type_ref>
          </base_type>
          <term><multiply/>
            <constant_ref>water_volume_per_ornament</constant_ref>
            <function_call><sizeof/>
              <attribute_ref>ornaments</attribute_ref>
            </function_call>
          </term>
        </derived_attr>
      </derive_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** The "garden" is the main entity
of the model. All information within
** the description which is not related to the garden and its contents
** has been ignored e.g. Mr Jones himself and the local  horticultural
** society.
** The set aggregate has been used for the relationship between a
** "garden" and the "beds" in it. List and array were rejected as
** there ** is no indication within the description of ordering. Bag was
** rejected
** as a "bed" cannot occur in the "garden" more than once.
        </embedded_remark>
      <entity_id>garden</entity_id>
      <explicit_attr_block>
        <explicit_attr>
          <attribute_id>has_pond</attribute_id>
          <base_type>
            <entity_ref>fish_pond</entity_ref>
          </base_type>
        </explicit_attr>
```

```
        <explicit_attr>
          <attribute_id>has_greenhouse</attribute_id>
          <base_type>
            <entity_ref>greenhouse</entity_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
          <attribute_id>climatic_temperature_range</attribute_id>
          <base_type>
            <entity_ref>temperature_range</entity_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
          <attribute_id>has_beds</attribute_id>
          <base_type>
            <set_type>
              <bound_spec>
                <lower_bound>
                  <integer_literal>5</integer_literal>
                </lower_bound>
                <upper_bound>
                  <integer_literal>5</integer_literal>
                </upper_bound>
              </bound_spec>
              <base_type>
                <entity_ref>bed</entity_ref>
              </base_type>
            </set_type>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
    </entity_decl>
    <rule_decl>
      <embedded_remark>** The "only_one_garden" rule ensures that the model
        only allows one ** garden to exist within
the domain. </embedded_remark>
      <rule_id>only_one_garden</rule_id>
      <applies_to_entities>
        <entity_ref>garden</entity_ref>
      </applies_to_entities>
      <where_clause>
        <domain_rule>
          <label>mr_jones_has_one_garden</label>
          <logical_expression>
            <relation_expression><equal/>
              <function_call><sizeof/>
                <population>
                  <entity_ref>garden</entity_ref>
                </population>
              </function_call>
              <integer_literal>1</integer_literal>
            </relation_expression>
          </logical_expression>
        </domain_rule>
      </where_clause>
```

```
    </rule_decl>
    <entity_decl>
      <embedded_remark>** A "greenhouse" is a greenhouse
within Mr Jones's
        "garden". It ** contains a set of at least
one "greenhouse_plants". The
        "greenhouse" ** also has an associated
"temperature". ** The "greenhouse" is
        constrained to associated with one and only
one ** "garden". Since there is
        "only_one_garden" in the domain, there is
** no ** need to explicitly constrain
        the model to only allow one **
"greenhouse". </embedded_remark>
      <entity_id>greenhouse</entity_id>
      <explicit_attr_block>
        <explicit_attr>
  <attribute_id>enforced_temperature_range</attribute_id>
          <base_type>
  <type_ref>temperature_range</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>holds_plants</attribute_id>
          <base_type>
            <set_type>
              <bound_spec>
                <lower_bound>
  <integer_literal>1</integer_literal>
                </lower_bound>
  <upper_bound><indeterminate/>
                </upper_bound>
              </bound_spec>
              <base_type>
  <entity_ref>greenhouse_plant</entity_ref>
              </base_type>
            </set_type>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <inverse_clause>
        <inverse_attr>
  <attribute_id>the_garden</attribute_id>
          <entity_ref>garden</entity_ref>
          <attribute_ref>has_greenhouse</attribute_ref>
        </inverse_attr>
      </inverse_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** A "bed" is a flower bed within
Mr Jones's "garden".
        It contains a set ** of at least one
"outdoors_plant". It also has an
        associated "ph" ** value. ** Each "bed" is
constrained to associated with one
```

