ISO TC184/SC4/WG11 N052

Date: 1997-05-04 **Supersedes WG11 N002**

PRODUCT DATA REPRESENTATION AND EXCHANGE

Part: 1? **Title:** **EXPRESS-X Language Reference Manual**

Purpose of this document as it relates to the target document is:

x	Primary content	Current status:	Working Draft
	Issue discussion	Version	
	Alternate proposal		
	Partial content		

ABSTRACT:

Draft of EXPRESS-X language reference manual.

KEYWORDS

EXPRESS
EXPRESS-X
Mapping language

Document status/dates (dd/mm/yy)

Part Documents		Other SC4 Documents	
X	Working draft		Working
	Project draft		Released
	Released draft		Confirmed
	Technically complete		Released
	Editorially complete		
	ISO Committee Draft		

Project leader:	Martin Hardwick	**Editor:**	Ian Bailey
Address:	STEP Tools, Inc. 1223 Peoples Avenue Troy, NY 12180 USA	**Address:**	EuroSTEP Limited 60 Denecroft Crescent Hillingdon, UB10 9HX UK
Telephone:	+1 (518) 276-2848	**Telephone:**	+44 468 892362
Fax:	+1 (518) 276-8471	**Fax:**	
E-mail:	hardwick@steptools.com	**E-mail:**	ian.bailey@eurostep.co.uk

Comments to Reader:

This draft remains technically and editorially incomplete. It is being distributed to elicit feedback from interested parties and to allow early prototype implementation to begin. Potential implementers should be aware of the early and volatile nature of the language.

Contents

Page

Figures

Tables

Foreword

The International Organization for Standardization (ISO) 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 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 10303-1**?** was prepared by Technical Committee ISO/TC 184, *Industrial automation systems and integration*, Subcommittee SC4, *Industrial data*.

ISO 10303 consists of the following parts under the general title Industrial automation systems and integration - Product data representation and exchange:

— Part 1, Overview and fundamental principles;

— Part 11, Description methods: The EXPRESS language reference manual;

— Part 12, Description methods: The EXPRESS-I language reference manual;

— Part 21, Implementation methods: Clear text encoding of the exchange structure;

— Part 22, Implementation methods: Standard data access interface specification;

— Part 23, Implementation methods: C++ language binding to the standard data access interface;

— Part 24, Implementation methods: C language binding to the standard data access interface;

— Part 26, Implementation methods: Interface definition language binding to the standard data access;

— Part 31, Conformance testing methodology and framework: General concepts;

— Part 32, Conformance testing methodology and framework: Requirements on testing laboratories

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

Printed in Switzerland

and clients;

— Part 33, Conformance testing methodology and framework: Structure and use of abstract test suites;

— Part 34, Conformance testing methodology and framework: Abstract test methods;

— Part 35, Conformance testing methodology and framework: Abstract test methods for SDAI implementations;

— Part 41, Integrated generic resource: Fundamentals of product description and support;

— Part 42, Integrated generic resources: Geometric and topological representation;

— Part 43, Integrated generic resources: Representation structures;

— Part 44, Integrated generic resources: Product structure configuration;

— Part 45, Integrated generic resources: Materials;

— Part 46, Integrated generic resources: Visual presentation;

— Part 47, Integrated generic resources: Shape variation tolerances;

— Part 49, Integrated generic resources: Process structure and properties;

— Part 101, Integrated application resources: Draughting;

— Part 104, Integrated application resources: Finite element analysis;

— Part 105, Integrated application resources: Kinematics;

— Part 106, Integrated application resources: Building construction core model;

— Part 201, Application protocol: Explicit draughting;

— Part 202, Application protocol: Associative draughting;

— Part 203, Application protocol: Configuration controlled design;

— Part 204, Application protocol: Mechanical design using boundary representation;

— Part 205, Application protocol: Mechanical design using surface representation;

— Part 207, Application protocol: Sheet metal die planning and design;

— Part 208, Application protocol: Life cycle management - Change process;

— Part 209, Application protocol: Composite and metallic structural analysis and related design;

— Part 210, Application protocol: Electronic assembly, interconnect, and packaging design;

— Part 212, Application protocol: Electrotechnical design and installation

— Part 213, Application protocol: Numerical control process plans for machined parts;

— Part 214, Application protocol: Core data for automotive design;

— Part 215, Application protocol: Ship arrangement;

— Part 216, Application protocol: Ship moulded forms;

— Part 217, Application protocol: Ship piping;

— Part 218, Application protocol: Ship structures;

— Part 220, Application protocol: Process planning, manufacture, and assembly of layered electronic products;

— Part 221, Application protocol: Functional data and their schematic representation for process plant;

— Part 222, Application protocol: Exchange of product data for composite structures;

— Part 223, Application protocol: Exchange of design and manufacturing product information for casting parts;

— Part 224, Application protocol: Mechanical product definition for process plans using machining features;

— Part 225, Application protocol: Building elements using explicit shape representation;

— Part 226, Application protocol: Ship mechanical systems;

— Part 227, Application protocol: Plant spatial configuration;

— Part 228, Application protocol: Building services: Heating, ventilation, and air conditioning;

— Part 229, Application protocol: Exchange of design and manufacturing product information for forged parts;

— Part 230, Application protocol: Building structural frame: Steelwork;

— Part 231, Application protocol: Process engineering data: Process design and process specification of major equipment;

— Part 232, Application protocol: Technical data packaging core information and exchange;

— Part 301, Abstract test suite: Explicit draughting;

— Part 302, Abstract test suite: Associative draughting;

— Part 303, Abstract test suite: Configuration controlled design;

— Part 304, Abstract test suite: Mechanical design using boundary representation;

— Part 305, Abstract test suite: Mechanical design using surface representation;

— Part 307, Abstract test suite: Sheet metal die planning and design;

— Part 308, Abstract test suite: Life cycle management - Change process;

— Part 309, Abstract test suite: Composite and metallic structural analysis and related design;

— Part 310, Abstract test suite: Electronic assembly, interconnect, and packaging design;

— Part 312, Abstract test suite: Electrotechnical design and installation;

— Part 313, Abstract test suite: Numerical control process plans for machined parts;

— Part 314, Abstract test suite: Core data for automotive mechanical design;

— Part 315, Abstract test suite: Ship arrangement;

— Part 316, Abstract test suite: Ship moulded forms;

— Part 317, Abstract test suite: Ship piping;

— Part 318, Abstract test suite: Ship structures;

— Part 320, Abstract test suite: Process planning, manufacture, and assembly of layered electronic products;

— Part 321, Abstract test suite: Functional data and their schematic representation for process plant;

— Part 322, Abstract test suite: Exchange of product data for composite structures;

— Part 323, Abstract test suite: Exchange of design and manufacturing product information for casting parts;

— Part 324, Abstract test suite: Mechanical product definition for process plans using machining features;

— Part 325, Abstract test suite: Building elements using explicit shape representation;

— Part 326, Abstract test suite: Ship mechanical systems;

— Part 327, Abstract test suite: Plant spatial configuration;

— Part 328, Abstract test suite: Building services: Heating, ventilation, and air conditioning;

— Part 329, Abstract test suite: Exchange of design and manufacturing product information for forged parts;

— Part 330, Abstract test suite: Building structural frame: Steelwork;

— Part 331, Abstract test suite: Process engineering data: Process design and process specification of major equipment;

— Part 332, Abstract test suite: Technical data packaging core information and exchange;

— Part 501, Application interpreted construct: Edge-based wireframe;

— Part 502, Application interpreted construct: Shell-based wireframe;

— Part 503, Application interpreted construct: Geometrically bounded 2D wireframe;

— Part 504, Application interpreted construct: Draughting annotation;

— Part 505, Application interpreted construct: Drawing structure and administration;

— Part 506, Application interpreted construct: Draughting elements;

— Part 507, Application interpreted construct: Geometrically bounded surface;

— Part 508, Application interpreted construct: Non-manifold surface;

— Part 509, Application interpreted construct: Manifold surface;

— Part 510, Application interpreted construct: Geometrically bounded wireframe;

— Part 511, Application interpreted construct: Topologically bounded surface;

— Part 512, Application interpreted construct: Faceted boundary representation;

— Part 513, Application interpreted construct: Elementary boundary representation;

— Part 514, Application interpreted construct: Advanced boundary representation;

— Part 515, Application interpreted construct: Constructive solid geometry;

— Part 517, Application interpreted construct: Mechanical design geometric presentation;

— Part 518, Application interpreted construct: Mechanical design shaded representation;

The structure of this International Standard is described in ISO 10303-1. The numbering of the parts of the International Standard reflects its structure:

— Parts 11 to 13 specify the description methods,

— Parts 21 to 26 specify the implementation methods,

— Parts 31 to 35 specify the conformance testing methodology and framework,

— Parts 41 to 49 specify the integrated generic resources,

— Parts 101 to 106 specify the integrated application resources,

— Parts 201 to 232 specify the application protocols,

— Parts 301 to 332 specify the abstract test suites,

— Parts 501 to 518 specify the application interpreted constructs, and

Should further parts of ISO 10303 be published, they will follow the same numbering pattern.

