Appendix B: An Ontology of Higher-Level Constraints

This appendix is supplementary to Section 6.4. Its purpose is to analyse what higher-level constraints (HLCs) are needed to support a comparative understanding of constraints on information models. This kind of understanding will be needed for tasks such as schema-to-schema mapping (Section 1.3), where constraints on the target model are to be transferred to the source model (just those constraints that are not implied by the source model anyway) so that those constraints can be checked for records in the source repository *before* they are mapped to the target repository (see Section 1.3.2).

HLCs serve to characterise the abstract semantics of the constraints on information models, which enables abstract similarities to be seen between fragments of EXPRESS [34] code which look very different at surface level. HLCs will be useful for assessing the semantic equivalence between different models, i.e. for realising that (or to what extent) two things written differently mean the same thing, and what the differences are (they may be only partially equivalent, or not equivalent at all). Although it is beyond the scope of this thesis actually to compare HLCs, it will provide a useful building block for doing so in the future.

The product of this appendix will be an ontology of HLCs. *Ontology* is a word that means different things in different contexts. Here, it means a classification of the kinds of HLCs that will be useful for supporting comparative understanding of constraints. The approach taken is to analyse different models of the same things, in which equivalent constraints are expressed in different ways. From analysing these examples, HLCs are invented to characterise the abstract semantics of the constraints on the example models. Cases are a natural by-product of this analysis, as they record the association between EXPRESS code fragments (or lower-level HLCs) and the HLCs which characterise the semantics of the constraints implied by those fragments (or lower-level HLCs).

True to the spirit of case-based reasoning, this appendix will present a series of worked examples in which fragments of models are analysed for their HLCs (and hence cases to suggest those HLCs). The ontology is by no means supposed to be complete - it does not cover every construct in EXPRESS, nor does it attempt to specify exhaustively every possible HLC and every possible case to extract it. The possibilities are infinite, so the best that can be hoped for is a sensible subset of HLCs which arise from experience (based on the premise that what was useful in the past is likely to be useful in the future). The set of HLCs and cases can be expanded to provide more coverage as needed. This appendix therefore just tries to give a flavour of the kinds of HLCs that might be useful. What is really important is the process of inventing HLCs (and collecting cases) by means of analysing worked examples. It would be possible to expand the ontology by continuing along similar lines, driven by the need to cover more aspects of the constraints on models.

Section A.2 of Appendix A, which contains the EXPRESS model for the HLC frame classes, is a useful supplementary reference for this appendix, as is Appendix C, which includes all the cases that have been implemented, written in FIL (the frame instance language that was introduced in Section 5.3). Some excerpts from this case library are included in this appendix, but Appendix C enables the reader to see how *any* case referred to here is implemented.

B.1 'Exactly One' Constraints

Consider the two models in Figures B-1 and B-2. What do they have in common?

```
SCHEMA gender_by_enumeration;
     ENTITY person;
          sex : gender;
     END_ENTITY;

     TYPE gender = ENUMERATION OF (male, female);
     END_TYPE;
END_SCHEMA;
```

Figure B-1. Representing a person's gender by the value of an enumeration type.

```
SCHEMA gender_by_subtyping;
     ENTITY person ABSTRACT SUPERTYPE OF (ONEOF (man,woman));
     END_ENTITY;

     ENTITY man SUBTYPE OF (person);
     END_ENTITY;

     ENTITY woman SUBTYPE OF (person);
     END_ENTITY;
END_SCHEMA;
```

Figure B-2. Representing a person's gender by subtypes.

The two models both represent people as having gender. Both models constrain each instance of `person` to have exactly one of two genders, but they represent this differently. Figure B-1 represents this constraint as a mandatory attribute `sex` of type `gender`, which must take exactly one of the values `male` and `female`. Figure B-2 represents this constraint by declaring that `person` is an abstract supertype, which can only be instantiated to be one of the subtypes `man` and `woman`.

Under the obvious interpretation of the correspondence between the two models, the constraints in Figures B-1 and B-2 are semantically equivalent, in the sense that instances of `gender_by_enumeration::person` which do not violate the constraints on that entity are guaranteed not to violate any constraints on `gender_by_subtyping::person` if they are mapped from `gender_by_enumeration` to `gender_by_subtyping`, and vice versa. A computer could realise this by:

- Extracting from both models an HLC which says that an instance of `person` must have exactly one of two genders (whichever way genders are represented);
- Comparing the HLCs on the two models to see that they correspond and are (in this case, totally) semantically equivalent to each other.