```
        and only one ** "garden". Since there is
"only_one_garden" in the domain, and
        the ** "garden" is related to exactly five
flower "beds" there is no **need to
        explicitly constrain the model to only
allow only five ** "beds". ** The
        acidity of each "bed" is different i.e.
unique. </embedded_remark>
      <entity_id>bed</entity_id>
      <explicit_attr_block>
        <explicit_attr>
          <attribute_id>acidity</attribute_id>
          <base_type>
            <type_ref>ph</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>holds_plants</attribute_id>
          <base_type>
            <set_type>
              <bound_spec>
                <lower_bound>
  <integer_literal>1</integer_literal>
                </lower_bound>
  <upper_bound><indeterminate/>
                </upper_bound>
              </bound_spec>
              <base_type>
  <entity_ref>outdoors_plant</entity_ref>
              </base_type>
            </set_type>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <inverse_clause>
        <inverse_attr>
  <attribute_id>the_garden</attribute_id>
          <entity_ref>garden</entity_ref>
  <attribute_ref>has_beds</attribute_ref>
        </inverse_attr>
      </inverse_clause>
      <unique_clause>
        <unique_rule>
  <label>every_bed_has_a_different_acidity</label>
          <attribute_ref>acidity</attribute_ref>
        </unique_rule>
      </unique_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** A "plant" is a flowering plant
within either the
        "greenhouse" or a ** "bed". These kinds of
"plant" are distinct and are
        represented by the ** two subtypes of
"plant", namely "greenhouse_plant" and **
```

```
        "outdoors_plant". ** All "plants" have an associated
"flower_colour".
They must
        have a ** unique Latin "plant_name". In
addition they may have one or more **
        English "plant_names". The English
"plant_names" are not unique ** since **
        several different species of "plants" may
have the same English **
        "plant_name". ** The description states
that none of the "plants" inside the **
        "greenhouse" can survive outside it. This,
together with the fact ** that Mr
        Jones is a keen gardener, has been taken to
imply that the ** "plants" he
        places inside the "greenhouse" can survive
there. ** Similarly, the
        "outdoors_plants" must be able to survive
in the ** "garden". Therefore, the
        "temperature_range" which a "plant" can **
survive in is relevant to all
        "plants".</embedded_remark>
      <entity_id>plant</entity_id>
      <abstract_supertype>
        <supertype_one_of>
  <entity_ref>greenhouse_plant</entity_ref>
  <entity_ref>outdoors_plant</entity_ref>
          <entity_ref>aquatic_plant</entity_ref>
        </supertype_one_of>
      </abstract_supertype>
      <explicit_attr_block>
        <explicit_attr>
          <attribute_id>colour</attribute_id>
          <base_type>
  <type_ref>flower_colour</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>latin_name</attribute_id>
          <base_type>
  <type_ref>plant_name</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>english_names</attribute_id><optional/>
          <base_type>
            <set_type>
              <bound_spec>
                <lower_bound>
  <integer_literal>1</integer_literal>
                </lower_bound>
  <upper_bound><indeterminate/>
                </upper_bound>
              </bound_spec>
              <base_type>
```

```
  <type_ref>plant_name</type_ref>
              </base_type>
            </set_type>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>survival_temperature_rnage</attribute_id>
          <base_type>
  <entity_ref>temperature_range</entity_ref>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <unique_clause>
        <unique_rule>
  <label>the_latin_name_of_a_plant_is_unique</label>
  <attribute_ref>latin_name</attribute_ref>
        </unique_rule>
      </unique_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** A "greenhouse_plant" is a
"plant" which can only
        survive within the ** "greenhouse". In
order to ensure that the
        "greenhouse_plant" may grow ** within the
"greenhouse" it has an associated
        "temperature" range. ** A "greenhouse_plant" must be inside the
"greenhouse".
        ** A "greenhouse_plant" must be able to
survive in the ** "temperature_range"
        of the "greenhouse". ** A
"greenhouse_plant" cannot survive in the
        "temperature_range" of ** the ** "garden".
Note that this does not prevent the
        "temperature_ranges" ** of ** the "garden"
and "greenhouse" from overlapping.
        *) </embedded_remark>
      <entity_id>greenhouse_plant</entity_id>
      <subtype_of>
        <entity_ref>plant</entity_ref>
      </subtype_of>
      <inverse_clause>
        <inverse_attr>
  <attribute_id>the_greenhouse</attribute_id>
          <entity_ref>greenhouse</entity_ref>
  <attribute_ref>holds_plants</attribute_ref>
        </inverse_attr>
      </inverse_clause>
      <where_clause>
        <domain_rule>
          <label>r1</label>
          <logical_expression>
            <embedded_remark>A
greenhouse plant can survive in the greenhouse
  temperature</embedded_remark>
```

