NISTIR 5927

On the Translation of Kif/Frame Ontologies to EXPRESS

Peter R. Wilson
Catholic Univeristy of America

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

NIST

NISTIR 5927

On the Translation of Kif/Frame Ontologies to EXPRESS

Peter R. Wilson
Catholic Univeristy of America

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

August 1996

On the Translation of Kif/Frame Ontologies to EXPRESS

Peter R. Wilson
Catholic University of America*
pwilson@cme.nist.gov

August 1996

Abstract

This report describes the translation of three groups of ontologies specified using the Frame variant of the Knowledge Interchange Format (KIF) language into information models specified using the *EXPRESS* information modeling language, as defined in International Standard ISO 10303-11:1994. This work was undertaken to better understand the relationship between KIF/Frame and *EXPRESS*. From the work done to date the capabilities of KIF/Frame and *EXPRESS* appear to be broadly similar but they differ in detail.

Contents

*This work was performed as a Guest Researcher at the National Institute of Standards and Technology.

1 Introduction

The Knowledge Interchange Format (KIF [Gin91], [GF92]) is being proposed as an ANSI Standard [X395]. There is another language, namely *EXPRESS*, which has recently become an ISO International Standard [SW94], [ISO94]. Both these languages are intended to enable the precise and formal modeling of information. An exercise was undertaken to translate some of the KIF-defined ontologies into *EXPRESS* in order to try and determine whether this was possible. In particular, several ontologies were translated from versions described using the automatically generated generic frame representations.

This document reports on the general results from this exercise. The Ontologies used were specified in a generic Frame language that had been generated automatically from the original KIF specification. The ontologies were obtained in December 1995 from:
`http://www-ksl.stanford.edu/knowledge-sharing/ontologies`.
The *EXPRESS* renditions of the selected ontologies are given elsewhere [Wil96a], [Wil96b], [Wil96d]. It is assumed that the reader has some knowledge of both KIF (and the corresponding frame language) and the *EXPRESS* language.

Section 2 provides an overview of the processes used to translate from the KIF/Frame ontologies into *EXPRESS* models. Some observations resulting from this are given in Section 3. The next section (4) discusses the basic elements of Frame and *EXPRESS* and notes their similarities and differences. Conclusions are given in Section 5. Finally, an *EXPRESS* model is given in Appendix A as a challenge for KIF/Frame experts to translate into an ontology.

2 An overview of the translation process

No language exactly maps one-for-one into another language. This, though, does not necessarily imply the languages are not equally expressive. There are two forms of translation, one that I call *transliteration* and the other called *idiomatic*. In the transliteration case the result looks or sounds strange to a native speaker. Let us take the French sentence *Il pleut comme une vache!* as an example. A transliteration of this into English is: 'It is raining like a cow!'. On the other hand, an idiomatic translation will probably result in 'It is raining cats and dogs!'. In turn, a transliteration of this back into French would puzzle most French speakers.

Essentially, the goal of a good translation is to end with an idiomatic rather than a transliterated result.

Briefly, the translation from the frame language to *EXPRESS* was done in the following manner.

2.1 Stage 1

This was a fairly mechanical process.

1. For each frame theory, create a similarly named *EXPRESS* Schema.

2. For each frame construct of the form `define-frame NAME`, create an *EXPRESS* Entity called `Name`. The name translation scheme used was:

   - Change `NAME` to `Name`.
   - Change `FIRST-SECOND` to `FirstSecond`.
   - Change `FIRST.SECOND` to `FirstAndSEcond`.

   Where the frame has an `own-slot` of kind `SUBCLASS-OF NAME`, or similar, make the *EXPRESS* entity a `SUBTYPE OF (Name)`. That is, map the frame `SUBCLASS-OF` into an *EXPRESS* subtype.

   Where the frame has an `own-slot` of kind `SUBCLASS-PARTITION (setof NAME1 NAME2)`, or similar, make the *EXPRESS* entity corresponding to the frame a `SUPERTYPE OF (ONEOF(Name1, Name2))`.

   Where a frame has a `template-slot`, make this an attribute of the *EXPRESS* entity.