The system described in this thesis only implements the first step, the extraction of HLCs. Comparing HLCs to assess semantic equivalence is a difficult problem which requires knowing the structural and computational mapping between the two models (Brown, Filer and Moosa [11]), i.e. that `gender_by_enumeration::person` corresponds to `gender_by_subtyping::person`, and that a value of `male` for `gender_by_enumeration::person:sex` corresponds to `gender_by_subtyping::man` whereas a value of `female` corresponds to `gender_by_subtyping::woman`. Future research will investigate how to represent such correspondences and hence how to compare HLCs on two models (this is discussed further in Appendix G, Section G.3).

Figures B-3 and B-4 show how the cases for extracting ‘exactly one’ constraints from Figures B-1 and B-2 respectively are represented in FIL.

Because it is a case as opposed to an actual model, Figure B-3 abstracts away from those details of Figure B-1 that are not fundamental to the lesson the case teaches. The frame `explicit_attribute#1` corresponds to the attribute `sex`, and `defined_type#1` to the type `gender`, but the case specifies a name for neither, since it is valid no matter what their names are. This case does not need even to mention the entity `person`. The variable `?list` is used instead of the actual list `["male", "female"]`, since the case is valid for any list of enumeration symbols.

The HLC `exactly_one#1` in Figure B-3 says that `explicit_attribute#1` is constrained to have exactly one of the properties in `?list`, and that the HLC is extracted from `defined_type#1`.

The HLC `exactly_one#1` in Figure B-4 says that `entity_type#1` is constrained to have exactly one of the properties in `?set`, and that the HLC is extracted from `oneof_expression#1`.

```
INSTANCE case_frame#1;
     problem = explicit_attribute#1;
     solution = exactly_one#1;
END_INSTANCE;

(****************)
(* Problem part *)
(****************)

INSTANCE explicit_attribute#1;
     tipe = defined_type#1;
     is_optional = false;
END_INSTANCE;

INSTANCE defined_type#1;
     tipe = enumeration_type(?list);
END_INSTANCE;

(*****************)
(* Solution part *)
(*****************)

INSTANCE exactly_one#1;
     choice_set = ?list;
     extracted_from = {defined_type#1};
     constrainee = explicit_attribute#1;
END_INSTANCE;
```

Figure B-3. A case for extracting 'exactly one' HLCs from models like Figure B-1.

There are even more ways that an 'exactly one' constraint can be realised in EXPRESS, such as as a select type, as illustrated in Figure B-5 (a case for which has been implemented - see Appendix C, Section C.4.2), or as a collection of boolean flags constrained by a domain rule such that exactly one of them must be true and all the others false, as illustrated in Figure B-6 (no case for which has been implemented). The important point is that a particular HLC can be realised in EXPRESS in several different ways, and those ways can be encoded as cases which enable that HLC to be recognised in similar fragments of models that are encountered in the future.

```
INSTANCE case_frame#1;
     problem = entity_type#1;
     solution = exactly_one#1;
END_INSTANCE;

(****************)
(* Problem part *)
(****************)

INSTANCE entity_type#1;
     is_abstract = true;
     supertype_of = oneof_expression#1;
END_INSTANCE;

INSTANCE oneof_expression#1;
     expressions = ?set;
END_INSTANCE;

(*****************)
(* Solution part *)
(*****************)

INSTANCE exactly_one#1;
     choice_set = ?set;
     extracted_from = {oneof_expression#1};
     constrainee = entity_type#1;
END_INSTANCE;
```

Figure B-4. A case for extracting ‘exactly one’ HLCs from models like Figure B-2.

```
SCHEMA gender_by_select;
     ENTITY person;
          sex : gender;
     END_ENTITY;

     TYPE gender = SELECT (manhood, womanhood);
     END_TYPE;

     ENTITY manhood;
     END_ENTITY;

     ENTITY womanhood;
     END_ENTITY;
END_SCHEMA;
```

Figure B-5. ‘Exactly one’ can be realised as a select type.

```
SCHEMA gender_by_boolean_flags;
     ENTITY person;
          is_male : BOOLEAN;
          is_female : BOOLEAN;
     WHERE
          (is_male XOR is_female);
     END_ENTITY;
END_SCHEMA;
```

Figure B-6. ‘Exactly one’ can be realised as constrained boolean flags.

