ISO TC184/SC4/QC/N163
Date: 2000-09-21

Reference Path Syntax of Mapping Tables

Alfonsas Stonis (alfonsas.stonis@lksoft.com)
LKSoftWare GmbH (http://www.lksoft.com)

Contents

Scope

STEP Application Protocols (AP) consist of an Application Reference Model (ARM), Application Integrated Model (AIM) and a Mapping Table between the ARM and the AIM. The Mapping Table consists of 5 columns:

- Application Object
- AIM Element
- Source
- Rules
- Reference Path

This document defines a formal specification of the data to the Reference Path. The syntax of the specification is given as WSN (similar to the definitions in EXPRESS). This document fits also for future Application Protocols which may use a Mapping Specification in the form of nested paragraphs instead of a table (see ISO TC184 SC4 N1029).

This specification does not cover

- style guide lines, e.g. when to make a new line
- problems related to bi-directional mapping.
- tool support, e.g.: there is no information about how to test or prove that a tool supports this mapping specification. It also does not provide information how to report or detect errors in a mapping specification.

Most people think that the mapping specifies how to map a hypothetical ARM population into an AIM population. But, in fact, the mapping defines only how to convert some mating AIM population to the ARM. Many practical population problems from ARM to AIM are not addressed by the mapping table.

Background

Till today there exists no formal specification of the Reference Path. The "STEP ARM-AIM Mapping Table Language MTL2.0, Draft", March 12, 1996, from Steven J. Clark and James D. Kindrick defines some details of the Reference Path,

but the given structural rules are not sufficient.

LKSoft has developed a MappingTableCompiler to convert the mapping tables of AP210, 212, 214 and others into a population of the mapping_schema, which is an extension of the SDAI_dictionary_schema (see WG11N050 and N051). This tool is adjusted to the so called "M" format used by AP212/214 and the sgml format of AP210.

While compiling Mapping Tables many bugs where detected and reported back to the AP developers. Another major problem was to deal with the many variants how references paths are written. Till now no strict order was defined how to arrange AND conditions, OR conditions, sub-supertype relations, constraints and real path information. This proposal specifies one variant out of the many possible and used ones. The variant chosen is the one which is most strict and uniquely identifiable. The analyzed APs 210, 212 and 214 would need only minor editorial changes to be in accordance with the specification given here.

Mapping Tables are not strict with what is meant by "entity". An entity can be a simple entity data type, a complex entity data type or a complex entity data type with several leaves (see ISO 10303-11:1994 and ISO 10303-21:1994). Usually complex entity data types with several leaves are written in the following form `[leave1]... [leaveN]`. But this leads to a confusion with AND conditions for which also [...]...[...] is used. To avoid this confusion in this specification the leaves of complex entity data types are combined with a "+", similar as it is defined in annex A.1.3 of ISO 10303-22.

The end of a reference path is indicated by a "**.**".

There exist cases when alternative paths in the Reference Path which join later on, e.g.:

... (A.a1->B)(A<-B.b1) (B.b2="xx")(B.b2="yy") ...

Such a situation is very hard to recognize. Because of such and similar situations the semicolon ";" as reference path separator is proposed. It increases the readability of the mapping and highly simplifies development of mapping compilers. So, for users it is proposed to write

... (A.a1->B)(A<-B.b1) ; (B.b2="xx")(B.b2="yy") ...

A possible alternative would be to write (A.a1->B)(A<-B.b1) B (B.b2="xx")(B.b2="yy") instead, but then the grammar would be more complicated.

WSN notation

Like in EXPRESS the Wirth Syntax Notation (WSN) is adopted for this specification. The following notation elements are used:

`[]` - optional
`{}` - zero or more
. – the end of the production.