Annex A forms an integral part of this part of ISO 10303. Annex B is for information only.

Introduction

ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. The objective is to provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system. The nature of this description makes it suitable not only for neutral file exchange, but also as a basis for implementing and sharing product databases and archiving.

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, application modules, abstract test suites, implementation methods, and conformance testing. The series are described in ISO 10303-1. This part of ISO 10303 is a member of the description methods series.

This part of ISO 10303 specifies the EXPRESS-X mapping language.

INTERNATIONAL STANDARD © ISO ISO/NWI 10303-1?

Industrial automation systems and integration — Product data representation and exchange — Part 1?: Description methods: The EXPRESS-X language reference manual

1 Scope

This part of ISO 10303 defines a language by which the relationship of data defined by implicit or explicit models in the EXPRESS language can specified. The language is called EXPRESS-X.

EXPRESS-X is a structural data mapping language. It consists of language elements that allow an unambiguous specification of the relationship between models.

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

— Mapping data defined by one EXPRESS model to data defined by another EXPRESS model.

— Mapping data defined by one version of an EXPRESS model to data defined by another version of that EXPRESS model, where the two schemas have different names.

— Specification of requirements for data translators for data sharing and data exchange applications.

— Formal specification of high-level or alternate views of data defined by an EXPRESS model.

— Providing an alternate notation for application protocol mapping tables.

— Allowing for bi-directional mappings where mathematically possible.

— Concatenation of mappings sharing a common model.

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

— Mapping of data defined using means other than EXPRESS.

— Mapping of data defined using the second edition of EXPRESS.

— Identification of the version of an EXPRESS schema.

— Graphical representation of constructs in the EXPRESS-X language.

2 Normative references

The following standards contain provisions that, through reference in this text, constitute provisions of this part of ISO 10303. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO 10303 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

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*.

3 Definitions

3.1 Terms defined in ISO 10303-1

This part of ISO 10303 makes use of the following terms defined in ISO 10303-1.

— data;

— data specification language;

— exchange structure;

— implementation method;

— information;

— information model.

3.2 Terms defined in ISO 10303-11

This part of ISO 10303 makes use of the following terms defined in ISO 10303-11.

— complex entity data type;

— complex entity (data type) instance;

— constant;

— entity;

— entity data type;

— entity (data type) instance;

— instance;

— partial complex entity data type;

— partial complex entity value;

— population;

— simple entity (data type) instance;

— subtype/supertype graph;

— token;

— value.

3.3 Other definitions

For the purposes of this part of ISO 10303, the following definitions apply:

3.3.1 map: a declaration that defines the algorithm for translating one (or more) source entity types to one (or more) target entity types.

3.3.2 view: an alternative organization of the information in an EXPRESS model.

3.3.3 extent: the set of valid bindings that form the basis for a view.

3.3.4 binding: an ordered tuple of values against which view attribute expressions may be evaluated.

3.3.5 view attribute: a named expression defined in a view declaration that may be evaluated against a binding to produce a value.

4 Conformance requirements

4.1 Formal specifications written in EXPRESS-X

4.1.1 Lexical language

A formal specification written in EXPRESS-X shall be consistent with a given level as specified below. A formal specification is consistent with a given level when all checks identified for that level as well as all lower levels are verified for the specification.

Levels of checking

Level 1: Reference checking. This level consists of checking the formal specification to ensure that it is syntactically and referentially valid. A formal specification is syntactically valid if it matches the syntax generated by expanding the primary syntax rule (`syntax`) given in Annex A. A formal specification is referentially valid if all references to EXPRESS-X items are consistent with the scope and visibility rules defined in clauses 10 and 11.

Level 2: Type checking. This level consists of checking the formal specification to ensure that it is consistent with the following:

- expressions shall comply with the rules specified in clause 12 and in ISO 10303-11:1994 clause 12;

- assignments shall comply with the rules specified in ISO 10303-11:1994 clause 13.4.

Level 3: Value checking. This level consists of checking the formal specification to ensure that it is consistent with statements of the form, ‘A shall be greater than B’, as specified in clause 7 to 14. This is limited to those places where both A and B can be evaluated from literals and/or constants.

Level 4: Complete checking. This level consists of checking the formal specification to ensure that it is consistent with all stated requirements as specified in this part of ISO 10303.

4.2 Implementations of EXPRESS-X

4.2.1 EXPRESS-X language parser

An implementation of an EXPRESS-X language parser shall be able to parse any formal specification written in EXPRESS-X, consistent with the constraints as specified in annex **?(PICS)** and the conformance class associated with that implementation. An EXPRESS-X language parser shall be said to conform to a particular checking level (as defined in 4.1.1) if it can apply all checks required by that level (and any level below it) to a formal specification written in EXPRESS-X.

The implementer of an EXPRESS-X language parser shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented in the form specified in annex **?(PICS)** for the purpose of conformance testing.

4.2.2 EXPRESS-X mapping engine

An implementation of an EXPRESS-X mapping engine shall be able to evaluate and/or execute any formal specification written in EXPRESS-X, consistent with the constraints as specified in annex PICS and the conformance class associated with that implementation. The execution and/or evaluation of a mapping is relative to one or more source data sets; the specification of how these data sets are made available to the mapping engine is outside the scope of this part of ISO 10303.

The implementer of an EXPRESS-X mapping engine shall state any constraints that the implementation imposes on the number and length of identifiers, on the range of processed numbers, and on the maximum precision of real numbers. Such constraints shall be documented in the form specified in annex PICS for the purpose of conformance testing.

An implementation of an EXPRESS-X mapping engine may take many forms; the following list is not exhaustive, but represents the more common examples of mapping engine implementation. An EXPRESS-X mapping engine shall be said to conform to a particular type of mapping paradigm (as described below) if it meets all of the criteria for that type.

4.2.2.1 Push mapping

An implementation shall be said to be a push mapping implementation if it meets all of the following criteria:

- The mapping engine accepts one or more source data sets, and produces one or more output data sets.
- The output data sets are derived from the input data sets by the execution and evaluation of all of the mapping declarations from the mapping schema.

– Every instance in the source data sets is mapped as specified in the mapping schema into the output data sets.

4.2.2.2 Pull mapping

An implementation shall be said to be a pull mapping implementation if it meets all of the following criteria:

– The mapping engine accepts one or more source data sets.

– Specified target data instances are derived from the input data sets by the execution and evaluation of the appropriate declarations from the mapping schema.

– Updates to target data instances are propagated back to the source data sets by the inverse execution and evaluation of the appropriate declarations from the mapping schema.

4.2.2.3 View mapping

An implementation shall be said to be a view mapping implementation if it meets all of the following criteria:

– The mapping engine accepts a single data set described by an EXPRESS information model.

– The source data instances may be accessed through the evaluation of view declarations from the mapping schema.

– Updates may be made to viewed source data instances, and these updates are propagated back to the source data set by the inverse evaluation of the appropriate declarations from the mapping schema.

4.3 Conformance classes

An implementation shall be said to conform to conformance class 1 if it processes all the declarations that may appear in a schema view declaration.

An implementation shall be said to conform to conformance class 2 if it processes all the declarations that may appear in a schema map declaration that does not contain any external references to a schema view declaration.

An implementation shall be said to conform to conformance class 3 if it processes any EXPRESS-X specification that conforms to this part of ISO 10303.

5 Fundamental principles

The reader of this document is assumed to be familiar with the following concepts, in addition to the concepts described in clause 5 of ISO 10303-11:1994.

EXPRESS-X provides for the specification of:

- alternative views of the data described by an information model described in EXPRESS;
- the transformation of data described by elements of one EXPRESS model into data described by elements of another EXPRESS model.

Many of the elements of the EXPRESS-X language are assigned names. The names allow other language elements to reference the associated representation. The use of the name in the definition of other language elements constitutes a reference to the underlying representation. While the syntax of the language uses an identifier for the name, the underlying representation must be examined to understand the meaning.

A schema view written in the EXPRESS-X language describes a set of alternate views of the data described by an EXPRESS model. Each view is based upon an ordered tuple of values, called a binding. The definition of a view describes the set of valid bindings (called the extent) and one or more view attributes, expressions that can be evaluated relative to a given binding. The values constituting a binding may be instance references, values of the primitive EXPRESS types, or other bindings.

A schema map written in the EXPRESS-X language describes how elements of one EXPRESS model (the source model) may be transformed into elements of another (the target model). A schema map is composed principally of map and type map declarations. A schema map may reference definitions from an EXPRESS-X schema view.