```
            <function_call>
  <function_ref>is_sub_range</function_ref>
              <qualified_factor>
  <attribute_ref>the_greenhouse</attribute_ref>
                <qualifier>
  <attribute_ref>enforced_temperature_range</attribute_ref>
                </qualifier>
              </qualified_factor>
  <qualified_factor><self/>
                <qualifier>
  <entity_ref>plant</entity_ref>
  <attribute_ref>survival_temperature_range</attribute_ref>
                </qualifier>
              </qualified_factor>
            </function_call>
          </logical_expression>
        </domain_rule>
        <domain_rule>
          <label>r2</label>
          <logical_expression>
            <embedded_remark>A
greenhouse plan cannot survive in the garden
  temperature</embedded_remark>
            <unary_op><not/>
              <function_call>
  <function_ref>is_sub_range</function_ref>
  <qualified_factor>
  <attribute_ref>the_greenhouse</attribute_ref>
  <qualifier>
  <attribute_ref>the_garden</attribute_ref>
  <attribute_ref>climatic_temperature_range</attribute_ref>
                  </qualifier>
  </qualified_factor>
  <qualified_factor><self/>
  <qualifier>
  <entity_ref>plant</entity_ref>
  <attribute_ref>survival_temperature_range</attribute_ref>
  </qualifier>
  </qualified_factor>
              </function_call>
            </unary_op>
          </logical_expression>
        </domain_rule>
      </where_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** An "outdoors_plant" is a
"plant" which can survive
        outside of the ** "greenhouse". It must be grown in a flower "bed".
In
order to
        ensure ** that the "outdoors_plant" can
cope with the acidity of the "bed" it
        ** has an associated "ph_range". ** The
"ph_range" of the "outdoors_plant" must
```

```
        include the "ph" of the ** "bed".
        <embedded_remark>ENTITY outdoors_plant
SUBTYPE OF (plant);
          survival_ph_range : ph_range;
INVERSE the_beds : SET [1 : ?] OF bed FOR
          holds_plants; WHERE r1 : (* *)
QUERY (b &lt;* the_beds | value_is_within_range
          (b.acidity, survival_ph_range)) = the_beds; r2 : (* An outdoors
plant can
          survive in the garden temperature
*) is_sub_range (the_beds
  [1].the_garden.climatic_temperature_range,
  SELF\plant.survival_temperature_range); END_ENTITY;
          </embedded_remark></embedded_remark>
      <entity_id>outdoors_plant</entity_id>
      <subtype_of>
        <entity_ref>plant</entity_ref>
      </subtype_of>
      <explicit_attr_block>
        <explicit_attr>
  <attribute_id>survival_ph_range</attribute_id>
          <base_type>
            <entity_ref>ph_range</entity_ref>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <inverse_clause>
        <inverse_attr>
          <attribute_id>the_beds</attribute_id>
          <entity_ref>bed</entity_ref>
  <attribute_ref>holds_plants</attribute_ref>
          <inverse_set>
            <bound_spec>
              <lower_bound>
  <integer_literal>1</integer_literal>
              </lower_bound>
  <upper_bound><indeterminate/>
              </upper_bound>
            </bound_spec>
          </inverse_set>
        </inverse_attr>
      </inverse_clause>
      <where_clause>
        <domain_rule>
          <embedded_remark>The ph range of
the outdoors plant must include the
            ph value ** of the bed
</embedded_remark>
          <label>r1</label>
          <logical_expression>
            <relation_expression><equal/>
              <query>
  <variable_id>b</variable_id>
  <aggregate_source>
  <attribute_ref>the_beds</attribute_ref>
```