Lexical elements

The following rules specify how certain combinations of characters are interpreted as lexical elements within the mapping syntax:

```
digits = digit {digit}
digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
entity_name = complex_entity_name | simple_entity_name.
simple_entity_name = id.
complex_entity_name = id "+" id {"+" id}.
select_type = id.
attribute = entity_name "." attribute_name.
attribute_name = id.
type = id.
```

```
id = letter {digit | letter | "_"}.
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" |
"o" | "p" | "q" | "r" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
```

Grammar rules

The following rules define how mapping specification is constructed from lexical elements. White space (white space, tab symbol or new line symbol) and/or remark(s) may appear between any two tokens in these rules.
The top production rules are `entity_reference_path` and `attribute_reference_path`.

```
enity_reference_path = [constraint] ".".
attribute_reference_path = [path_elements] ".".
path_elements = path_element {";" path_element}.
constraint = "{" [path_elements ";"] path_element | constraining_element) "}".
path_element = constraint | path.
path = or_element | backward | forward | repeat | strict_entity.
strict_entity = "|" entity_name "|".
or_element = "(" path_elements ")" "(" path_elements ")" {"(" path_elements ")"}.
backward = id_backward "<-" attribute_backward.
forward = attribute [aggregate] "->" id_forward {subtype}.
id_backward = entity_name {select_backward | nested_aggregate_backward}.
select_backward = "=" select_type.
nested_aggregate_backward = "=" aggregate_type_name aggregate.
aggregate = "[" ("i" | digits) "]".
attribute_backward = attribute aggregate {subtype}.
subtype = "=>" entity_name.
id_forward = {select_forward | nested_aggregate_forward} type.
select_forward = select_type "=".
nested_aggregate_forward = aggregate_type_name aggregate "=".
constraining_element = attribute [aggregate] ("=" | ("->" (select_forward |
nested_aggregate_forward) {select | nested_aggregate}) value.
value = enumeration_value | integer_value | string_value | logical_value | real_value.
repeat = "(" path_elements ")*".

template_usage = "/" template_name "(" [parameter {"," parameter}] ")/".
parameter = entity_name | attribute_name | value.

template = template_definition template_body.
template_definition = "/" template_name "(" parameter_definition {"," parameter_definition}
")/".
parameter_definition = id.
template_body = {path_elements ";"}.

remark = "(*" {letter | digit | remark} "*)".
```

Elements of the syntax

digits

A sequence of digit. `digits` are used for indexing of aggregate members.
Examples are "5", "83".

digit

One giphen.

entity_name

This is either a simple_entity_name or a complex_entity_name.

simple_entity_name

The name of a simple entity data type or a complex entity data type with only one leaf. This notation does not distinguish between these two cases.
Examples are "product", "cartesian_point".

Note: If there is a need to limit to some specific complex entity data type (excluding its subtypes) the `strict_entity` (to be defined bellow) notation shall be used.

complex_entity_name

There are two ways how to specify a complex entity data type with several leaves. One way is to list only the leave entity data types. The other way is to write all entity data types the complex entity data type consist of. Both ways are allowed. The only requirement is that the listed entities define a valid complex entity data type -- this trivial requirement was not always met in existing mapping tables. Also complex entity data types with only one leaf can be written in this style.
Examples are "area_unit+si_unit", "bacground_colour+colour_rgb".

Note: This notation does not prohibit instances of some other complex entity data types that include the specified entity data type. If this is required, the `strict_entity` notation shall be used.

select_type

The name of a select data type.
Example: "person_or_organization".

attribute

The name of an entity and an its attribute ("`entity.attribute`").

attribute_name

The name of an attribute (without the entity name). The attribute may be a part of the preceding entity or of one of its supertypes.
Example: "`name`", "`description`", "`items`".

type

The name of a defined data type.
Example: "`label`", "`text`", "`approved_item`".

id

An EXPRESS identifier.

letter

A single letter from the Latin alphabet.

entity_reference_path

It can be used for entity mapping reference path or for attribute mapping reference path in the case of identical mapping. The first element of the entity reference path must start from the same entity data type as it is described in the AIM element from

the mapping of ARM entity.

attribute_reference_path

It is used only to express attribute reference path. Path elements that are not within braces "`{`" and "`}`" defines attribute reference path. Elemets within "{" and "}" define addition restrictions on the path. In the case of identical mapping attribute reference path may be empty. The first element in attribute reference path must start from the same type as entity mapping.

constraint

Constraint defines restrictions on the entity instance but does not define a path. It only limits possible values to some subset, but does not say how to go from one value to another.
Example: "`{A.a = 'name'}`".