The specification of a map is based upon an extent of bindings. For each valid binding in the extent, the body of the map is executed in order to create and populate one or more instances in the target model. A map specification that meets certain criteria is said to be reversible; reversible maps allow a change to data defined by the target model to be propagated back to the source model.

The specification of a type map defines how data described by EXPRESS defined types may be transformed between the source and target model.

EXPRESS function and procedure specifications may form part of an EXPRESS-X specification in order to support the definition of views, maps, or type maps.

The EXPRESS-X language does not describe an implementation environment. In particular, EXPRESS-X does not specify:

- how references to names are resolved;
- how other schemas, schema views, or schema maps are known;
- how input and output data sets are specified;

– how mappings are executed for instances that do not conform to an EXPRESS schema.

6 Language specification syntax

The notation used to present the syntax of the EXPRESS-X language is defined in this clause.

The full syntax for the EXPRESS-X language is given in Annex A. Portions of those syntax rules are reproduced in various clauses to illustrate the syntax of a particular statement. Those portions are not always complete. It will sometimes be necessary to consult Annex A for the missing rules. The syntax portions within this part of ISO 10303 are presented in a box. Each rule within the syntax box has a unique number toward the left margin for use in cross-references to other syntax rules.

The syntax of EXPRESS-X is defined in a derivative of Wirth Syntax Notation (WSN).

NOTE — See annex B for a reference.

The notational conventions and WSN defined in itself are given below.

```
syntax         = { production } .
production     = identifier '=' expression '.' .
expression     = term { '|' term } .
term           = factor { factor } .
factor         = identifier | literal | group | option | repetition .
identifier     = character { character } .
literal        = '''' character { character } '''' .
group          = '(' expression ')' .
option         = '[' expression ']' .
repetition     = '{' expression '}' .
```

– The equal sign `'='` indicates a production. The element on the left is defined to be the combination of the elements on the right. Any spaces appearing between the elements of a production are meaningless unless they appear within a literal. A production is terminated by a period `'.'`.

– The use of an identifier within a factor denotes a nonterminal symbol that appears on the left side of another production. An identifier is composed of letters, digits, and the underscore character. The keywords of the language are represented by productions whose identifier is given in uppercase characters only.

– The word literal is used to denote a terminal symbol that cannot be expanded further. A literal is a sequence of characters enclosed in apostrophes. For an apostrophe to appear in a literal it must be written twice, i.e., `''''`.

– The semantics of the enclosing braces are defined below:

- curly brackets `'{ }'` indicates zero or more repetitions;

- square brackets `'[ ]'` indicates optional parameters;
- parenthesis `'( )'` indicates that the group of productions enclosed by parenthesis shall be used as a single production;
- vertical bar `'|'` indicates that exactly one of the terms in the expression shall be chosen.

The following notation is used to represent entire character sets and certain special characters which are difficult to display:

– **`\a`** represents any character from ISO/IEC 10646-1;

– **`\n`** represents a newline (system dependent) (see clause 7.1.5.2 of ISO 10303-11:1994).

7 Basic language elements

This clause specifies the basic elements from which an EXPRESS-X mapping specification is composed: the character set, remarks, symbols, reserved words, identifiers, and literals.

The basic language elements are the same as those for the EXPRESS language defined in Clause 7 of ISO 10303-11, with the exceptions noted below.

7.1 Reserved words

The reserved words of EXPRESS-X are the keywords and the names of built-in constants, functions, and procedures. Any reserved word in EXPRESS (ISO 10303-11:1994) shall also be a reserved word in EXPRESS-X. The reserved words shall not be used as identifiers. The reserved words of EXPRESS-X are described below.

In the case that a legal EXPRESS identifier is a reserved word in EXPRESS-X, schemas using that identifier can be mapped by renaming the conflicting identifier using the AS keyword in the REFERENCE clause.

7.1.1 Keywords

In addition to the keywords of EXPRESS defined in ISO 10303-11:1994, the following are keywords of EXPRESS-X.

Table 1 — Additional EXPRESS-X keywords

ALIAS	EACH	END_INLINE_FUNCTION	END_MAP
END_SCHEMA_MAP	END_SCHEMA_VIEW	END_VIEW	IDENTIFIED_BY
IMPORT_FUNCTIONS	IMPORT_MAPPING	INLINE_FUNCTION	MAP
OID	PARTITION	SCHEMA_MAP	SCHEMA_VIEW
SOURCE	TARGET	TYPE_MAP	VIEW

8 Data types

This clause defines the data types provided as part of the language. Every view attribute, map attribute, or type map has an associated data type.

The data types are the same as those for the EXPRESS language defined in clause 8 of ISO 10303-11:1994, with the exceptions noted below.

8.1 Complex entity data type

Complex entity data types are established implicitly by entity declarations related in an inheritance hierarchy (see ISO 10303-11:1994, clause 9.2). An entity data type is assigned an entity identifier by the user. An entity data type is referenced by this identifier. A complex entity data type is referenced by an expression that lists the partial complex entity data types that are combined to form it, separated by the keyword AND.

The partial complex entity data types may be listed in any order.

Any partial complex entity data types that are included in another partial complex entity data type via inheritance are not listed.

Syntax:

```
46   complex_entity_ref = entity_ref { AND entity_ref } .
```

Rules and restrictions:

a) Each entity_ref shall be a reference to an entity which is visible in the current scope.

b) The referenced complex entity data type shall describe a valid domain within some schema (see ISO 10303-11:1994, annex B).

c) A given entity_ref shall occur at most once within a complex_entity_ref.

EXAMPLE 1 – Given the following entity declarations:

```
ENTITY super SUPERTYPE OF ONEOF(a,c);
END_ENTITY;

ENTITY a SUBTYPE OF (super);
END_ENTITY;

ENTITY b SUBTYPE OF (super);
END_ENTITY;

ENTITY c SUBTYPE OF (super);
END_ENTITY;
```

The following are valid complex entity data types:

```
a AND b
```

```
b AND c
```

The following are not valid complex entity data types:

```
a AND b AND super
```

```
a AND c
```

```
b AND c AND c
```

8.2 Binding data type

Binding data types are established by `VIEW` declarations (see clause 9.3). A binding data type is assigned an identifier in the defining schema map or schema view. A binding data type is referenced by this identifier.

A value of a binding data type may be produced in the following ways:

- taking a member of an extent;
- evaluating an explicit binding expression;
- as the value of a binding parameter in the definition of one extent over another.

Syntax:

```
24    binding_ref = binding_id .
```

Rules and restrictions:

a) binding_ref shall be a reference to a binding which is visible in the current scope.

8.3 Extent data type

Extent data types are established by `VIEW` declarations (see clause 9.3). An extent data type is assigned an extent identifier in the defining schema map or schema view. An extent data type is referenced by this identifier. This identifier is the same as that used for the binding data type.

The extent is the set of all valid bindings.

Syntax:

```
26   extent_ref = extent_id .
```

Rules and restrictions:

b) extent_ref shall be a reference to an extent which is visible in the current scope.

EXAMPLE 2 – The following declaration defines a new extent data type; each member of the extent is a binding of type circle:

```
VIEW circle;
FROM (e : ellipse);
BEGIN_VIEW
  ...
END_VIEW;
```

9 Declarations

This clause defines the various declarations available in EXPRESS-X. An EXPRESS-X declaration creates a new EXPRESS or EXPRESS-X item and associates an identifier with it. The item may be referenced elsewhere by writing the name associated with it.

The principle capabilities of EXPRESS-X are found in the following declarations:

– View;

– Map;

– Schema_view;

– Schema_map;

– Type_map.

In addition, an EXPRESS-X specification may contain the following declarations defined in ISO 10303-11:1994:

– Constant;

– Function;

– Procedure.

Mapping declarations are always explicit.

9.1 Schema_view declaration

A schema_view declaration defines a common scope for a collection of related mapping declarations. A schema_view may contain the following kinds of declarations:

– constant declaration (clause 9.5);

– function declaration (clause 9.6);

– procedure declaration (clause 9.7);

– view declaration (clause 9.3).

The order in which declarations appear within a schema_view declaration is not significant.

Declarations in one schema_view or EXPRESS schema may be made visible within the scope of another schema_view via an interface specification as described in clause 11.

Syntax:

```
110   schema_view_decl = SCHEMA_VIEW schema_view_id schema_view_body
                    END_SCHEMA_VIEW ';' .
109   schema_view_body = { reference_clause } [ constant_decl ]
                    { function_decl | procedure_decl | view_decl } .
```

EXAMPLE 3 – Ap203_arm may be the name of a schema_view that contains declarations defining a view over the schema config_control_design in terms of the domain expert's understanding of the information requirements.

```
SCHEMA_VIEW ap203_arm;

REFERENCE FROM config_control_design;

VIEW part_version ...

(* other mapping declarations as appropriate *)

END_SCHEMA_VIEW;
```

9.2 Schema_map declaration

A schema_map declaration defines a common scope for a collection of related mapping declarations.