An interesting variant of the ‘exactly one’ HLC is ‘at most one’, which could arise in Figures B-1 and B-5 if the `sex` attribute were optional, or in Figure B-2 if the `person` supertype were not abstract. The constraint in Figure B-6 would be `NOT(is_male AND is_female)`. It would then be possible for a person to have no sex at all - it is not clear why that would be desirable, but the point here is that the ‘at most one’ HLC is a *semantically weaker* variant of ‘exactly one’.

The ‘at most one’ HLC is weaker than ‘exactly one’ in the sense that:

- Mapping from a model with an ‘exactly one’ constraint to a model with an ‘at most one’ constraint could not violate the ‘at most one’ constraint, because ‘exactly one’ logically implies ‘at most one’;
- However, mapping from a model with an ‘at most one’ constraint to a model with an ‘exactly one’ constraint could violate the ‘exactly one’ constraint in the case that the instance being mapped has *none* of the properties in the choice set of the ‘at most one’ constraint. In order to be able to detect this state of affairs prior to carrying out the mapping, it is necessary to transform the constraint on the source model to be ‘at least one’ as well as ‘at most one’, and to check ‘at least one’ on each instance before it is mapped.[1]

---

1. Assuming that it does conform to the source model and is therefore already ‘at most one’ by construction.

The above paragraph demonstrates that a constraint on one model might be *partially equivalent* to a constraint on another model (rather than it must either be totally equivalent or not equivalent at all). For example, ‘at most one’ is partially equivalent to ‘exactly one’. Future work on comparing and transferring constraints for schema-to-schema mapping will include recognising this partial equivalence and transferring only as much of the constraint as is necessary to check against the source in order to ensure that no instances which violate the constraints on the target will be mapped. ‘Exactly one’ is equivalent to ‘at most one’ and ‘at least one’ in conjunction, so only ‘at least one’ needs to be transferred to the source model.

This sort of reasoning will require a theory of *semantic strength* to determine whether one constraint implies another (i.e. whether the first constraint is violated whenever the second constraint is violated). According to Section 9.3 of Pratt [54], one formula of the propositional calculus is logically stronger than another formula if the former implies the latter but not vice versa. This idea of logical strength can surely be generalised to constraints. The challenge will be to develop a decision procedure for determining whether one constraint implies another.

In summary, three cases have been implemented for extracting ‘exactly one’:

- from a mandatory enumeration type attribute (Appendix C, Section C.4.1);
- from a mandatory select type attribute (Appendix C, Section C.4.2);
- from an abstract supertype (Appendix C, Section C.4.3).

Similarly, three cases have been implemented for extracting ‘at most one’:

- from an optional enumeration type attribute (Appendix C, Section C.5.1);
- from an optional select type attribute (Appendix C, Section C.5.2);
- from a non-abstract supertype (Appendix C, Section C.5.3).

B.2 Bound and Range Constraints

Extracting bound constraints from relational order operations, and extracting a range constraint from a pair of bounds (one lower, the other upper), are used as a running example throughout Chapter 6, so there is little detail that has to be added here.

The idea of the ‘bound’ HLC is to capture the abstract notion of the value of a variable being constrained to fall on one side of a fixed numeric limit (above the limit if it is a lower bound, below if it is an upper bound). The ‘range’ HLC captures the common idiom of a value being constrained to fall between two bounds.

There are various different ways to express a bound using the operators ‘<’, ‘<=’, ‘>=’ and ‘>’, some of which are semantically equivalent, others not so. The current version of the constraint-understanding system allows bounds to vary along the following four boolean-valued dimensions:

- Whether the bound is *lower* or *upper*;
- Whether the bound is *inclusive* or *exclusive* (inclusive bounds are written using ‘<=’ or ‘>=’, whereas exclusive bounds are written using ‘<’ or ‘>’);
- Whether the bound is *discrete* or *continuous* (it is discrete if the variable is of type `INTEGER`, and continuous if the variable is of type `REAL`);
- Which way round the bound is written (e.g. “`a > b`” or “`b < a`” - these are semantically equivalent).

Since these four dimensions can vary orthogonally, there are sixteen variants of the bound-extracting case in the case library. Figure B-7 shows these sixteen cases (in a simple free-text form rather than FIL, for the sake of perspicuity).

There is also an infinite number of variants of the case for extracting bounds, according to the value of `?limit`, but they are all encoded implicitly by variables (see Section 6.3).