path

Path is one of several possible reference path elements to travel from one set of instances to another set or to the set of values.

path_elements

It is one or more path elements separated by semicolon ";". The end type of previous `path_element` and the start type of next `path_element` must be the same.

path_element

It defines a way how to go from one set of entity instances to another set of instances. `path_element` may also define how to go from the set of instances to the set of some simple values (strings, numbers, enumeration elements). An attribute of entity is one of the kernel parts of `path_element` and it is always present. `path_element` never defines any constraints on the incoming set.

strict_entity

`strict_entity` restricts the set of instances to the specified entity data type. It does not allow to have instances of any other entity data type.

or_element

It is a short cut to write alternative mappings. The other way is to define complete mappings for every possible mapping alternative. o`r_elements` may be numbered. If they are numbered, the same numbering must be used in whole mapping of entity and its attributes.
Example: "`(A.a1 -> B)(A.a2 -> C; C.c1 ->B)(A <- B.b1);`","`({A.name = 'xx'})({A.name = 'zz'};`
`({A.description = 'z'})({A.description = 'x'}););`","`(A.a1 -> B)(A <- B.b1);(B.b2 = 'x')(B.b2 = 'y')`".

backward

This element is for inverse attribute relationships (the set of instances that is the beginning of the backward element is referenced by attribute of the set of instances that is the result of backward operation). The end set of backward operation may be limited to a set of subtypes of the entities that defines an attribute which is used in backward operation.
Example: "`A <- B.b1`","`A <- C.c1[i]`".

forward

Forward element defines simple (direct) attribute usage. It may define additional constraints on the end set of instances to

limit it to more specific subtype.
Example: "`A.a1 -> C`","`A.a2 -> D = B`".

id_backward

It defines only the name of entity that is referenced by some attribute. It can be the simple entity name, the select data type and entity name, aggregate and entity name or combination of these. Aggregate is used to show constraints on nested aggregates. `id_backward` and `id_forward` differs in that they are used for backward and forward relationships, and because of this the sequence of elements is different, even if they consist of the same elements.
Example: "`A`","`A = D = C[i]`".

select_backward

This element is used to select an item of a select data type, but only for backward elements. It includes equal sign "=" and name of select type.
Example: "`= B`".

nested_aggregate_backward

This element is only for backward references. It includes equality sign "=" followed by aggregate type name (aggregate must be defined as Express named type) and the aggregate symbol with aggregate member index in it.
Example: "`= C[i]`".

aggregate

Aggregate symbol is used to define which aggregate member is used in reference path. The exact member of aggregate can be specified by writing its index. It can be imposed only on those aggregates that allow access by index. Index may be left unspecified, writing "`i`" instead of the index position, then every element of aggregate can be used to met mapping constraints.
Aggregate member restriction does not constrain the size of the aggregate. For such purpose Express rules should be used. It also can not be used to restrict the number of elements referencing or referenced by the aggregate (for example: it can not be used to impose a restriction that only one instance from aggregate can reference to this particular instance or to an instance of some particular type). For this purpose the Express rules should be applied. Constraints of this kind are outside mapping constraints and the scope of mapping syntax.
Example: "`[i]`","`[3]`".

attribute_backward

It defines attribute that is used in backward operation. It consists of entity name, dot symbol "`.`" and attribute name. The entity can be reduced to some its subtype.
Example: "`A.a1`","`A.a2[i]`".

subtype

Using subtype elements, both forward and backward attribute relationships can be restricted to some specific subtype. In both cases subtype element constrains the last set that is the result of attribute relation operation to some specified subtype of the entity. The specified entity must be a subtype of the entity data type that is the ending data type of the path if the `subtype` would be not used.
Example: "`=> B`".

id_forward

It defines an identifier for forward attribute relationship. In the simplest case it can be only the name of the entity. In more complex case it can be item of select data type or selection of member for aggregate.
Example: "`A`","`C[i] = A`","`D = C[1] = A`".

select_forward

It defines the selection of one alternative from all possible select items. It consists of select type followed by equality sign "=".
Example: "`D =`".