EXAMPLE 4 – IGES2STEP may be the name of a schema_map that contains declarations for translating geometry defined using and EXPRESS model base upon IGES into STEP AP203.

The order in which declarations appear within a schema_map declaration is not significant. In particular, the order of the declarations has no effect upon the resulting mapping.

Declarations in one schema_map may be made visible within the scope of another schema_map via an interface specification as described in clause 11.

A schema_map may contain the following kinds of declarations:

- constant declaration (clause 9.5);
- function declaration (clause 9.6);
- procedure declaration (clause 9.7);
- type_map declaration (clause 0);
- map declaration (clause 9.4).

Syntax:

```
107   schema_map_decl = SCHEMA_MAP schema_map_id ';' schema_map_body
                        END_SCHEMA_MAP ';' .
106   schema_map_body = view_or_map_decl { view_or_map_decl } .
77    interface_specification = source_interface_spec target_interface_spec
                                { map_interface_spec }
                                { external_functions_spec } .
```

An EXPRESS-X specification consists of one or more mapping schemas, each of which defines the required view materialization process for a view of an EXPRESS model.

A mapping schema specification in EXPRESS-X is similar to a schema specification in EXPRESS.

The specification of the body of a mapping schema in EXPRESS-X has the same form as the specification of the body of a schema in EXPRESS, with two exceptions. When defining a mapping schema, it is necessary to create declarations that define the mappings. As a result, EXPRESS-X has an expanded set of allowable declarations for use in defining the mapping schema. It is also necessary in EXPRESS-X to include a global section that identifies the base schema and the view schema for the mappings.

EXAMPLE 5 – Example.

```
SCHEMA_MAP map_name;
  TARGET t1, t2, t2;
  SOURCE s1, s2;
IMPORT MAPPING other_map;
```

9.3 View declaration

A view declaration creates a view extent and declares an identifier to refer to it, and defines one or more view expressions that may be evaluated for a given binding in the extent. A view declaration consists of one or more partitions.

Each partition defines part of the entire view extent. Partitions may be named; the name is optional if there is only a single partition.

Each view expression defines an expression in terms of a member of the view extent. A view expression may be evaluated to produce a reference to data in the underlying model, a binding, or a full or partial extent.

EXAMPLE 6 – The following view collects the information about persons serving in roles within organizations. This information is collected from two instances of person_and_organization and cc_design_person_and_organization_assignment. The two instances must be related via the assigned_person_and_organization attribute of the cc_design_person_and_organization_assignment. Three attribute reference expressions are given.

```
VIEW arm_person_role_in_organization;
FROM (pao : person_and_organization,
      ccdpaoa : cc_design_person_and_organization_assignment)
WHERE ccdpaoa.assigned_person_and_organization :=: pao;
SELECT
  person : person := pao.the_person;
  org : organization := pao.the_organizaion;
  role : label := ccdpaoa.role.name;
END_VIEW;
```

Syntax:

```
137   view_decl = view_head view_partition { view_partition } END_VIEW ';' .
141   view_head = VIEW view_id ';' .
```

Rules and restrictions:

a) Each attribute expression reference declared in the view declaration shall be unique within the declaration.

b) If a view is composed of more than a single partition, each partition shall be named.

9.3.1 View partitions

A single partition defines a portion of a view extent. All the partitions together define the entire extent for a given view. A partition is composed of a name, one or more binding parameters, a set of binding constraints, and one or more view attributes. The name is optional if there is only a single partition.

Each binding parameter has a type; this type will be either an entity data type, a view extent data type, or an aggregate of type SET with a base type that is either an entity or a view extent data type. The binding parameters are specified explicitly or are inherited from another view.

Each binding parameter is potentially bound to each member of its type extent (or to multiple members if the `IDENTIFIED_BY` clause is used), determined by the set of constraints. The set of all bindings forming the view extent is the subset of the Cartesian product of the view parameters that satisfy all the view constraints.

A view constraint is deemed to be satisfied if it does not evaluate to FALSE (i.e., if it evaluations to either TRUE or UNKNOWN).

Each binding parameter of a partition represents an entity instance or view binding from which the view attributes may be derived. The data type of the parameter establishes the domain over which the bindings will range.

Each constraint is a logical expression that shall be valid for any binding in the portion of the extent defined by the partition.

EXAMPLE 7 – In AP 201, the application object Organization can be mapped to either a person, an organization, or both a person and organization entity in the AIM. This is specified in EXPRESS-X as follows:

```
VIEW arm_organization
PARTITION a_single_person :
  FROM (p : person)
    ...
PARTITION a_single_organization :
  FROM (o : organization)
    ...
PARTITION a_person_in_an_organization :
  FROM (po : person_and_organization)
    ...
END_VIEW;
```

Syntax:

```
151   view_partition = [ PARTITION partition_id ':' ] ( view_from_clause |
                    subtype_of_clause ) view_body .
139   view_from_clause = FROM '(' view_parameter { ';' view_parameter } ')'
                      [ where_clause ] [ identified_by_clause ]';' .
116   subtype_of_clause = SUBTYPE OF view_extent_ref
                        [ PARTITION partition_ref ] [ where_clause ] ';' .
65    identified_by_clause = IDENTIFIED_BY expression { ',' expression }
                             ';' .
149   view_parameter = parameter_id { ',' parameter_id } ':'
                      view_parameter_type .
150   view_parameter_type = entity_ref | view_extent_ref | SET [ bound_spec ]
                           OF ( entity_ref | view_extent_ref ) .
```

Rules and restrictions:

a) The partitions shall be named if there is more than one of them.

9.3.2 View attributes

View attributes are expressions that may be evaluated for a given binding in the view extent. Each view expression shall be named and shall have a type. The result of evaluating the expression shall be assignment compatible with the specified type.

EXAMPLE 8 – Circle may be the name of an extent defined to contain all ellipse instances with equal length major and minor axes. For a given binding in this extent, we may evaluate two expressions to obtain the radius and the position of the circle.

```
VIEW circle;
FROM (e : ellipse) WHERE e.semi_axis_1 = e.semi_axis_2;
SELECT
  radius : positive_length_measure := e.semi_axis_1;
  position : axis2_placement := e.position;
END_VIEW;
```

Rules and restrictions:

9.3.3 View inheritance

A view may inherit its binding parameters from another view (specifically, from one partition of another view). This shall be specified with the `SUBTYPE OF` clause. The extent for the view is that subset of the extent of the “supertype” view that satisfies any additional constraints for the new view.

EXAMPLE 9 – The following view is defined using inheritance.

```
VIEW person;
FROM ...
END_VIEW;

VIEW male SUBTYPE OF person;
WHERE gender = ’M’;
...
END_VIEW;
```

9.3.4 IDENTIFIED_BY

The `IDENTIFIED_BY` clause is used to group multiple parameter bindings together logically based on a shared attribute value. This clause also changes how explicit bindings are called (see clause 0).

All the bindings in an extent that have the same value for the `IDENTIFIED_BY` expression are logically grouped together. If there is more than one such instance, the data type of the binding parameter must be an aggregate data type.

EXAMPLE 10 – For the following extent, all the departments with the same company are grouped together into the same binding.

```
VIEW company;
FROM (d : SET[1:?] OF department)
IDENTIFIED_BY d.company_name;
 ...
```

9.4 Map declaration

The MAP declaration supports the specification of correspondences between semantically equivalent elements of two or more pre-defined EXPRESS models that can be structured in a different way. The mapping between the complete models is structured in several MAP declarations. Each MAP declaration specifies how base instances of one or more types are to be mapped to target instances. That is, a MAP declaration supports the mapping from many target entities to many source entities in a single declaration.

A map declaration consists of a header and body of one or more map statements. The purpose of the header is to define the conditions under which one or many new target instance(s) should be created from one or more instances in a base model. The main purpose of the body is to define how the values of the attributes for a newly created instance are to be computed. The concept of alternative views, i.e. partitions, is also available for MAPs.

Syntax:

```
85   map_decl = MAP map_decl_header   [ map_partition_decl ] map_decl_body
                                     { [ map_partition_decl ] map_decl_body }
                END_MAP ';' .
```

9.4.1 Header of the MAP declaration

The header lists one or more target entities that are built from elements of the source EXPRESS models. It is not required that those target entities are related to each other by relationships.