Problem: `?variable : INTEGER; WHERE (?variable >= ?limit);`
Solution: Inclusive discrete lower bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?limit <= ?variable);`
Solution: Inclusive discrete lower bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?variable > ?limit);`
Solution: Exclusive discrete lower bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?limit < ?variable);`
Solution: Exclusive discrete lower bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?variable <= ?limit);`
Solution: Inclusive discrete upper bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?limit >= ?variable);`
Solution: Inclusive discrete upper bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?variable < ?limit);`
Solution: Exclusive discrete upper bound of `?limit` on `?variable`.

Problem: `?variable : INTEGER; WHERE (?limit > ?variable);`
Solution: Exclusive discrete upper bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?variable >= ?limit);`
Solution: Inclusive continuous lower bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?limit <= ?variable);`
Solution: Inclusive continuous lower bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?variable > ?limit);`
Solution: Exclusive continuous lower bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?limit < ?variable);`
Solution: Exclusive continuous lower bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?variable <= ?limit);`
Solution: Inclusive continuous upper bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?limit >= ?variable);`
Solution: Inclusive continuous upper bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?variable < ?limit);`
Solution: Exclusive continuous upper bound of `?limit` on `?variable`.

Problem: `?variable : REAL; WHERE (?limit > ?variable);`
Solution: Exclusive continuous upper bound of `?limit` on `?variable`.

Figure B-7. The sixteen cases for extracting bound constraints.

The obvious way to write bounds that are semantically equivalent is to write them the other way round (as shown above). There are other ways, however, such as "`?variable > ?limit`" being equivalent to "`?variable >= (?limit + 1)`", just when the variable is discrete (i.e. an `INTEGER`). However, the current version of FIL is not expressive enough to handle this case in its full generality, because slot fillers can only be literals or variables, not arbitrary expressions (in this case, adding one to the limit). FIL could be extended to handle arbitrary expressions in the future.

The main limitation of the current set of cases for extracting bounds is that they can only match relational operations where one of the operands is an explicit attribute and the other is a numeric literal. The `variable` slot is filled with the attribute and the `limit` slot is filled with the literal. The difficulty in other circumstances is that it is unclear which operand is the variable being constrained and which is the limit. For example, the expression `(father.age > age)` in Figure 1-2 states that one's father must be older than oneself. This could be interpreted as a lower bound on the father's age (where the limit is one's own age), or as an upper bound on one's own age (where the limit is the father's age), both being true in this case. Where it is sensible to interpret such a relational operation as a bound at all, the interpretation of which operand is the variable and which is the limit could be done in a case-based manner (this is a 'fuzzier' problem than any the current constraint-understanding system can handle).

Another interesting example of semantic equivalence arises from the duality between *numeric* range constraints and *symbolic* range constraints. Consider Figures B-8 and B-9. Figure B-8 expresses a numeric range constraint, and Figure B-9 expresses the same constraint symbolically.

```
TYPE t  = INTEGER;
WHERE
     {3 <= SELF <= 6};
END_TYPE;
```

Figure B-8. A numeric range constraint.

```
TYPE t = ENUMERATION OF (three, four, five, six);
END_TYPE;
```

Figure B-9. A symbolic range constraint.

Under the obvious interpretation of the mapping between type `t` in Figure B-8 and type `t` in Figure B-9, the constraints they imply are semantically equivalent. This could be seen by extracting an ‘exactly one’ HLC from Figure B-9, and a ‘range’ HLC from Figure B-8, followed by extracting an ‘exactly one’ from the ‘range’.

There would have to be a case for extracting an ‘exactly one’ from a ‘range’, provided that the range is discrete (so that the choice set for the ‘exactly one’ can be enumerated). This case has not been implemented, because apart from the fact that it is quite obscure, it would again require the use of an arbitrary expression (as opposed to just literals or variables) to enumerate the choice set for the ‘exactly one’ from the limits of the ‘range’. For example, it would have to compute that the choice set for the ‘exactly one’ extracted from the range `{3 <= SELF <= 6}` is `{3,4,5,6}`.

Nevertheless, it is an instructive example of semantic equivalence - constraints can be expressed in all sorts of different ways. Essentially, all constraints can be thought of as restricting the valid values of a type (or the valid instances of an entity) to a subset of a larger set which is not so constrained. In the above example, these subsets are represented explicitly, which can be a useful basis for comparing the semantics of constraints.