nested_aggregate_forward

It defines the nested aggregates for forward attribute reference. It is used to address only the nested aggregates or aggregates within select.
Example: "`C[1] =`", "`C[i] =`".

constraining_element

Constraining element is an element that constrains value of an attribute. The value can be string, number or enumeration element. This element can be used only within `constraint` and only as the last element of `constraint`.
Example: "`A.a3 = 'name'`", "`A.a4 = .T.`".

value

`value` defines all valid values for `constraining_element`: enumeration_value, integer_value, string_value, logical_value, real_value.
Example: "`.BUY.`", "`32`", "`'name'`", "`.T.`", "`21.4`".

repeat

It defines statement that can be repeated zero or more times. The statements within this construct must start and end with the same entity data type or at the end of these statements there can be subtype of the start entity data type. There should be at least one path statement in `repeat` statement.
Example: "`(A <-B.b1; B.b2 -> A)*`".

template_usage

It defines how to use templates in reference path. It includes special symbols "/" to show that the used name is template and optional list of parameters for the template.

parameter

Parameter is a part of the path that is passed to a template. It can be a name of the entity, attribute name, combination entity name and attribute name or some value.

template

Template is a reusable part of the reference path. Template itself does not define any mapping, but may be used in many mappings to simplify the mapping path. The templates consist of two parts: template definition and template body. Template definition defines how template will appear and how it will be used in the reference path. Template body defines the a part of the reference path that will be inserted instead of the `template_usage` in some reference path.
Template is a very powerful tool to shrink mapping tables. The main problem with the templates is that they are hard to compile. The errors in templates can not be detected at the time when the templates are compiled. Errors are detected only at the time when instantiation of a template is checked. However, templates may also help to prevent some errors, because it is simple to write and to check the code once instead of doing this many times.

template_defintion

`template_definition` defines the signature of the template. It consists of the name of the template and the parameter list for this template. The later is optional.

parameter_definition

`parameter_definition` defines the parameter that may be passed to the template. At the template instantiation time the name of the parameter will be replaced with the value that is passed to the template. The current definition of `parameter` does not specify the type of parameter. It is very flexible but also very dangerous way to describe parameters of the template. This means that simple string replacement is used to replace the values for the templates. This allows constructing of various templates the meaning of which will be dependent on the value that is passed as a parameter. It also makes big problems for tool (for humans too) to correctly instantiate the values that are passed as parameters.
An alternative way to define parameter is to include the name and the type of the parameter. Then it is much easier to test that correct values are submitted as parameters to the template. In this parameter_definition will look like this:

```
parameter_definition = id : type.
```

The small modification of parameter allows preventing some errors while using templates. It also makes easier to use and to read templates, because one already knows the type what can be a value for parameter.

template_body

Template body contains `path_elements`. The only difference from the usual path is that it allows the usage of template parameters.

remarks

It defines the comments that can be inserted in the mapping specification. The contents of the comment may be any symbol except a combination of the closing parentheses for the comments `"*)"`. Remarks may be nested.

Not supported cases:

Incomplete constraints.

Constraints like `"(A.a1->)(B.b1->)C"` are not supported. It is impossible or at least very hard to write grammar rules that will support such cases without allowing ambiguity.

Supertype constraint

`"A <= B"` A is a subtype of B. This construct is not included because it is redundant. It also makes problems to parse the mapping syntax if the path separator `";"` is not used. This constraint may be added to the mapping syntax by changing the `path` and adding new rule `supertype`.

```
path = or_element | backward | forward | repeat | strict_entity | supertype.
supertype = entity_name "<=" entity_name.
```

"AND" constraint

```
[A.a1 = 'name'][A.a2 = 'description']
```

The value of A.a1 must be `'name'` and the value of A.a2 must be `'description'`. This constraint is not included because it may yield illegal constraints that are much harder to find and to correct. It may be easily replaced with the pairs of `'constraint_element'`. For the above example as follows:

```
{A.a1 = 'name'};{A.a2 = 'description'};
```

The main problem related to this constraint is usage of it. If these is AND constraint that consist of two parts, the after following this constraint you have two choices:
1. The result is a set of values that consist of pairs (one element of the pair from each part).
2. The result is a set of instances where for every instance in this set there are both and constraints satisfied.