Syntax:

```
89    map_partition_decl = PARTITION
                           partition_id ':' [ partition_entities ] .
87    map_decl_header = target_entity_ref_list [ AS group_name ] .
118   target_entity_ref_list = target_entity_ref_list_el
                               { ',' target_entity_ref_list_el } .
119   target_entity_ref_list_el = [ target_entity_ref_alias_id ':'
                                   [ ALIAS '[' bound_1 ':' bound_2 ']' OF ] ]
                                 target_entity_ref .
```

A target entity shall not be mapped in more than one MAP declaration in which the headers of those declarations consist only of a single target entity. In that case, partitions would have to be used. However, one target entity can be mapped in more than one MAP declarations (say n), if n-1 MAP declarations are group mappings. To support the call of a target entity mapping inside a group mapping, the MAP declaration of the group mapping is given a name (`group_name`).

EXAMPLE 11 – In the example below, a pump in the source data model is mapped to a set of target entities.

Example to be completed.

```
MAP xx AS group_for_pump
FROM p : pump
  -- attribute mappings of the target entities
END_MAP;
```

The initial values of the attributes of the newly created instance(s) are indeterminate.

9.4.2 Partitions within a MAP declaration

The partition concept is the same as described within the VIEW declaration (see clause 9.3.1). It is extended so that partitions can be defined for a list of target entities.

Syntax:

```
89    map_partition_decl = PARTITION
                         partition_id ':' [ partition_entities ] .
94    partition_entities = target_entity_ref_list .
```

If multiple target entities are listed in the header of the MAP declaration, different subset of those entities can be used for the partitions.

9.4.3 Body of the MAP declaration

The body of the MAP declaration consists of a FROM clause, an optional WHERE clause, and an optional IDENFIFIED_BY clause. The semantics of those clauses are defined below.

The project clause selects one source entity that corresponds to the target entity. Thus, the prerequisite of the project clause is that only one target entity is specified in the header, and so no group mapping has to be applied. The project clause would be the default mapping for one target entity to one source entity where all attributes have the same name, the same type and the same value. Alternatively, attribute assignment statements can be specified in the `map_attribute_decl_block`. The attribute declaration block also may contain the instantiation loop control (see syntax below).

Syntax:

```
86    map_decl_body = from_clause [ where_clause ] [ identified_by_clause ]
                      [ project_clause ] [ proj_clause_or_map_attr_decl ]
                      END_VIEW ';' .
97    proj_clause_or_map_attr_decl = project_clause
                                     | map_attribute_decl_block .
81    map_attribute_decl_block = entity_instantiation_loop
                                 | map_attr_decl_stmt_list .
80    map_attr_decl_stmt_list = map_attribute_declaration
                              { map_attribute_declaration } .
82    map_attribute_declaration = [ target_entity_ref '.' ]
                                  target_attribute_ref ':='
                                  map_attr_assgnmt_expr ';' .
```

9.5 Constant declaration

Constants may be defined for use within the WHERE clause of a view or map declaration, or within the body of a map declaration or algorithm.

Constant declarations are as defined in ISO 10303-11:1994 clause 9.4.

9.6 Function declaration

Functions may be defined for use within the WHERE clause of a view or map declaration, or within the body of a map declaration.

Function declarations are as defined in ISO 10303-11:1994 clause 9.5.1.

9.7 Procedure declaration

Procedures may be defined for use within the body of a map declaration.

Procedure declarations are as defined in ISO 10303-11:1994 clause 9.5.2.

9.8 Type map declaration

A type map declaration specifies how a value of a defined type is mapped to a value of another type within the scope of a schema map.

Syntax:

```
126   type_mapping_stmt = TYPE_MAP target_type_ref { target_type_ref } FROM
                          source_type_ref { source_type_ref } ';'
                          { type_map_stmt_body } END_TYPE_MAP ';' .
124   type_map_stmt_body = [ schema_ref '.' ] base_type ':='
                           type_expression ';' .
```

The following specifies the mapping between the types `dollar` and `dmark`.

```
TYPE_MAP dmark FROM dollar;
  dmark := 1.5 * dollar;
  dollar := dmark / 1.5;
END_TYPE_MAP;
```

The mapping is applied whenever the source attribute type is type compatible with one of the first types and the target attribute type is type compatible with one of the second types.

Rules and restrictions:

a) Body is not needed if just renaming.

b) No more than two expressions; if second is omitted then reverse mapping is implicit.

c) The two expressions shall be inverses of each other.

d) No entity instances shall be mapped by the TYPE_MAP. The base type shall not be an entity type.

10 Scope and visibility

An EXPRESS-X declaration creates an identifier that can be used to reference the declared item in other parts of the schema_view (or in other schema_views). Some EXPRESS-X constructs implicitly declare items, attaching identifiers to them. In those areas where an identifier for a declared item may be referenced, the declared item is said to be visible. An item may only be referenced where its identifier is visible. For the rules of visibility, see clause 10.2. For further information on referring to items using their identifiers, see clause 12.

Certain EXPRESS-X items define a region (block) of text called the scope of the item. This scope limits the visibility of identifiers declared within it. Scope can be nested; that is, an EXPRESS-X item which establishes a scope may be included within the scope of another item. There are constraints on which items may appear within a particular EXPRESS-X item's scope. The constraints are usually enforced by the syntax of EXPRESS-X.

For each of the items specified in table 2 below the following subclauses specify the limits of the scope defined, if any, and the visibility of the declared identifier both in general terms and with specific details.

Table 2 — Scope and identifier defining items

Item	Scope	Identifier
view attribute		•
view	•	•
partition	•	•
schema_view	•	•
To be completed.		

10.1 Scope rules

The general scope rules are as defined in ISO 10303-11:1994.

10.2 Visibility rules

The general visibility rules are as defined in ISO 10303-11:1994.

10.3 Explicit item rules

The following clauses provide more detail on how the general scoping and visibility rules apply to the various EXPRESS-X items.

10.3.1 Schema_view

Visibility: A schema_view identifier is visible to all other schema_views.

Scope: A schema_view declaration defines a new scope. This scope extends from the keyword SCHEMA_VIEW to the keyword END_SCHEMA_VIEW that terminates that schema_view declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a schema_view declaration:

- constant;
- function;
- map;
- procedure;
- rule;

- type_map;
- view.

10.3.2 View

Visibility: A view identifier is visible in the scope of the function, procedure, rule, or schema_view in which it is declared. A view identifier remains visible within inner scopes which redeclare that identifier.

Scope: A view declaration defines a new scope. This scope extends from the keyword VIEW to the keyword END_VIEW which terminates that entity declaration.

Declarations: The following EXPRESS-X items may declare identifiers within the scope of a view declaration:

- view expression;
- partition label.

10.3.3 View partition label

Visibility: A partition label is visible in the scope of the view in which it is declared.

10.3.4 View expression

Visibility: A view expression identifier is visible in the scope of the view in which it is declared.

11 Interface specification

This clause specifies the constructs that enable items declared in one schema, schema_view, or schema_map to be visible in another schema_view or schema_map. The REFERENCE specification allows enables item visibility.

Syntax:

```
interface_specification = reference_clause .
```

A foreign declaration is any declaration which appears in a foreign schema, schema_view, or schema_map (which is not the current schema_view or schema_map).

A foreign EXPRESS or EXPRESS-X item may be given a new name in the current schema_view or schema_map. The item shall be referred to in the current schema by the new name if given following the

AS keyword. This can be used in order to rename EXPRESS items that would otherwise use EXPRESS-X reserved words as their identifier.

11.1 Reference interface specification

A REFERENCE specification enables the following items, declared in a foreign schema, schema_view, or schema_map, to be visible in the current schema_view or schema_map:

– View;

– Map;

– Type_map;

– Constant;

– Entity;

– Function;

– Procedure;

– Type.

The REFERENCE specification gives the name of the foreign schema, and optionally the names of EXPRESS or EXPRESS-X items declared therein. If there are no names specified, all the items declared in the foreign schema, schema_view, or schema_map are visible within the current schema_view or schema_map.

Syntax:

```
reference_clause = REFERENCE FROM foreign_ref [ '(' resource_or_rename
                   { ',' resource_or_rename } ')' ] ';' .
foreign_ref = schema_ref | schema_view_ref | schema_map_ref .
```

Rules and restrictions:

To be completed.

11.2 Implicit interfaces

To be completed.

11.3 SCHEMA_MAP interfaces

A schema_map interface specification allows items defined in foreign schema to be visible within the schema map. It also define the source and target schemas.

Syntax:

```
77    interface_specification = source_interface_spec target_interface_spec
                                { map_interface_spec }
                                { external_functions_spec } .
```

11.3.1 Source schema interface

The source schema interface specifies the name of the source schema.

Syntax:

```
114  source_interface_spec = SOURCE [ source_schema_alias_id ':' ] schema_ref
                             { ',' [ source_schema_alias_id ':' ]
                             schema_ref } ';' .
```

11.3.2 Target schema interface

The target schema interface specifies the name of the target schema.

Syntax:

```
120  target_interface_spec = TARGET [ target_schema_alias_id ':' ] schema_ref
                             { ',' [ target_schema_alias_id ':' ]
                             schema_ref } ';' .
```

11.3.3 Map interface

The map interface specifies how one schema map may reference another.