   Incorporate the simpler frame axioms into the *EXPRESS* model.

   Some examples of these rules are:

```
(define-frame DOCUMENT
 :theory bibliographic-data
 :own-slots (
  (INSTANCE-OF class)
  (SUBCLASS-OF biblio-thing)
  (SUBCLASS-PARTITION
    (setof book proceedings ....))
  (DOCUMENTATION
    "A document is ..."))
 :template-slots (
  (DOC.TITLE
    (Slot-Cardinality 1))))
```

```
ENTITY Document
  SUBTYPE OF (BiblioThing)
  SUPERTYPE OF (ONEOF(Book,
                      Proceedings,
                      ...));
END_ENTITY;
```

```
(define-frame DOC.TITLE
 :theory bibliographic-data
 :own-slots (
  (ARITY 2)
  (RANGE title)
  (DOMAIN document)
  (INSTANCE-OF function)
```

```
ENTITY DocAndTitle;
  TheDoc   : Document;
  TheTitle : Title;
END_ENTITY;
```

```
  (SUBCLASS-OF biblio-thing)
  (SUBCLASS-PARTITION
    (setof book proceedings ....))
  (DOCUMENTATION
    "The title of a document ...")))
```

```
(define-frame DOCTORAL-THESIS              ENTITY DoctoralThesis
 :theory bibliographic-data                  SUBTYPE OF (Thesis);
 :own-slots (                              END_ENTITY;
  (INSTANCE-OF class)
  (SUBCLASS-OF thesis)
  (DOCUMENTATION
    "PhD thesis document.")))
```

3. Remove the DOCUMENTATION from each frame and place it as an *EXPRESS* descriptive comment.

4. For each *EXPRESS* entity that effectively corresponds to an *EXPRESS* primitive type (such as Integer), replace it by an *EXPRESS* Type instead.

   For example:

```
(define-frame DAY-NUMBER                   TYPE DayNumber = INTEGER;
 :theory bibliographic-data                WHERE
 :own-slots (                                limited : {0 < SELF <= 31};
  (INSTANCE-OF class)                      END_TYPE;
  (SUBCLASS-OF integer)
  (DOCUMENTATION
    "integer representing day of month."))
 :axioms (
         ( <=> (day-number ?day-of-month)
          (and (integer ?day-of-month)
               (=< 0 ?day-of-month)
               (=< ?day-of-month 31)))
         (inherited-slot-value day-number =< 31)))
```

5. If possible, convert any frame classes that are exhaustively enumerated into an *EXPRESS* Enumeration Type.

   For example

```
(define-frame MONTH-NAME                   TYPE MonthName = ENUMERATION OF
 :theory bibliographic-data                                 (January,
 :own-slots (                                                February,
  (INSTANCE-OF class)                                        ...);
```

```
  (ALL-INSTANCES                              END_TYPE;
    (setof january february ...))
  (DOCUMENTATION
    "The months of the year ...")))

(define-frame JANUARY
 :theory bibliographic-data
 :own-slots (
  (INSTANCE-OF month-name)))

...
```

This process resulted in transliterated *EXPRESS* models that captured most of the intent of the ontologies translated. However, the models were not complete at this point and requires further work; principally formulating the frame axioms in terms of *EXPRESS* constructs and constraint language.

2.2 Stage 2

This stage is intended to complete the Stage 1 *EXPRESS* model.

1. Some axioms are of the form shown in the frame definition below, which has been taken from a bibliographic ontology:

```
(define-frame DOC.SERIES-TITLE
  :theory bibliographic-data
  :own-slots (
   (ARITY 2)
   (RANGE title)
   (INSTANCE OF function)
   (DOCUMENTATION ))
  :axioms (
          (=> (doc.series-title ?doc ?title)
           (or (book ?doc) (proceedings ?doc)))))
```