```
  </aggregate_source>
  <logical_expression>
  <function_call>
  <function_ref>value_is_within_range</function_ref>
  <qualified_factor>
    <variable_ref>b</variable_ref>
    <qualifier>
      <attribute_ref>acidity</attribute_ref>
    </qualifier>
  </qualified_factor>
  <attribute_ref>survival_ph_range</attribute_ref>
  </function_call>
  </logical_expression>
              </query>
              <population>
  <entity_ref>the_beds</entity_ref>
              </population>
            </relation_expression>
          </logical_expression>
        </domain_rule>
        <domain_rule>
          <embedded_remark>An outdoors plant
can survive in the garden
            temperature </embedded_remark>
          <logical_expression>
            <function_call>
  <function_ref>is_sub_range</function_ref>
              <qualified_factor>
                <population>
  <entity_ref>the_beds</entity_ref>
                </population>
                <qualifier>
  <index_qualifier>
  <low_index>
    <integer_literal>1</integer_literal>
  </low_index>
  </index_qualifier>
  <attribute_ref>the_garden</attribute_ref>
  <attribute_ref>climatic_temperature_range</attribute_ref>
                </qualifier>
              </qualified_factor>
  <qualified_factor><self/>
                <qualifier>
  <entity_ref>plant</entity_ref>
  <attribute_ref>survival_temperature_range</attribute_ref>
                </qualifier>
              </qualified_factor>
            </function_call>
          </logical_expression>
        </domain_rule>
      </where_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>ENTITY aquatic_plant SUBTYPE OF
(plant);
```

```
        aquatic_plant_type: aquatic_plant_types;
oxygen_volumetric_requirement:
        positive_integer; aquatic_plant_size:
positive_integer; END_ENTITY;
        </embedded_remark>
      <entity_id>aquatic_plant</entity_id>
      <subtype_of>
        <entity_ref>plant</entity_ref>
      </subtype_of>
      <explicit_attr_block>
        <explicit_attr>
  <attribute_id>aquatic_plant_type</attribute_id>
          <base_type>
            <type_ref>aquatic_plant_types</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>oxygen_volumetric_requirement</attribute_id>
          <base_type>
  <type_ref>positive_integer</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
  <attribute_id>aquatic_plant_size</attribute_id>
          <base_type>
  <type_ref>positive_integer</type_ref>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
    </entity_decl>
    <function_decl>
      <embedded_remark>** This function checks that a real value lies
within
        the specified ** "real_value_range".
</embedded_remark>
      <function_id>value_is_within_range</function_id>
      <formal_parameter_block>
        <formal_parameter>
          <parameter_id>v</parameter_id>
          <real>
          </real>
        </formal_parameter>
        <formal_parameter>
          <parameter_id>r</parameter_id>
  <entity_ref>real_value_range</entity_ref>
        </formal_parameter>
      </formal_parameter_block>
      <function_return_type><boolean/>
      </function_return_type>
      <statement_block>
        <return_stmt>
          <term><and/>
            <bracketed_expression>
  <relation_expression><greater_than_or_equal/>
  <parameter_ref>v</parameter_ref>
```