Syntax:

```
88    map_interface_spec = IMPORT_MAPPING schema_map_ref
                           { ',' schema_map_ref } ';' .
```

11.3.4 External functions

The external functions interface specifies the interface to a function that is not defined in EXPRESS but is defined externally and will be called as part of the mapping. 111111

EXAMPLE 12 – A mapping may call an external function to convert geometry from advanced BREP into a faceted representation.

Syntax:

```
54   external_functions_spec = IMPORT_FUNCTIONS file_ref .
27   file_ref = file_id .
```

12 Expressions

Expressions are combinations of operators, operands, and function calls that are evaluated to produce a value. Anything that is an expression as defined in ISO 10303-11:1994 clause 12 is also a valid expression in EXPRESS-X. In addition, the following subclauses describe how expressions may be used with bindings and extents.

Precedence of operators and the order of evaluation of expressions are as defined in ISO 10303-11:1994 clause 12.

Entity constructors create instances that are local only to the function or procedure and do not exist in either the target or the source.

12.1 Explicit binding

An explicit binding is used to specify a particular member of an extent by providing instances which are bound to the variables in the FROM clause, using a function-like syntax. The result of an explicit binding is a binding data type.

If an `IDENTIFIED_BY` clause is present in the definition of the extent, then the arguments of the explicit binding shall match that clause.

EXAMPLE 13 – Explicit bindings are useful for describing a relationship between two views:

```
VIEW my_line;
FROM (l : line);
SELECT
  point : my_point := my_point(l.pnt);
  ...
END_VIEW;

VIEW my_point;
FROM (cp : cartesian_point)
  ...
```

12.2 Partial explicit binding

A partial explicit binding is an explicit binding in which one or more of the parameters is indeterminate. The result of a partial explicit binding is the subset of the extent that matches the parameter values that are provided.

EXAMPLE 14 – In the following, the various versions associated with a part are collected by using a partial explicit binding. The result will be the subset of the extent for which the second component of the binding is equal to the specified product instance.

```
VIEW part;
FROM (p : product)
SELECT
  versions : version_and_its_product
     := version_and_its_product(?, p);
END_VIEW;

VIEW version_and_its_product;
FROM (pdf : product_definition_formation, p : product)
WHERE p :=: pdf.of_product;
SELECT
  the_version : product_definition_formation := pdf;
END_VIEW;
```

12.3 Inline views

An inline view is the definition and evaluation of a view simultaneously. Inline views do not have names, and may not be bound to explicitly. The result of an inline view is an extent (i.e., a set of bindings).

EXAMPLE 15 – In the following example the versions of a part are collected by using an inline view.

```
VIEW part
FROM (p : part)
SELECT
  versions : product_defintion_formation
      := VIEW FROM (pdf : product_definition_formation)
              WHERE pdf.of_product :=: p;
END_VIEW;
```

An inline view can always be replaced with an explicit or partial binding to a named view.

12.4 Operations on extents

An extent is a set of bindings; as such, it may be used in expressions where a set is appropriate, and in particular an extent may be iterated over to visit each binding.

12.5 View expression evaluation

Given a binding, a view attribute expression may be evaluated. The result will be an instance in the underlying data set, a binding, or an extent.

12.6 Enhancements to expressions in assignment statements

EXPRESS-X extends EXPRESS expressions to introduce casting and references to manually instantiated entity instances.

Syntax:

```
expression = simple_expression [ rel_op_extended
             simple_expression ] .
rel_op_extended = rel_op | IN | LIKE | IS .
simple_expression = term { add_like_op term } .
add_like_op = '+' | '-' | OR | XOR .
term = factor { multiplication_like_op factor } .
multiplication_like_op = '*' | '/' | DIV | MOD | AND | '||' .
factor = simple_factor [ '**' simple_factor ] .
simple_factor = aggregate_initializer        | entity_constructor   |
                enumeration_reference        | interval             |
                query_expression             |
                ( [ unary_op ] ( '(' expression ')' | primary ) ) .
primary = literal | ([cast] qualifiable_factor { qualifier }) .
literal = binary_literal | integer_literal | logical_literal |
          real_literal   | string_literal .
cast = '{' simple_types | entity_id | type_id '}' .
qualifiable_factor =   attribute_ref | constant_factor |
                       function_call | general_ref     |
                       instance_ref  | population .
instance_ref = instance_id .
instance_id = '#' extended_id .
```

12.7 FOREACH expression

Description to be completed.

Syntax:

```
59   foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg
                             { AND variable_id IN foreach_in_clause_arg }
                    [ from_clause { from_clause } ] [ where_clause ]
                    RETURN attr_assgnmt_expr ';' .
60   foreach_in_clause = IN foreach_in_clause_arg
                         { ',' foreach_in_clause_arg } .
61   foreach_in_clause_arg = view_or_source_path_expr
                           | view_or_source_entity_ref .
62   foreach_return_expr = attr_assgnmt_expr .
```

12.8 Conditional map expression

Description to be completed.

```
Syntax:

84   map_cond_attr_expr = IF boolean_expression THEN map_attr_assgnmt_expr
                          [ ELSE map_attr_assgnmt_expr ]
                          END_IF ';' .
80   map_attr_decl_stmt_list = map_attribute_declaration
                               { map_attribute_declaration } .
82   map_attribute_declaration = [ target_entity_ref '.' ]
                                  target_attribute_ref ':='
                                  map_attr_assgnmt_expr ';' .
```

13 Executable statements

EXPRESS-X has sixteen types of statements for use inside MAP declarations. Many of these statement types are taken directly from EXPRESS. Those that are either not in EXPRESS or are modified from their definition in EXPRESS include the assignment statement, the FROM statement, the WHEN statement, the initialize statement, the DELETE statement, and the instantiation statement.

13.1 Enhanced assignment statement

For attribute assignment, the right hand side is evaluated as an EXPRESS expression and is then assigned to the left hand side.

The enhanced assignment statement is used to allow concatenation onto existing aggregates.

Syntax:

```
assignment_stmt = [coercion] general_ref { qualifier }
                  (':=' | '+=' | '-=') expression ';' .
```

An assignment statement is used to define the computation required to compute the value for an attribute in a mapping declaration and uses the typical assignment statement format found in programming languages such as Pascal and C. It is also similar to the assignment statement in EXPRESS. On the left of an assignment operator is the name of an entity attribute with any necessary qualifications (e.g., group or index qualifiers). On the right of the assignment operator is an expression, the value of which is used to initialize the attribute specified on the left of the assignment operator.

If the type of the value produced by the expression on the right of the assignment operator has a simple type (i.e., Number, Integer, Real, Boolean, Logical, String, or Binary), then this value is used to initialize the attribute specified to the left of the assignment operator. On the other hand, if the type of the value produced by the expression is not a simple type, then a pointer to the value is assigned to the attribute specified to the left of the assignment operator.

In addition to the standard assignment operator there is the a special version with the following meaning:

`A += expression` is equivalent to `A := A + expression`

This special form of the assignment operator is useful for efficient memory management when assigning values to attributes that are aggregate types.

13.2 FOR statement

The FOR statement is used to control the instantiation of target instances. Without the FOR statement, for each qualified source instance or set of source instances given by the FROM and WHERE clause one target instance of each entity listed in the mapping header (target_entity_ref_list). The FOR statement allows to instantiate more than one target instance.

The FOR statement consists of a loop-control statement and a block of attribute-assignment statements. The loop-control statement specifies the exact number of target instances that are built from each qualified source instance resp. set of qualified source instances. The block of attribute-assignment statements defines the mapping of all attributes of one or many target entities to some expressions.

Syntax:

```
49   entity_instantiation_loop = FOR instantiation_loop_control
                                 map_attribute_decl_block END_FOR ';' .
75   instantiation_loop_control = instantiation_foreach_control
                                  | instantiation_repeat_control .
74   instantiation_foreach_control = EACH variable_id IN
                                           source_attribute_ref
                                     { AND variable_id IN
                                            source_attribute_ref } .
58   for_loop_repeat_control = [ increment_control ] [ while_control ]
                               [ until_control ] .
```

The loop control statements defines either an iterator over elements of a source attribute of type aggregate or it defines an iterator in a very similar way to the EXPRESS repeat loop.

In the first case, a so-called unnest operation will be applied. That is, the loop-control statement defines an iteration over a source attribute of type aggregate. In each iteration step, the next element of the source attribute is assigned to a variable (variable_id). The value of this element can thus be used inside the FOR statement. For example, for each element of the source attribute of type aggregate a target instance can be generated and the element value can be assigned to a corresponding target attribute of type.