B.3 Equations

Another form of semantic equivalence that can be captured in HLCs concerns equations. An *equation* is a set of expressions (usually two expressions) that are constrained to evaluate to the same value. Consider Figures B-10 and B-11, which are alternative models of rectangles.

```
SCHEMA rectangle_area_domain_rule;
     ENTITY rectangle;
          width : REAL;
          height : REAL;
          area : REAL;
     WHERE
          area = width * height;
     END_ENTITY;
END_SCHEMA;
```

Figure B-10. A model of rectangles in which area is constrained by a domain rule.

```
SCHEMA rectangle_area_derived_attribute;
     ENTITY rectangle;
          width : REAL;
          height : REAL;
     DERIVE
          area : REAL := width * height;
     END_ENTITY;
END_SCHEMA;
```

Figure B-11. A model of rectangles in which area is constrained by a derived attribute.

In both these models, the area of a rectangle is constrained to be equal to the product of the width and height of the rectangle. Although `area` is stored explicitly in one model and derived in another, the two models are semantically equivalent in terms of their constraints. That is to say, it is not possible to violate the constraints on these models by mapping valid instances from one model to the other. The common abstraction which characterises this semantic equivalence is the equation *`area`* = *`width`* * *`height`*.

Figure B-12 shows a case for extracting an 'equation' HLC from (the expression in) a domain rule. It illustrates that an equation is represented as a set of expressions which are constrained to be equal ({`area`, (`width * height`)}, in the case of Figure B-10). The case for extracting an equation from a derived attribute is very similar in the obvious ways to this one, and is not shown here (see Appendix C, Section C.8.2).

```
INSTANCE case_frame#1;
     problem = binary_operation#1;
     solution = equation#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE binary_operation#1;
     operator = equals;
     first_expression = ?x;
     second_expression = ?y;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE equation#1;
     equals = {?x, ?y};
     extracted_from = {binary_operation#1};
END_INSTANCE;
```

Figure B-12. A case for extracting an 'equation' HLC from a domain rule.

Equations are a useful building block for an even higher level of semantic equivalence. There are essentially three ways[2] of writing the equation *`area`* = *`width`* * *`height`*:

- `area = width * height`
- `width = area / height`
- `height = area / width`

2. Ignoring trivialities such as which way round the equation is written, the commutativity of multiplication, and frivolous oddities such as adding zero or multiplying by one.

An abstraction which captures the equivalence between these three equations is a *multiplier*. A 'multiplier' HLC consists of two multiplicands and a product, but abstracts away from whether that relationship is expressed as a multiplication or a division.

Two cases suffice to extract 'multiplier' HLCs from the above three forms of equation. The case given in Figure B-13 covers multiplication (*). The case given in Figure B-14 covers division (/), and suffices for both forms of equation that cover division because multiplication is commutative - it does not matter which multiplicand is used as the quotient and which is used as the divisor.

```
INSTANCE case_frame#1;
     problem = equation#1;
     solution = multiplier#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
     equals = {binary_operation#1, ?z};
END_INSTANCE;

INSTANCE binary_operation#1;
     operator = multiply;
     first_expression = ?x;
     second_expression = ?y;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE multiplier#1;
     multiplicands = {?x, ?y};
     product = ?z;
     extracted_from = {equation#1};
END_INSTANCE;
```

Figure B-13. A case for extracting a 'multiplier' HLC from an equation of the form $z = x * y$.

```
INSTANCE case_frame#1;
     problem = equation#1;
     solution = multiplier#1;
END_INSTANCE;

(***********)
(* Problem *)
(***********)

INSTANCE equation#1;
     equals = {binary_operation#1, ?x};
END_INSTANCE;

INSTANCE binary_operation#1;
     operator = divide;
     first_expression = ?z;
     second_expression = ?y;
END_INSTANCE;

(************)
(* Solution *)
(************)

INSTANCE multiplier#1;
     multiplicands = {?x, ?y};
     product = ?z;
     extracted_from = {equation#1};
END_INSTANCE;
```

Figure B-14. A case for extracting a 'multiplier' HLC from an equation of the form $x = z / y$.

Another HLC, 'adder', has also been implemented. It covers equations of the following forms:

- `z = x + y`
- `x = z − y`
- `y = z − x`

The cases for extracting 'adder' HLCs are obviously analogous to the cases for extracting 'multiplier' HLCs, and are not shown here (see Appendix C, Section C.9). In fact, the fact that they are analogous means that the constraint-understanding system should be able to derive either from the other using adaptation.[3]

The cases for extracting 'adder' and 'multiplier' HLCs from equations could be extended to handle more general equations of the forms $s = a_1 + ... + a_n$ and $p = m_1 * ... * m_n$. The `adder` and `multiplier` frame classes already enable this in that the slot types of `addends` and `multiplicands` are bags of expressions (see Appendix A, Section A.2). This generalisation could be tackled using adaptation in the future.