```
  <qualified_factor>
  <parameter_ref>r</parameter_ref>
  <qualifier>
  <attribute_ref>minimum_value</attribute_ref>
  </qualifier>
  </qualified_factor>
              </relation_expression>
            </bracketed_expression>
            <bracketed_expression>
  <relation_expression><less_than_or_equal/>
  <parameter_ref>v</parameter_ref>
  <qualified_factor>
  <parameter_ref>r</parameter_ref>
  <qualifier>
  <attribute_ref>maximum_value</attribute_ref>
  </qualifier>
  </qualified_factor>
              </relation_expression>
            </bracketed_expression>
          </term>
        </return_stmt>
      </statement_block>
    </function_decl>
    <function_decl>
      <embedded_remark>** This function checks that one "real_value_range"
is
        completely ** contained within another.
</embedded_remark>
      <function_id>is_sub_range</function_id>
      <formal_parameter_block>
        <formal_parameter>
          <parameter_id>r1</parameter_id>
  <entity_ref>real_value_range</entity_ref>
        </formal_parameter>
        <formal_parameter>
          <parameter_id>r2</parameter_id>
  <entity_ref>real_value_range</entity_ref>
        </formal_parameter>
      </formal_parameter_block>
      <function_return_type><boolean/>
      </function_return_type>
      <statement_block>
        <return_stmt>
          <bracketed_expression>
            <term><and/>
              <function_call>
  <function_ref>value_is_within_range</function_ref>
  <qualified_factor>
  <parameter_ref>r1</parameter_ref>
  <qualifier>
  <attribute_ref>minimum_value</attribute_ref>
  </qualifier>
  </qualified_factor>
  <parameter_ref>r2</parameter_ref>
              </function_call>
```

```
              <function_call>

  <function_ref>value_is_within_range</function_ref>
                <qualified_factor>
  <parameter_ref>r1</parameter_ref>
  <qualifier>
  <attribute_ref>maximum_value</attribute_ref>
  </qualifier>
  </qualified_factor>
  <parameter_ref>r2</parameter_ref>
              </function_call>
            </term>
          </bracketed_expression>
        </return_stmt>
      </statement_block>
    </function_decl>
  </schema_decl>
  <schema_decl>
    <embedded_remark>============================ S C H E M A
      ====================== </embedded_remark>
    <schema_id>support_items</schema_id>
    <type_decl>
      <embedded_remark>** The appropriate base type to
use for "temperature" is
        not clear from ** the description. However,
given that Mr Jones takes great
        care of ** "temperature" values it was
decided to model it as a real value.
        </embedded_remark>
      <type_id>temperature</type_id>
      <underlying_type>
        <real>
        </real>
      </underlying_type>
    </type_decl>
    <entity_decl>
      <embedded_remark>** Both "temperatures" and "ph"
values for a plant are
        within a ** "real_value_range". </embedded_remark>
      <entity_id>real_value_range</entity_id>
      <abstract_supertype>
        <supertype_one_of>
  <entity_ref>temperature_range</entity_ref>
          <entity_ref>ph_range</entity_ref>
        </supertype_one_of>
      </abstract_supertype>
      <explicit_attr_block>
        <explicit_attr>
  <attribute_id>minimum_value</attribute_id>
          <base_type>
            <real>
            </real>
          </base_type>
        </explicit_attr>
        <explicit_attr>
```

```
          <attribute_id>maximum_value</attribute_id>
          <base_type>
            <real>
            </real>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
      <where_clause>
        <domain_rule>
  <label>the_values_must_be_sensible</label>
          <logical_expression>
  <relation_expression><greater_than_or_equal/>
  <attribute_ref>maximum_value</attribute_ref>
  <attribute_ref>minimum_value</attribute_ref>
            </relation_expression>
          </logical_expression>
        </domain_rule>
      </where_clause>
    </entity_decl>
    <entity_decl>
      <embedded_remark>** A "plant" is able to survive within a
        "temperature_range". The ** "garden" will
throughout the year have a
        "temperature_range" which the **
"outdoors_plants" must be able to survive in.
        Similarly, the ** "greenhouse" has a
"temperature_range" which the
        "greenhouse_plants" ** can survive in.
</embedded_remark>
      <entity_id>temperature_range</entity_id>
      <subtype_of>
        <entity_ref>real_value_range</entity_ref>
      </subtype_of>
      <explicit_attr_block>
        <explicit_attr>
          <qualified_attribute>
  <entity_ref>real_value_range</entity_ref>
  <attribute_ref>minimum_value</attribute_ref>
          </qualified_attribute>
          <base_type>
  <type_ref>temperature</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
          <qualified_attribute>
  <entity_ref>real_value_range</entity_ref>
  <attribute_ref>maximum_value</attribute_ref>
          </qualified_attribute>
          <base_type>
  <type_ref>temperature</type_ref>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
    </entity_decl>
    <type_decl>
```