EXAMPLE 16 – In the following example, all item versions of one item are grouped together in the source data model. In contrast, each item version is a stand-alone instance in the target data model. This example shows that the FOR loop specifies an iteration over the elements of the source attribute `item_with_versions.id_of_versions`. For each source instance and for each element in that attribute a target instance is created. The target attribute `item_id` is mapped in the same way for all the target instances which of `item_version` which correspond to the same underlying `item_with_versions`. The target attribute `version_id` is assigned to the value of the iterator variable `version_iterator`.

```
ENTITY item_version; --target data model
  item_id    : STRING;
  version_id : STRING;
END_ENTITY;

ENTITY item_with_versions; -- source data model
  id             : STRING;
  id_of_versions : LIST OF STRING;
END_ENTITY;

MAP item_version
FROM iwv : item_with_versions;
  FOR EACH version_iterator OF iwv.id_of_versions
    item_id    := iwv.id;
    version_id := version_iterator;
  END_FOR;
END_MAP;
```

For example, the following target instances are built from the source instance below.

```
Target instance set:
  #1 = item_version(1,10);
  #2 = item_version(1,11);
  #3 = item_version(1,12);
Source instance set:
  #1 = item_with_versions(1,(10,11,12));
```

Alternatively, the repeat-loop control statement known from EXPRESS can be used to specify the iteration steps.

EXAMPLE 17 – In the following example, all item versions of one item are grouped together in the source data model. In contrast, each item version is a stand-alone instance in the target data model. This example shows that the FOR loop specifies an iteration over the elements of the source attribute `item_with_versions.id_of_versions`. For each source instance and for each element in that attribute a target instance is created. The target attribute `item_id` is mapped in the same way for all the target instances which of `item_version` which correspond to the same underlying `item_with_versions`. The target attribute `version_id` is assigned to the value of the iterator variable `version_iterator`.

```
SCHEMA target;

ENTITY parent;
END_ENTITY;

ENTITY child;
  parent : parent;
END_ENTITY;

END_SCHEMA; -- target
```

```
SCHEMA source;

ENTITY parent;
  children : INTEGER;
END_ENTITY;

END_SCHEMA; -- source
```

```
MAP tp : parent
FROM sp : parent
END_MAP;

MAP c : ALIAS [0:?] OF child
FROM p : parent
  FOR i := 1 TO p.children
    parent := parent();
  END_FOR;
END_MAP;
```

Alternatively, one single MAP can be specified for this example as shown below.

```
MAP  tp : parent, c : ALIAS [0:?] OF child
AS   group_parent_and_child
FROM sp: parent
  FOR i := 1 TO p.children
    parent := parent@group_parent_and_child();
  END_FOR;
END_MAP;
```

This statement can only be used within a MAP declaration.

14 Built-in functions and procedures

14.1 OID function

Returns the OID of an instance. Can only be compared to other OID.

Can only be used in the identified by clause.

Annex A
(normative)
EXPRESS-X language syntax

This annex defines the lexical elements of the language and the grammar rules that these elements shall obey.

NOTE — This syntax definition will result in ambiguous parsers if used directly. It has been written so as to convey information regarding the use of identifiers. The interpreted identifiers define tokens that are references to declared identifiers, and therefore should not resolve to simple_id. This requires a parser developer to enable identifier reference resolution and return the required reference token to a grammar rule checker.

All of the grammar rules of EXPRESS specified in annex A of ISO 10303-11:1994 are also grammar rules of EXPRESS-X. In addition, the grammar rules specified in the remainder of this annex are grammar rules of EXPRESS-X.

A.1 Tokens

The following rules specify the tokens used in EXPRESS-X. Except where explicitly stated in the syntax rules, no white space or remarks shall appear within the text matched by a single syntax rule in the following clauses.

A.1.1 Keywords

This subclause gives the rules used to represent the keywords of EXPRESS-X.

NOTE — This subclause follows the typographical convention that each keyword is represented by a syntax rule whose left hand side is that keyword in uppercase.

```
 1   ALIAS = 'alias' .
 2   EACH = 'each' .
 3   END_INLINE_FUNCTION = 'end_inline_function' .
 4   END_MAP = 'end_map' .
 5   END_SCHEMA_MAP = 'end_schema_map' .
 6   END_SCHEMA_VIEW = 'end_schema_view' .
 7   END_VIEW = 'end_view' .
 8   IDENTIFIED_BY = 'identified_by' .
 9   IMPORT_FUNCTIONS = 'import_functions' .
10   IMPORT_MAPPING = 'import_mapping' .
11   INLINE_FUNCTION = 'inline_function' .
12   MAP = 'map' .
13   OID = 'oid' .
14   PARTITION = 'partition' .
15   SCHEMA_MAP = 'schema_map' .
16   SCHEMA_VIEW = 'schema_view' .
17   SOURCE = 'source' .
18   TARGET = 'target' .
19   TYPE_MAP = 'type_map' .
20   VIEW = 'view' .
```

A.1.2 Character classes

```
21   digit = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’ .
22   letter = ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’g’ | ’i’ | ’j’ | ’j’
            | ’k’ | ’l’ | ’m’ | ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’
            | ’v’ | ’w’ | ’x’ | ’y’ | ’z’ .
23   simple_id = letter  { letter | digit | ’_’ } .
```

A.1.3 Interpreted identifiers

```
24   binding_ref = binding_id .
25   entity_ref = entity_id .
26   extent_ref = extent_id .
27   file_ref = file_id .
28   general_ref = parameter_ref | variable_ref .
29   network_ref = network_id .
30   schema_map_ref = schema_map_id .
31   schema_ref = schema_id .
32   source_schema_alias_ref = source_schema_alias_id .
33   source_schema_ref = source_schema_alias_ref | schema_ref .
34   target_entity_ref = ( target_schema_alias_ref | target_schema_ref ) ’.’
                         entity_ref .
35   target_schema_alias_ref = target_schema_alias_id .
36   target_schema_ref = target_schema_alias_ref | schema_ref .
37   view_entity_ref = view_entity_id .
```