The constraint-understanding system could also be extended to deal with other arithmetical operations and their inverses, extending its potential ability to see the semantic equivalence between equations which are just rearrangements of one another.

The idea behind the 'adder' and 'multiplier' HLCs comes from the field of *constraint propagation* (see, for example, Chapter 15 of Forbus and de Kleer [28]), where the role of constraints is not so much to enforce integrity (as the constraints on information models do), but rather to specify declaratively the way information can flow when the constraints are so tight that it is possible to compute unspecified information from given information. For example, any one of `area`, `width` and `height` could be calculated if values for the other two were known. In constraint propagation, constraints (such as adders and multipliers) can be connected together to form *constraint networks*, through which information may flow in any direction appropriate to the subset of values which is given as input.

3. The current version of the constraint-understanding system only supports simple reinstantiation, which is not powerful enough to adapt multipliers to be adders - that would require knowing:
   - that the '`*`' and '`/`' operators are associated with the 'multiplier' HLC whereas the '+' and '−' operators are associated with the 'adder' HLC;
   - that '`/`' is to '`*`' as '−' is to '+';
   - that addition and multiplication are appropriate operations from which to extract HLCs like adders and multipliers, whereas many other operations are not.

B.4 Existence Constraints

The constraint that an attribute is mandatory (i.e. that it must have a value other than null) can be expressed either by the absence of the keyword `OPTIONAL`, as in Figure B-15, or by a domain rule, as in Figure B-16.

```
ENTITY e;
     a : t;
END_ENTITY;
```

Figure B-15. Encoding that an attribute is mandatory by omitting the `OPTIONAL` keyword.

```
ENTITY e;
     a : OPTIONAL t;
WHERE
     EXISTS(a);
END_ENTITY;
```

Figure B-16. Encoding that an attribute is mandatory by a domain rule.

Figure B-15 illustrates the usual way to express mandatory attributes, but it may have to be done as in Figure B-16 in some circumstances:

- There may be complex existence constraints on an attribute which need to be expressed as a logical condition containing occurrences of the `EXISTS` predicate, as demonstrated in Figure B-17;
- When transferring constraints from a target model to a source model for schema-to-schema mapping (see Section 1.3.2), an attribute may be mandatory in the target but optional in the source. It would be better to express the constraint that the attribute on the transformed source model must have a non-null value using a domain rule (as in Figure B-16) than to delete the keyword `OPTIONAL` on that attribute, because the latter solution involves making a structural change to the source model whereas the former merely adds a rule. Making structural changes to the source model is ill-advised, because it may affect the way instances are implemented rather than just adding rules to be checked by

the mapping program. It would require another structural mapping - from the source model to the transformed source model!

```
SCHEMA people_age_model;
     ENTITY person;
          name : STRING;
          age : OPTIONAL INTEGER;
          date_of_birth : OPTIONAL date;
     WHERE
          some_age_information_must_exist:
               EXISTS(age) OR EXISTS(date_of_birth);
     END_ENTITY;

     ENTITY date;
          day : INTEGER(2);
          month : INTEGER(2);
          year : INTEGER(4);
     END_ENTITY;
END_SCHEMA;
```

Figure B-17. An example of a more complicated existence constraint.

An HLC ‘non null’ (which represents the abstract semantics that an attribute must have a value other than null) has been implemented, together with two cases to extract it from EXPRESS fragments like Figures B-15 and B-16. These cases are not shown here (see Appendix C, Section C.6).

As an aside, note the types of the attributes in the entity `date` in Figure B-17. These imply range constraints on those attributes, e.g. `INTEGER(2)`[4] implies `{-99 <= day <= 99}`.[5] The system could have a case to extract a ‘range’ HLC from an attribute whose type is an integer thus constrained.[6]

4. In EXPRESS, `INTEGER(n)` means an *n*-digit integer (not to be confused with `REAL(n)`, which means a real number with a precision of *n* digits).
5. A ‘better’ information model would constrain days and months more tightly than this, of course, but that is beside the point of this example.