```
      <embedded_remark>** The description mentions both
Latin names and English
        names for ** "plants". These two `types' of
"plant_name" were considered to
        have ** sufficient in common to justify
their modelling by a single defined **
        type. ** It is not clear if some
constraints should be specified for **
        "plant_name". For example, it may sensible
to ensure that it is ** not ** empty
        and that it is alphabetic. </embedded_remark>
      <type_id>plant_name</type_id>
      <underlying_type>
        <string>
        </string>
      </underlying_type>
    </type_decl>
    <type_decl>
      <embedded_remark>** The acidity of the "beds" and
the range acceptable to
        ** "outdoors_plants" has been modelled by
the defined type "ph". It is ** not
        clear which base type should be used for
"ph"; however, given that ** Mr Jones
        takes pride in compatibility of values, it
was decided to use ** the real type.
        ** A "ph" value is constrained to being between zero and fourteen.
In **
        reality, it is highly unlikely that Mr
Jones would ever use "ph" ** values near
        the extremes. However, there is no
indication of more ** realistic limits
        within the domain description.
</embedded_remark>
      <type_id>ph</type_id>
      <underlying_type>
        <real>
        </real>
      </underlying_type>
      <where_clause>
        <domain_rule>
  <label>the_ph_is_between_0_and_14</label>
          <logical_expression>
            <interval>
  <interval_low_inclusive>
                <integer_literal>0</integer_literal>
  </interval_low_inclusive>
              <interval_item><self/>
              </interval_item>
  <interval_high_inclusive>
  <integer_literal>14</integer_literal>
  </interval_high_inclusive>
            </interval>
          </logical_expression>
        </domain_rule>
```

```
      </where_clause>
    </type_decl>
    <entity_decl>
      <embedded_remark>** An "outdoors_plant", indeed any
"plant", can survive
        a range of "ph" ** values. </embedded_remark>
      <entity_id>ph_range</entity_id>
      <subtype_of>
        <entity_ref>real_value_range</entity_ref>
      </subtype_of>
      <explicit_attr_block>
        <explicit_attr>
          <qualified_attribute>
  <entity_ref>real_value_range</entity_ref>
  <attribute_ref>minimum_value</attribute_ref>
          </qualified_attribute>
          <base_type>
            <type_ref>ph</type_ref>
          </base_type>
        </explicit_attr>
        <explicit_attr>
          <qualified_attribute>
  <entity_ref>real_value_range</entity_ref>
  <attribute_ref>maximum_value</attribute_ref>
          </qualified_attribute>
          <base_type>
            <type_ref>ph</type_ref>
          </base_type>
        </explicit_attr>
      </explicit_attr_block>
    </entity_decl>
    <type_decl>
      <type_id>positive_integer</type_id>
      <underlying_type><integer/>
      </underlying_type>
      <where_clause>
        <domain_rule>
          <label>positive_only</label>
          <logical_expression>
  <relation_expression><greater_than/><self/>
  <integer_literal>0</integer_literal>
            </relation_expression>
          </logical_expression>
        </domain_rule>
      </where_clause>
    </type_decl>
    <type_decl>
      <type_id>percentage</type_id>
      <underlying_type>
        <real>
        </real>
      </underlying_type>
      <where_clause>
        <domain_rule>
          <label>between_0_and_100</label>
```