   This specification is basically saying that only a Book or a Proceedings (which are two among several kinds (subtypes) of Document) can have a Series-Title. These kinds of axioms were translated into WHERE rules specifying the required type restrictions. For example, the above frame could be translated into:

```
ENTITY DocAndSeriesTitle;
  SeriesTitle : Title;
  Doc         : Document;
WHERE
```

```
  wr1 : ('BIBLIO.BOOK' IN TYPEOF(Doc)) OR
        ('BIBLIO.PROCEEDINGS' IN TYPEOF(Doc));
END_ENTITY;
```

2. Change binary relations to (optional and/or list) attributes.

   The model resulting from Stage 1 has many entities that look like binary relationships. That is, there are entities Ent1, Ent2 and BinRel where BinRel is like:

```
ENTITY BinRel;
  at1 : Ent1;
  at2 : Ent2;
END_ENTITY;
```

   For a more idiomatic translation these BinRel are candidates for replacement by attributes of appropriate cardinality in either Ent1 or Ent2 like:

```
ENTITY Ent1;
  -- previous attributes
  RelatedTo : Ent2;
END_ENTITY;
```

   Taking the specification above of DocAndSeriesTitle as a concrete example, this could be reconfigured as:

```
ENTITY Document
  -- other stuff
  DocSeriesTitle : OPTIONAL Title;
WHERE
  wr1 : NOT EXISTS(DocSeriesTitle) XOR
        (EXISTS(DocSeriesTitle) AND
          (('BIBLIO.BOOK' IN TYPEOF(SELF)) XOR
           ('BIBLIO.PROCEEDINGS' IN TYPEOF(SELF))));
END_ENTITY;
```

3. In many cases value-related axioms in the frame model could be translated into derived attributes in the *EXPRESS* model.

   For example:

```
(define-frame INHERITS-TITLE-FROM-DOCUMENT
 :theory bibliographic-data
 :own-slots (
  (INSTANCE-OF class)
```

```
  (SUBCLASS-OF publication-reference))
 :axioms (
   (<=> (inherits-title-from-document ?ref)
    (and (publication-reference ?ref)
      (same-values ?ref ref.title (compose doc.title ref.document))))
   (same-slot-values inherits-title-from-document ref.title
    (compose doc.title ref.document))))
```

can be translated into

```
ENTITY InheritsTitleFRomDocument
  SUBTYPE OF (PublicationReference);
DERIVE
  SELF\BibReference.RefTitle : Title := Ref.Document.DocTitle;
END_ENTITY;
```

2.3 Stage 3

This stage is basically tidying up the *EXPRESS* model and presenting it in an idiomatic manner.

1. Add in any elements missing from the Frame model.

   For example, one part of the bibliographic ontology dealt with the referencing of a paper that had been published in a printed proceedings. Data fields were defined for the author, title, and so on of the paper but the means of referencing the proceedings themselves was missing. In all fairness, though, the ontologies were not claimed to be necessarily complete.

2. In some cases binary relations were converted into *EXPRESS* functions called from derived attributes. For instance, consider the following:

```
ENTITY a;
  -- a stuff
END_ENTITY;

ENTITY b;
  -- b stuff
END_ENTITY;

ENTITY ab;
  at1 : a;
  at2 : b;
END_ENTITY;
```

This can be written as:

```
ENTITY a;
  -- a stuff
DERIVE
  bs : SET OF b := BinA(SELF);
END_ENTITY;

ENTITY b;
  -- b stuff
END_ENTITY;

ENTITY ab;
  at1 : a;
  at2 : b;
END_ENTITY;

FUNCTION BinA(Arg : a) : SET OF b;
-- Use the USEDIN function on Arg to get to the ab's.
-- RETURN(SET OF b); -- all those b's associated with Arg
END_FUNCTION;
```

This is at first sight not a particlularly useful transformation. However, there were cases in the Frame ontologies that, after translation into *EXPRESS* appeared like:

```
ENTITY a;
  -- a stuff
END_ENTITY;

ENTITY b;
  -- b stuff
END_ENTITY;
```

with a an axiom concerning some relationship between a's and b's, although though was no frame specifying the relationship — this was missing from the ontology. For instance, such an axiom might be that no more than three instances of b could be associated with each instance of a. This could be dealt with in the *EXPRESS* model by introducing an appropriate function (call).

```
ENTITY a;
  -- a stuff
WHERE
  limit : SIZEOF(BinA(SELF)) <= 3;
END_ENTITY;

ENTITY b;
```

```
  -- b stuff
END_ENTITY;

FUNCTION BinA(Arg : a) : SET OF b;
-- RETURN(SET OF b); -- all those b's associated with Arg
END_FUNCTION;
```