Just as the existence constraint implied by a mandatory explicit attribute can be expressed explicitly in a rule, so could the existence constraint implied by an inverse attribute (see Section 1.2.1.2). For example, Figure B-18 shows how the existence constraint implied by the inverse attribute `supervisee` in Figure 5-9 could be expressed as an explicit domain rule. There are two motivations for having an HLC to capture this semantic equivalence (although this has not actually been implemented):

- The argument given earlier in this section about not making structural changes to the transformed source model when transferring constraints for schema-to-schema mapping also applies to inverse attributes;
- Some real models, such as the EDIF (Electronic Design Interchange Format) information model (Lau [40]), actually use this style of modelling instead of inverse attributes. It would be very useful, for example when mapping to a different version of the model which uses inverse attributes, to be able to see this semantic equivalence.

```
SCHEMA student_supervisor_relationship;
     ENTITY student;
          name : STRING;
          supervisor : staff_member;
     END_ENTITY;

     ENTITY staff_member;
          name : STRING;
     WHERE
          supervises_one_student:
               SIZEOF(USEDIN(SELF,
                    'student_supervisor_relationship.student.supervisor'
                     )       ) = 1;
     END_ENTITY;
END_SCHEMA;
```

Figure B-18. The existence constraint implied by an inverse attribute, expressed as an explicit rule.

6. Were it not for the fact that the constraint-understanding system uses an existing EXPRESS parser, ExJava (see Appendix E, Section E.3), which throws away the necessary information, e.g. it reads `INTEGER(2)` as though it were just `INTEGER`. It also rewrites interval expressions as, for example, `((-99 <= day) AND (day <= 99))` instead of `{-99 <= day <= 99}`, which is why there are no cases for extracting range constraints directly from interval expressions.

The domain rule `supervises_one_student` says that, for each instance `SELF` of `staff_member`, there must be exactly one instance of the entity `student` whose value for the attribute `supervisor` is `SELF`.[7] This is the same constraint that is implied by the inverse attribute `supervisee` in Figure 5-9.

As another example of semantic equivalence (again, not implemented), imagine that a record company allows an artist to release an album with a minimum of six tracks and a maximum of twenty. This could be modelled as in Figure B-19, or alternatively as in Figure B-20.

```
SCHEMA records_model_a;
     ENTITY album;
          artist : STRING;
          title : STRING;
          tracks : LIST [6 : 20] OF track;
     END_ENTITY;

     ENTITY track;
          name : STRING;
     END_ENTITY;
END_SCHEMA;
```

Figure B-19. A record company's model of albums and their tracks.

The motivation for Figure B-20 might be that it is a source model for a record company that places no constraints on the number of tracks on an album, which has been transformed to respect the constraints on the model in Figure B-19 when that is the target model for schema-to-schema mapping (see Section 1.3.2). Modifying the list's cardinality bounds directly when transforming the source model would be ill-advised because, once again, it is a structural change to the model.

---

7. The function `USEDIN` takes an entity instance and a role (the name of an attribute), and returns the set of entity instances for which the value of the given attribute is the given entity instance. The function `SIZEOF` returns the number of elements in a given aggregate. "`SIZEOF(...) = 1`" is a common idiom for constraining an aggregate to have exactly one element with a specified property.

```
SCHEMA records_model_b;
     ENTITY album;
          artist : STRING;
          title : STRING;
          tracks : LIST [0 : ?] OF track;
     WHERE
          (SIZEOF(tracks) >= 6) AND (SIZEOF(tracks) <= 20);
     END_ENTITY;

     ENTITY track;
          name : STRING;
     END_ENTITY;
END_SCHEMA;
```

Figure B-20. An alternative model of albums and their tracks.

The way to see the semantic equivalence between the models in Figures B-19 and B-20 is, of course, to extract 'bound' HLCs. A lower bound of six and an upper bound of twenty on `SIZEOF(tracks)` could be extracted almost as discussed in Section B.2 (except that the cases for extracting bounds currently allow the variable constrained by a bound only to be an explicit attribute, not an arbitrary expression such as `SIZEOF(tracks)`). Extra cases would be required to extract the equivalent 'bound' HLCs from the cardinality bounds of aggregations.