```
          <logical_expression>
            <interval>
  <interval_low_inclusive>
  <real_literal>0.0</real_literal>
  </interval_low_inclusive>
              <interval_item><self/>
              </interval_item>
  <interval_high_inclusive>
  <real_literal>100.0</real_literal>
              </interval_high_inclusive>
            </interval>
          </logical_expression>
        </domain_rule>
      </where_clause>
    </type_decl>
    <type_decl>
      <type_id>efficiency</type_id>
      <underlying_type>
        <select>
          <type_ref>percentage</type_ref>
          <type_ref>fractional_value</type_ref>
        </select>
      </underlying_type>
    </type_decl>
  </schema_decl>
</express_driven_data>
```

Annex H
(tbd)

EXPRESS schema interchange using the OMG XMI standard

The text of this annex has been provided by Dave Price (IBM and Pdes, Inc.) It is incomplete and has not been edited.

H.1 Purpose

The Object Management Group has standardized the XML Meta-data Interchange format called XMI. XMI provides a mechanism for the interchange of various types of meta-data using XML syntax. XMI includes, among other things, the capability to interchange data type information, class information, groupings of classes providing namespaces for the classes, associations between classes and inheritance between classes. This capability is built upon another OMG standard called the Meta-Object Facility or MOF. XMI compliant DTDs are based on MOF models of a modeling language. OMG has also standardized an XMI compliant DTD for the Unified Modeling Language or UML.

The purpose of this annex is to enable to use of OMGs XMI standard for the interchange of EXPRESS schemas between XMI compliant systems used to design and implement software systems. The use of XMI for that purpose is achieved by defining a one-way mapping of a subset of the EXPRESS language into the OMG XMI compliant UML DTD. In most cases, the strategy for specifying this mapping is to specify the UML concept into which an EXPRESS concept is mapped and relying on the OMG XMI standard to specify how that UML concept is encoded in the XMI compliant UML DTD. In some cases, the mapping of the EXPRESS concept directly into the XMI compliant UML DTD is specified.

All mappings into UML are into concepts that would appear in Static Class Diagrams. Any reference to a UML concept is a reference to the definition of that concept as found in OMG Unified Modeling Language Specification Version 1.3, June 1999 and clause 6 of that specification is the UML XMI DTD Specification. The introduction in that DTD is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- XMI Automatic DTD Generation -->
<!-- Date: Fri May 28 14:48:41 PDT 1999 -->
<!-- Metamodel: UML1.3 -->
```

NOTE - In order to completely implement an EXPRESS schema ,the EXPRESS concepts not mapped will likely have to be implemented by a software developer in application code not automatically generated by an EXPRESS parser.

```
<<From here on in are text that describe the mapping of EXPRESS concepts
into UML or the UML DTD itself. For example:>
```

H.2 Mapping EXPRESS simple data types

The EXPRESS STRING data type shall be mapped into the UML String data type.

H.3 Mapping EXPRESS ENTITY data types and their attributes

Each EXPRESS ENTITY concept shall be mapped as a UML Class where the name of the Class is the name of the ENTITY.

H.4 Mapping EXPRESS SCHEMAs and the interface specification

The EXPRESS Schema that is the "topmost" schema in a network of schemas shall be mapped into a UML Model. The name of the Model shall be the name of the EXPRESS schema.

Bibliography.

[1] ISO 10303-21:1994, *Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure*.

[2] ISO 10303-22: 1999, *Industrial automation systems and integration – Product data representation and exchange – Part 22: Implementation method: Standard data access interface specification*.

[3] PDML documentation.

Index

abs element 9
abstract_supertype element 9
acos element 9
add element 9
aggregate_initializer element 9
aggregate_source element 9
aggregate_type element 10
algorithm_head element 10
alias_stmt element 10
and element 10
applies_to_entities element 10
array_literal element 44
array_type element 10
asin element 10
assignment_stmt element 11
atan element 11
attribute_id element 11
attribute_instance element 45
attribute_ref element 11
author element 45
authorisation element 45
bag_literal element 45
bag_type element 11
base_type element 11
binary element 12
binary_literal element 12
binary_literal element 45
blength element 12
boolean element 12
bound_spec element 12
bracketed_expression element 12
case_action element 12
case_label element 13
case_stmt element 13
complex_entity_constructor element 13
const_e element 14
constant_block element 13
constant_decl element 14
constant_id element 14
constant_import element 14
constant_instances element 45
constant_ref element 14
cos element 15