Single subtype constraint

This constraint is indirectly included in mapping syntax. It is a part of attribute constraint. There was two problems related to this construct.

1. Using supertype constraint it is rather easy to construct the names of entities that are hard to read and to test. Such names can be written as complex entity names in more readable form.
2. This construct was used in mapping of an attribute to constrain its start type to a subtype of the entity to which its parent entity is mapped. For example, ARM entity is mapped to entity A, but the mapping of an attribute of this entity requires some subtype of A. Such constraints are not easy to detect. When implementing mapping you have first check whether there are such restrictions in mapping of attributes. An alternative to this is to explicitly specify the subtype in AIM element of ARM entity mapping.

This constraint may be incorporated to the mapping syntax by changing the `path` and adding new rule `single_supertype` (there is already rule `'supertype'`).

```
path = or_element | backward | forward | repeat | single_supertype.
supertype = entity_name => entity_name.
```

The first entity must be subtype of the second entity.

Single entity constraint

In the current mapping tables often it is very hard to distinguish where one constraint ends and another starts (in this mapping specification this problem is solved by introducing semicolon "`;`" as path separator). One way to help to do this was to use the entity name. Such a construct is no longer needed and is deprecated. If there is a need to add it for compatibly reason it can be added by changing the `path` construct.

```
path = or_element | backward | forward | repeat | strict_entity | entity_name.
```

Indexing of alternatives

There are two cases when it is useful to index alternatives.

Alternatives within one entity mapping

For one ARM entity there can be several alternative mappings. There are two ways how to show this. One way is to write alternatives in separate mappings, another way is to highlight the differences. The indexing of alternatives may be used to show the differences between separate mappings if they are written as one mapping. The indexing can be used only for the or_elements. The current syntax for indexing is as follows:

```
or_element = ["#" digits ":"] "(" path_elements ")" ["#" digits ":"] "(" path_elements ")" {["#"
digits ":"] "(" path_elements ")"}.
```

It would be more nice to move the index into or_element and have it as:

```
or_element = "(" ["#" digits ":"] path_elements ")" "(" ["#" digits ":"] path_elements ")" {"(" ["#"
digits ":"] path_elements ")"}.
```

This writing, maybe, looks strange at the first time, but it is easier to read and to parse with a computer and it prevents from making some errors.

Alternatives two entities relationship mapping

Assume that there is two alternatives of one entity mapping. Both alternatives are of the same AIM type, but they differs by mapping constraints. Now we have mapping of another entity. This entity has some attribute. This attribute points to the first entity. There are two mapping alternatives for this attribute. To link the attribute mapping alternatives to the entity mapping alternatives the indices of the entities must be linked with the indices of attribute mapping. This can be done in attribute mapping (now it is in the first column) by writing the name of the ARM entity and the index of its mapping (now only the name of ARM entity is written). This simple modification would allow relate alternatives explicitly instead of repeating the same constraints for the attribute and the entity.

Repeating of identifiers

Sometimes the Express identifiers (the names of entity data types and defined types) are repeated for every Reference path element. For example, "`A.a1 -> C C = B`". The new syntax proposes shortened notation "`A.a1 -> C = B`". Using this style, the reference path is shorter and more readable (the syntax rules are also simpler). If there is a need to have these additional identifiers, they can be added by changing several rules. Modified rules will be as follows:

```
id_backward = entity_name {select_backward | nested_aggregate_backward}.
select_backward = type "=" select_type.
nested_aggregate_backward = type "=" aggregate_type_name aggregate.
id_forward = type {select_forward | nested_aggregate_forward}.
select_forward = select_type "=" type.
nested_aggregate_forward = aggregate_type_name aggregate "=" type.
```

Mapping of ARM subtype relationships

There is no information how subtype relationships from the ARM are mapped to the AIM. There exist cases when the mapping of the ARM entity does not fulfill mapping requirements of the mappings of the supertype of this ARM entity. This problem is not solved there.