3. Remove synonyms from the *EXPRESS* model.

   Again taking the bibliographic ontology as an example, in some places the publisher was referred to as an 'institution', in others as an 'organization' and in yet other places as a 'publisher'. All these were collapsed in the single identifier 'publisher'.

4. Revise the *EXPRESS* model to convert, as far as possible, logical (procedural) constraints into the model structure.

   As an example, recall the constraints in the Document entity, i.e.,

```
ENTITY Document
  -- other stuff
  DocSeriesTitle : OPTIONAL Title;
WHERE
  wr1 : NOT EXISTS(DocSeriesTitle) XOR
        (EXISTS(DocSeriesTitle) AND
          (('BIBLIO.BOOK' IN TYPEOF(SELF)) XOR
           ('BIBLIO.PROCEEDINGS' IN TYPEOF(SELF))));
END_ENTITY;
```

   Checking the validity of an instance of Document requires the evaluation of the logical statement in the WHERE clause. This can instead be modeled structurally as:

```
ENTITY Document
  -- other stuff
END_ENTITY;

ENTITY Book
  SUBTYPE OF (Document);
-- other attributes
  DocSeriesTitle : OPTIONAL Title;
END_ENTITY;

ENTITY Proceedings
  SUBTYPE OF (Document);
-- other attributes
  DocSeriesTitle : OPTIONAL Title;
END_ENTITY;
```

3 Observations

Three groups of ontologies were firstly transliterated into *EXPRESS* models and then massaged into idiomatic *EXPRESS*. In some cases a group required the conversion of more than one ontology. The grouping and ontologies were:

1. `bibliographic-data` — the scope is bibliographic references, such as might appear at the end of a technical or scholarly document [Wil96a].
2. `constraints` — a general ontology describing the specification of constraints in the form of logical sentences [Wil96b].

   Some other ontologies were extensions of this.

   - `component-assemblies` — a general ontology about assemblies of things, including sub-assemblies and connections.
   - `components-with-constraints` — an ontology about components (from `component-assemblies` that have constraints (from `constraints`).
   - `mechanical-components` — an ontology specializing `component-assemblies` to assemblies of mechanical things.
3. `frame-ontology` — an ontology describing the generic frame language [Wil96d].

   This called on two other ontologies:

   - `kif-relations` — an ontology describing relationships among objects.
   - `kif-sets` — an ontology delaing with set theory.

These were all obtained in December 1995 from
`http://www-ksl.stanford.edu/knowledge-sharing/ontologies`.

Table 1 gives a comparison between the sizes of the different models in both the Frame and *EXPRESS* renditions. The letters at the top of the columns indicate the different ontologies.

Generally speaking, the *EXPRESS* models are similar or smaller in terms of the number of definitions with respect to the Frame models.

3.1 Theory vs. Schema

The Frame concept of `Theory` maps into the *EXPRESS* concept of `Schema`. Syntactically, a `Theory` and a `Schema` are treated differently. *EXPRESS* uses an *embedded* syntax so that the contents of a `Schema` are syntactically embedded between the constructs `BEGIN_SCHEMA name;` and `END_SCHEMA;`. The Frame language uses a *referential* syntax so that a construct identifies which `Theory` it is a member of.

Table 1: Statistical model comparisons

Frame					*EXPRESS*				
	B	CCA	F	Tot		B	CCA	F	Tot
Theories	1	4	3	8	Schemas	1	6	4	11
Misc		1	6	7	Types	6	4	3	13
Classes	66	19	40	125	Entities	69	37	117	223
Functions	40	6	27	73	Functions	3	10	1	14
Relations	19	15	41	75	Rules		1		1
Instances	12			12					
TOTAL	137	45	117	299	TOTAL	79	58	125	262

Key: B — bibliographic, CCA — constraints, components and assemblies, F — Frame.