data element 46
data_section_description element 46
data_section_header element 46
data_section_identification_name element 46
data_section_name element 46
declaration_block element 15
derive_clause element 15
derived_attr element 15
description element 46
domain_rule element 15
element_item element 16
element_list element 16
embedded_remark element 16
Entity Data Type Reference 4
entity_constructor element 16
entity_decl element 16
entity_id element 17
entity_import element 17
entity_instance element 46
entity_ref element 17
enumeration element 17
enumeration_id element 17
enumeration_ref element 17
enumeration_ref element 47
enumeration_reference element 18
equal element 18
escape_stmt element 18
exists element 18
exp element 18
explicit_attr element 18
explicit_attr_block element 18
EXPRESS-driven data 3
factor element 18
false element 19
fixed element 19
flat_complex_entity_instance element 47
formal_parameter element 19
formal_parameter_block element 19
format element 19
function_call element 20
function_decl element 20
function_id element 20
function_import element 20
function_ref element 20
function_return_type element 21
general_array_type element 21
general_bag_type element 21
general_list_type element 21
general_set_type element 21

generic_type element 22
greater_than element 22
greater_than_or_equal element 22
hibound element 22
high_index element 22
hiindex element 22
if_stmt element 22
import_all element 22
in element 23
increment element 23
increment_control element 23
indeterminate element 23
index_qualifier element 23
index_spec element 23
insert element 23
instance_equal element 24
instance_not_equal element 24
integer element 24
integer_divide element 24
integer_literal element 24
integer_literal element 47
interface_specification_block element 24
interval element 24
interval_high_exclusive element 25
interval_high_inclusive element 25
interval_item element 25
interval_low_exclusive element 26
interval_low_inclusive element 26
inverse_attr element 26
inverse_bag element 26
inverse_clause element 26
inverse_set element 27
ISO-10303-data element 47
label element 27
length element 27
less_than element 27
less_than_or_equal element 27
like element 27
list_literal element 47
list_type element 27
lobound element 27
local_variable_block element 28
local_variable_decl element 28
log element 28
log10 element 28
log2 element 28
logical element 28
logical_expression element 29
logical_literal element 29

logical_literal element 47
loindex element 29
low_index element 29
lower_bound element 29
mod element 29
multiply element 30
negate element 30
nested_complex_entity_instance element 48
nested_complex_entity_instance_subitem element 48
non_constant_instances element 48
not element 30
not_equal element 30
null_stmt element 30
number element 30
numeric_expression element 30
nvl element 30
odd element 31
optional element 31
or element 31
organisation element 48
originating_system element 48
otherwise element 31
parameter_id element 31
parameter_ref element 31
partial_entity_instance element 49
pi element 31
plus element 31
population element 32
precision_spec element 32
preprocessor_version element 49
procedure_call_stmt element 32
procedure_decl element 32
procedure_formal_parameter_block element 32
procedure_id element 32
procedure_import element 33
procedure_ref element 33
qualified_attribute element 33
qualified_factor element 33
qualifier element 33
query element 33
raise_to_power element 33
real element 34
real_divide element 34
real_literal element 34
real_literal element 49
reference_from element 34
relation_expression element 34
remove element 35
repeat_control element 35

repeat_stmt element 35
repetition element 35
return_stmt element 36
rolesof element 36
rule_decl element 36
rule_id element 36
schema_decl element 36
schema_id element 36
schema_instance element 49
schema_ref element 36
select element 37
self element 37
set_literal element 49
set_type element 37
simple_entity_instance element 49
simple_expression element 37
sin element 37
sizeof element 37
skip_stmt element 38
sqrt element 38
statement_block element 38
string element 38
string_literal element 38
string_literal element 50
subtract element 38
subtype_of element 38
supertype_and element 39
supertype_and_or element 39
supertype_of element 39
supertype_one_of element 39
tail_remark element 39
tan element 39
term element 40
time_stamp element 50
true element 40
type element 50
type_decl element 40
type_id element 40
type_import element 40
type_label_id element 41
type_label_ref element 41
type_ref element 41
typeof element 40
unary_op element 41
underlying_type element 41
unique element 41
unique_clause element 42
unique_rule element 42
unknown element 42

until element 42
upper_bound element 42
use_from element 42
usedin element 42
value element 42
value_in element 43
value_unique element 43
var_formal_parameter element 43
variable_id element 43
variable_ref element 43
where_clause element 43
while element 44
width_spec element 44
xor element 44