Finally, it is often difficult, when writing an information model, to choose between writing...

```
attribute : SET [0 : ?] OF element_type;
```

...and...

```
attribute : OPTIONAL SET [1 : ?] OF element_type;
```

...to constrain `attribute` to have zero or more elements.[8] Two cases have been written to extract an HLC called 'zero or more'[9] from attributes of the forms given above (see

8. This may seem not so much a constraint as a non-constraint, but it is a stronger constraint than that implied by `attribute : OPTIONAL SET [0 : ?] OF element_type`, where `attribute` could have a value of null or the empty set, which may have different meanings.

Appendix C, Section C.7).

B.5 The Conjunction HLC Revisited

Section 6.2 introduced a special kind of HLC, ‘conjunction’. Its main role was to provide the ‘glue’ on which to hang HLCs that need to be treated in conjunction (e.g. a ‘range’ HLC is extracted from the *conjunction* of two bounds). However, the ‘conjunction’ HLC also has a *semantic* role, like all the other HLCs. Consider Figures B-21 and B-22.

```
SCHEMA explicit_conjunction;
     ENTITY living_person;
          age : INTEGER;
     WHERE
          (age >= 0) AND (age < 130);
     END_ENTITY;
END_SCHEMA;
```

Figure B-21. An explicit conjunction.

```
SCHEMA implicit_conjunction;
     ENTITY living_person;
          age : INTEGER;
     WHERE
          age >= 0;
          age < 130;
     END_ENTITY;
END_SCHEMA;
```

Figure B-22. An implicit conjunction.

The constraint expressed by the domain rule in Figure B-21 is semantically equivalent to the constraint expressed by the two domain rules in Figure B-22:

- A ‘conjunction’ HLC can be extracted from Figure B-21 using the case shown in Figure 6-8. The ‘conjunction’ HLC is extracted from the `AND` operation, and the HLCs

9. For want of a better name - there is no guarantee that the upper bound on the set’s cardinality will sanction more than zero elements!

extracted from the two subexpressions (namely two bounds, and then a range is extracted from those) are hung on the 'conjunction' HLC as conjuncts (as in Figure 6-4).

- As explained in Section 6.2, the constraint-understanding system automatically extracts a 'conjunction' HLC from every entity in the input EXPRESS model. Semantically, it is as though all the constraints on an entity were conjuncts of a conjunction - the 'conjunction' HLC makes this explicit. All top-level constraints of the entity (i.e. those not extracted from subexpressions in domain rules etc.) are automatically set to be conjuncts of this 'conjunction' HLC.

Another possibility (though not implemented) would be to extract conjunctions from `AND` expressions in `SUPERTYPE` expressions. The system could also be extended with similar HLCs for other logical expressions (e.g. a 'disjunction' HLC for `OR` expressions), which could similarly be used to glue together HLCs extracted from subexpressions.

B.6 Summary

This appendix has presented a series of HLCs, and cases to extract them from models. It is not supposed to provide complete coverage of all the HLCs that anyone would possibly want to extract from EXPRESS models, but it illustrates how HLCs and cases can be invented by analysing examples of semantically equivalent models. The mechanics of adding new HLCs and cases to the system are straightforward:

- Adding a new kind of HLC entails writing an entity definition for it in the HLC frame class model (Appendix A, Section A.2);
- Adding cases to extract an HLC involves writing them as FIL files and inserting them in the case library (Appendix C);

It is not necessary to modify and recompile the source code of the constraint-understanding system when adding new HLCs and cases.[10]

All the HLCs and cases that have been implemented are discussed in this appendix (plus some that have not been implemented). A few examples of cases were used in this appendix to illustrate the ideas without drowning the reader in excessive detail; the complete case library can be found in Appendix C.

To summarise, the following HLCs have been implemented (which form the 'ontology' mentioned at the beginning of this appendix):

- `conjunction` (see Sections 6.2, B.5 and C.1);
- `bound` (see Sections 6.2, B.2 and C.2);
- `range` (see Sections 6.2, B.2 and C.3);
- `exactly_one` (see Sections B.1 and C.4);
- `at_most_one` (see Sections B.1 and C.5);
- `non_null` (see Sections B.4 and C.6);
- `zero_or_more` (see Sections B.4 and C.7);
- `equation` (see Sections B.3 and C.8);
- `adder` (see Sections B.3 and C.9);
- `multiplier` (see Sections B.3 and C.10);

---

10. Unless, of course, testing them reveals a bug, or the system needs to be extended to handle new features, or the HLC being added is one of those extremely rare ones that the program has to be told about at compile-time (the only example of this in the current constraint-understanding system is `conjunction`, as explained in Sections 6.2 and 6.3).