Both embedded and referential syntaxes have their positive and negative sides, paticularly when it comes to the means of extending a model, but this is not the place for a discussion of that point. However, I did find significant problems with the ontologies as presented in that there were things mentioned in one `Theory` that were not specified within that `Theory` (but which were presumably defined in another `Theory`). There was no indication in the Frame representation of where or what these (presumed) `Theories` were. This was particularly evident in the Constraint related Ontologies. The `Schema` construct in *EXPRESS* provides a scoping mechanism and there is a matching capability of formally specifying where specifications utilised from other `Schemas` are to be found.[1]

It thus appears that *EXPRESS* formally requires a model to be complete but I did not find this to be a formal requirement of the Frame specifications. (Neither did I find anything in any of the specific Frame Ontology documents to help in identifying missing specifications or their possible locations. The documentation style used for *EXPRESS* models in ISO 10303 specifically requires that the collection of `Schemas` forming a model be identified in the informal documentation as well as the *EXPRESS* language requirement.)

3.2 Classes, Relations, Entities and such

EXPRESS essentially has three constructs — `Schema`, `Entity` and `Rule`. In this I am including a `Type` as being a kind of `Entity` and a `Function` as one means of specifying a constraint that might appear in an `Entity` or a `Rule`.

The Frame ontologies use more constructs. In the studied ontologies these are typically `Class`, `Function`, `Relation`, `Instance`, and of course `Theory`. These constructs, except for `Theory` essentially have to be mapped into the *EXPRESS* notion of `Entity`. Conceptually this is no problem as *EXPRESS* effectively defines an `Entity` as 'a representation of something of interest'. In practice, there can be some difficulties.

[1] In fact this is a requirement of the language.

The mapping of Class to Entity is reasonably straightforward as I believe that Class and Entity are virtually synonymous.

Relation also maps well to Entity. When *EXPRESS* was being developed there were many discussions on explicitly distinguishing between 'entity' and 'relation' as in the Entity-Relationship modeling paradigm. However the decision was made to just have Entity as the more one looked at 'relationship' the more it looked and behaved like 'entity'. The Frame Relation then can be transliterated to the *EXPRESS* Entity. In the limited translation experience to date most of the Relations were binary relations with no attributes. Whenever one Entity references another Entity, *EXPRESS* automatically considers this to be a 'relationship'. It is rare to see unconstrained and unattributed 'binary relations' in *EXPRESS* models — these are typically modeled by a reference from one or other of the Entities concerned to the other. Thus, in many cases a Frame Relation eventually got translated into an attribute of an *EXPRESS* Entity.

A Frame Function has little in common with an *EXPRESS* Function except for the name and the general definition of a function as a map from a domain to a range. For example, the following could be a typical *EXPRESS* Function used to describe the combinations of days, months and years that denote a valid date.

```
FUNCTION date_is_valid(d : date): BOOLEAN;
(* given a date (day, month, year) check to ensure that the
   day, month and year compose a valid date, taking into
   account leap years. Return TRUE if date is valid, FALSE
   otherwise. *)
(* check year is positive i.e., AD dates only. *)
  IF (d.year <= 0) THEN RETURN (FALSE); END_IF;
(* check day range *)
  IF NOT {1 <= d.day <= 31} THEN RETURN (FALSE); END_IF;
(* check days in months of 30 days or less *)
  CASE d.month OF
    April,
    June,
    September,
    November  : RETURN (d <= 30);
    (* special check for February *)
    February  : RETURN (valid_leap_month(d));
    OTHERWISE : RETURN (TRUE);
  END_CASE;
END_FUNCTION;
```

On the other hand, here is a typical Frame function:

```
(define-frame REF.YEAR
 :theory bibliographic-data
 :own-slots (
  (ARITY 2)
```

```
  (RANGE year-number)
  (DOMAIN reference)
  (INSTANCE-OF function)
  (DOCUMENTATION
    "The year field is a function from a reference to the year in
which the publication was published.")))
```

Functions like this are modeled in *EXPRESS* as `Entity` attributes. For example:

```
ENTITY reference
-- other stuff
  publication_year : year;
END_ENTITY;
```

From studying the Frame ontology it appears as though the notions of `Class` and `Function` are basically two different aspects of the notion `Relation`. A `Relation` has one or more arguments. A `Class` is a unary `Relation` (a `Relation` with only one argument). A `Function` is a `Relation` where the 'value' of the last argument is completely determined by the preceeding arguments.