A.2 Grammar rules

```
38   add_like_op = ’+’ | ’-’ | OR | XOR .
39   attribute_qualifier = ’.’ attribute_ref .
40   boolean_expression = expression .
41   bound_1 = numeric_expression .
42   bound_2 = numeric_expression .
43   built_in_constant = CONST_E | PI | SELF | ’?’ .
44   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 .
45   BY_clause = BY increment .
46   complex_entity_ref = entity_ref { AND entity_ref } .
47   constant_factor = built_in_constant | constant_ref .
48   entity_id = simple_id .
49   entity_instantiation_loop = FOR instantiation_loop_control
                                  map_attribute_decl_block END_FOR ’;’ .
50   entity_qualifier = ’.’ entity_ref .
51   entity_type_constraint = ’{’ supertype_expression ’}’ .
52   enumeration_reference = [ type_ref ’.’ ] enumeration_ref .
53   expression = simple_expression [ rel_op_extended simple_expression ] .
54   external_functions_spec = IMPORT_FUNCTIONS file_ref .
55   factor = simple_factor [ ’**’ simple_factor ] .
56   file_id = simple_id .
```

```
57   for_expr = foreach_expr
              | FOR for_loop_repeat_control
                RETURN attr_assgnmt_expr ';' .
58   for_loop_repeat_control = [ increment_control ] [ while_control ]
                               [ until_control ] .
59   foreach_expr = FOR EACH variable_id IN foreach_in_clause_arg
                             { AND variable_id IN foreach_in_clause_arg }
                  [ from_clause { from_clause } ] [ where_clause ]
                  RETURN attr_assgnmt_expr ';' .
60   foreach_in_clause = IN foreach_in_clause_arg
                         { ',' foreach_in_clause_arg } .
61   foreach_in_clause_arg = view_or_source_path_expr
                           | view_or_source_entity_ref .
62   foreach_return_expr = attr_assgnmt_expr .
63   function_call = ( built_in_function | xpx_built_in_function
                   | function_ref ) [ actual_parameter_list ] .
64   group_qualifier = '\' entity_ref .
65   identified_by_clause = IDENTIFIED_BY expression { ',' expression }
                            ';' .
66   identified_by_clause = UNIQUELY_IDENTIFIED_BY identified_by_expression
                            { ',' identified_by_expression }';' .
67   identified_by_expression = view_or_source_path_expr |
                                oid_built_in_function .
68   increment = numeric_expression .
69   increment_control = variable_id ':=' bound_1 TO bound_2 [ BY_clause ] .
70   index_qualifier = '[' index1 [ ':' index2 ] ']' .
71   inline_funct_body = .map_aggr_attr_expression = foreach_expr .
72   inline_function = INLINE_FUNCTION inline_funct_body
                       END_INLINE_FUNCTION ';' .
73   inline_view_decl = VIEW from_clause [ x_where_clause ]
                        [ project_clause ] END_VIEW ';' .
74   instantiation_foreach_control = EACH variable_id IN
                                         source_attribute_ref
                                    { AND variable_id IN
                                          source_attribute_ref } .
75   instantiation_loop_control = instantiation_foreach_control
                                | instantiation_repeat_control .
76   instantiation_repeat_control = [ increment_control ] [ while_control ]
                                    [ until_control ] .
77   interface_specification = source_interface_spec target_interface_spec
                               { map_interface_spec }
                               { external_functions_spec } .
78   logical_expression = expression .
79   map_attr_assgnmt_expr = expression
                           | map_cond_attr_expr
                           | for_expr
                           | inline_function
                           | map_ call .
80   map_attr_decl_stmt_list = map_attribute_declaration
                             { map_attribute_declaration } .
81   map_attribute_decl_block = entity_instantiation_loop
                                | map_attr_decl_stmt_list .
82   map_attribute_declaration = [ target_entity_ref '.' ]
                                 target_attribute_ref ':='
                                 map_attr_assgnmt_expr ';' .
83   map_call = target_entity_ref [ '@' network_or_partition_qualification ]
                '(' expression { ',' expression } ')' .
84   map_cond_attr_expr = IF boolean_expression THEN map_attr_assgnmt_expr
                          [ ELSE map_attr_assgnmt_expr ]
                          END_IF ';' .
```

```
 85   map_decl = MAP map_decl_header   [ map_partition_decl ] map_decl_body
                                 { [ map_partition_decl ] map_decl_body }
            END_MAP ';' .
 86   map_decl_body = from_clause [ where_clause ] [ identified_by_clause ]
                 [ project_clause ] [ proj_clause_or_map_attr_decl ]
                 END_VIEW ';' .
 87   map_decl_header = target_entity_ref_list [ AS group_name ] .
 88   map_interface_spec = IMPORT_MAPPING schema_map_ref
                      { ',' schema_map_ref } ';' .
 89   map_partition_decl = PARTITION
                      partition_id ':' [ partition_entities ] .
 90   multiplication_like_op = '*' | '/' | DID | MOD | AND | '||' .
 91   network_or_partition_qualification = network_ref | partition_ref
                                      | network_ref '.' partition_ref .
 92   numeric_expression = expression .
 93   oid_built_in_function = OID '(' (view_or_source_entity_ref |
                         view_or_source_entity_alias_ref) ')' .
 94   partition_entities = target_entity_ref_list .
 95   population = entity_ref .
 96   primary = literal | ( qualifiable_factor { qualifier } ) .
 97   proj_clause_or_map_attr_decl = project_clause
                                | map_attribute_decl_block .
 98   proj_clause_or_view_attr_decl = project_clause |
                                 view_attr_decl_stmt_list .
 99   project_clause = SELECT view_or_source_entity_ref .
100   qualifiable_factor = attribute_ref | constant_factor | function_call
                      | general_ref | population | schema_ref |
                      '.' entity_ref source_schema_alias_ref |
                      target_schema_alias_ref | view_entity_ref |
                      '.' view_attribute_ref source_entity_ref .
101   qualifier = attribute_qualifier | group_qualifier | index_qualifier
             | entity_qualifier | view_attribute_qualifier .
102   rel_op = '<' | '>' | '<=' | '>=' | '<>' | '=' | ':<>:' | ':=:' .
103   rel_op_extended = rel_op | IN | LIKE .
104   schema_id = simple_id .
105   schema_map_body = interface_specification [ type_mapping ]
                   { declaration } .
106   schema_map_body = view_or_map_decl { view_or_map_decl } .
107   schema_map_decl = SCHEMA_MAP schema_map_id ';' schema_map_body
                   END_SCHEMA_MAP ';' .
108   schema_map_id = simple_id .
109   schema_view_body = { reference_clause } [ constant_decl ]
               { function_decl | procedure_decl | view_decl } .
110   schema_view_decl = SCHEMA_VIEW schema_view_id schema_view_body
               END_SCHEMA_VIEW ';' .
111   simple_expression = term { add_like_op term } .
112   simple_factor = aggregate_initializer | entity_constructor
                 | enumeration_reference | interval | query_expression
                 | ( [ unary_op ] ( '(' expression ')' | primary ) ) .
113   source_entity_ref = [ source_schema_ref '.' ] entity_ref .
114   source_interface_spec = SOURCE [ source_schema_alias_id ':' ] schema_ref
                         { ',' [ source_schema_alias_id ':' ]
                         schema_ref } ';' .
115   source_schema_alias_id = simple_id .
116   subtype_of_clause = SUBTYPE OF view_extent_ref
                   [ PARTITION partition_ref ] [ where_clause ] ';' .
117   supertype_expression  = .
118   target_entity_ref_list = target_entity_ref_list_el
                          { ',' target_entity_ref_list_el } .
```

```
119   target_entity_ref_list_el = [ target_entity_ref_alias_id ':'
                                   [ ALIAS '[' bound_1 ':' bound_2 ']' OF ] ]
                                 target_entity_ref .
120   target_interface_spec = TARGET [ target_schema_alias_id ':' ] schema_ref
                             { ',' [ target_schema_alias_id ':' ]
                             schema_ref } ';' .
121   target_schema_alias_id = simple_id .
122   term = factor { multiplickation_like_op factor } .
123   type_expression = .
124   type_map_stmt_body = [ schema_ref '.' ] base_type ':='
                          type_expression ';' .
125   type_mapping = type_mapping_stmt { type_mapping_stmt } .
126   type_mapping_stmt = TYPE_MAP target_type_ref { target_type_ref } FROM
                         source_type_ref { source_type_ref } ';'
                         { type_map_stmt_body } END_TYPE_MAP ';' .
127   until_control = UNTIL logical_expression .
128   view_aggr_attr_expression = inline_view_decl | foreach_expr .
129   view_attr_assgnmt_expr = expression
                              | view_cond_attr_expr
                              | view_aggr_attr_expression
                              | view_ call .attr_assgnmt_expr = expression
                              | view_or_map_aggr_attr_expression
                              | view_or_map_call .
130   view_attr_decl_stmt_list = view_attribute_decl
                                { view_attribute_decl } .
131   view_attribute_decl = view_attribute_id ':'
                           [ source_schema_ref '.' ] base_type
                           ':=' view_attr_assgnmt_expr ';' .
132   view_attribute_qualifier = '.' view_attribute_ref .
133   view_body =  [ proj_clause_or_view_attr_decl ] .
134   view_call = view_entity_ref '(' expression { ',' expression } ')' .
135   view_cond_attr_expr = IF boolean_expression THEN view_attr_assgnmt_expr
                           [ ELSE view_attr_assgnmt_expr ]
                           END_IF ';' .
136   view_decl = view_head view_body END_VIEW ';'
137   view_decl = view_head view_partition { view_partition } END_VIEW ';' .
138   view_entity_id = simple_id .
139   view_from_clause = FROM '(' view_parameter { ';' view_parameter } ')'
                        [ where_clause ] [ identified_by_clause ]';' .
140   view_from_clause = FROM [ view_parameter ':' ] extent_ref
                          { ','[ view_parameter ':' ] extent_ref }
                          [ where_clause ] ';' .
141   view_head = VIEW view_id ';' .
142   view_head = VIEW view_id view_from_clause [ identified_by_clause ]
143   view_or_map_aggr_attr_expression = view_aggr_attr_expression
                                        | map_aggr_attr_expression .
144   view_or_map_call = view_call | map_call .
145   view_or_map_decl = view_decl | map_decl .
146   view_or_source_entity_alias_id = simple_id .
147   view_or_source_entity_ref = view_entity_ref | source_entity_ref .
148   view_or_source_path_expr = [ schema_ref '.' ]
                                entity_ref [ '\' entity_ref ] '.'
                                { attribute_ref [ entity_type_constraint ]
                                [ '\' entity_ref ] '.' } attribute_ref
                                | view_ref '.' { attribute_ref
                                [ entity_type_constraint ]
                                [ '\' entity_ref ] '.' } attribute_ref .
149   view_parameter = parameter_id { ',' parameter_id } ':'
                      view_parameter_type .
```

```
150   view_parameter_type = entity_ref | view_extent_ref | SET [ bound_spec ]
                             OF ( entity_ref | view_extent_ref ) .
151   view_partition = [ PARTITION partition_id ':' ] ( view_from_clause |
                     subtype_of_clause ) view_body .
152   where_clause = WHERE domain_rule { ';' domain_rule } .
153   while_control = WHILE logical_expression .
154   xpx_built_in_function = OID .
```

A.3 Cross reference listing

To be completed.

Annex B
(informative)
Bibliography

EXPRESS-V language (ISO TC184/SC4/WG5 N251).

EXPRESS-M language (ISO TC184/SC4/WG5 N243).

BRITTY language.

Wirth, Niklaus, "*What can we do about the unnecessary diversity of notations for syntactic definitions?,*" Communications of the ACM, November 1977, v. 20, no. 11, p. 822.

Index