Mapping of Boolean and enumeration

If the domain for ARM attribute is boolean or enumeration type, then there is no information in mapping specification how to map the values of these attributes. For example there are information what is allowed values for the AIM instances, but there is no information when the value for the ARM attribute will be TRUE or the FALSE. This problem is not solved there.

Migrating to new syntax

The usage of proposed mapping syntax will make current mapping tables computer interpretable. This will allow to check them with the tools and to achieve better quality of mapping tables. The changes required by new mapping syntax are only editorial and the migration to the new syntax is quit strict. The following example illustrates the differences between old and new style of mapping. The example of old style of mapping is taken from SC4-N1029.

Old mapping	New mapping
ARM entity: approval; AIM element: approval; Source: ISO 10303-41; Reference path:	ARM entity: approval; AIM element: approval; Source: ISO 10303-41; Reference path:.
ARM attribute: date; AIM element: calendar_date; Source: ISO 10303-41; Reference path: approval <- approval date_time.dated_approval approval_date_time approval_date_time.date_time -> date_time_select date_time_select = date date => date calendar_date	ARM attribute: date; AIM element: calendar_date; Source: ISO 10303-41; Reference path: approval <- approval date_time.dated_approval; approval_date_time.date_time -> date_time_select = date => calendar_date.
ARM attribute: approval to organization; AIM element: PATH; Reference path: approval <- approval_person_organization.autohorized_approval approval_person_organization approval_person_organization.person_organization -> person_organization_select #1: (person_organization_select = person person) #2: (person_organization_select = organization organization)	ARM attribute: approval to organization; AIM element: PATH; Reference path: approval <- approval_person_organization.autohorized_approval; (#1: approval_person_organization.person_organization ->> person_organization_select = person) (#2: approval_person_organization.person_organization ->> person_organization_select = organization) (#3:

#3: (person_organization_select = person_and_organization person_and_organization)	approval_person_organization.person_organization -> person_organization_select = person_and_organization).
ARM entity: drawing; AIM element: draughting_drawing_revision; Source: ISO 10303-201; Reference path: draughting_drawing_revision <= drawing_revision	ARM entity: drawing; AIM element: draughting_drawing_revision; Source: ISO 10303-201; Reference path:. (* the old reference path carries no information *)
ARM attribute: drawing_number; AIM element: drawing_definition.drawing_number; Source: ISO 10303-101; Reference path: draughting_drawing_revision <= drawing_revision drawing_revision drawing_revision.drawing_identifier -> drawing.definition drawing_definition.drawing_number	ARM attribute: drawing_number; AIM element: drawing_definition.drawing_number; Source: ISO 10303-101; Reference path: drawing_revision.drawing_identifier -> drawing.definition.
ARM attribute: drawing_revision_id; AIM element: drawing_revision.drawing_identifier; Source: ISO 10303-101; Reference path: draughting_drawing_revision <= drawing_revision drawing_revision drawing_revision.drawing_identifier	ARM attribute: drawing_revision_id; AIM element: draughting_drawing_revision.drawing_identifier; Source: ISO 10303-101; Reference path:. (* the old reference path carries no information *)
ARM attribute: drawing to approval; AIM element: PATH; Reference path: draughting_drawing_revision <= drawing_revision drawing_revision approved_item = drawing_revision approved_item <- draughting_approval_assignment.approved_items[i]. draughting_approval_assignment <= approval_assignment approval_assighment.assigned_approval -> approval	ARM attribute: drawing to approval; AIM element: PATH; Reference path: draughting_drawing_revision = approved_item <- draughting_approval_assignment.approved_items[i]; draughting_approval_assignment.assigned_approval -> approval.
ARM attribute: dwaring to drawing_sheet; AIM element: PATH; Reference path: draughting_drawing_revision <= drawing_revision <= presentation_set <- area_in_set.in_set area_in_set {area_in_set => drawing_sheet_revision_usage} area_in_set.area -> presentation_area => drawing_sheet_revision	ARM attribute: dwaring to drawing_sheet; AIM element: PATH; Reference path: draughting_drawing_revision <- area_in_set.in_set => drawing_sheet_revision_usage; area_in_set.area -> presentation_area => drawing_sheet_revision.