3.3 Instances

The Frame language has a notion of 'instances'. As examples:

```
(define-frame REF.YEAR
 :theory bibliographic-data
 :own-slots (
  (ARITY 2)
  (RANGE year-number)
  (DOMAIN reference)
  (INSTANCE-OF function)
  (DOCUMENTATION
    "The year field is a function from a reference to the year in
which the publication was published.")))
```

or

```
(define-frame REF.AUTHOR
 :theory bibliographic-data
 :own-slots (
  (ARITY 2)
  (INSTANCE-OF relation)
  (RANGE author-name)
```

```
  (DOMAIN reference)
  (DOCUMENTATION
    "Relation between a reference and the name(s) of the
creator(s) of the publication.")))
```

It took me a long time to feel that I had come to an understanding of what this meant. In fact, the translation of the Frame ontology was explicitly undertaken in order to attempt to solve this.

Eventually I came to the conclusion that the explanation lay in set theory, where a Frame `Relation` effectively defines a set of that 'name', and similarly for `Class` and `Function`. The `INSTANCE-OF` syntax is then a set membership declaration. So, for example,
`INSTANCE-OF relation`
can be read as 'this is a member of the set "relation" '.

EXPRESS also has similar notions, in that `Schema`, `Entity` and so on are sets [Wil96c]. A declaration like
`ENTITY AnEntity;`
specifies that `AnEntity` is both a set named `AnEntity` and is a member of the set `Entity`.

4 Basic elements of Frame and EXPRESS

In broad terms both Frame and *EXPRESS* have three basic semantic elements, although the syntactical representations of these are different.

Frame

1. Theory — the collection of definitions for a particular ontology.
2. Relation — a definition (specification of real world thing of interest).
3. Axiom — a logical expression (or sentence) applied to relation(s).

EXPRESS

1. Schema — the collection of definitions for (a cohesive part of) a particular model.
2. Entity — a definition (specification of a real world thing of interest).
3. Constraint — A logical expression (or statement) applied to entity(ies).

Thus, at a broad level, Frame and *EXPRESS* deal with the same semantic elements; as is often the case, though, the devil is in the details.

Frame makes distinctions between unary relations, which are termed classes, relations where the value one argument is completely defined by the values of the other arguments, which are termed classes, and general n-ary relations which are called relations.

EXPRESS syntactically has entities and types, where a type can be thought of as a very restricted form of an entity. Effectively a Frame class maps to (the name of) an entity or type. A Frame function can typically be mapped to an entity with a derived attribute. General relations map to entities.

As we have already noted, a Frame theory and an *EXPRESS* schema are virtually synonymous, and so are Frame relations and *EXPRESS* entities. The major difference is between Frame axioms and *EXPRESS* constraints.

In Frame, an axiom is a logical sentence involving relations. These sentences may use the typical forms of predicate logic (e.g., 'there exists' or 'for all' and so on). In *EXPRESS* a constraint is also specified by a logical statement, but there is no built in language equivalent to 'for all' and friends. Instead, *EXPRESS* provides a programming language which enables the equivalent to 'for all', etc., to be stated. The programming language includes looping constructs, case statements and so forth. It is therefore often much easier to specify constraints in *EXPRESS* than in Frame, as in the latter case one is restricted to only using a series of logical sentences, whereas in the forming both general programming techinques are available in addition to logical expressions. That is, often the programming constructs provide a short, elegant and understandable means of specifying the variables and their value ranges that are used in the final constraint statements, whereas in general this is much harder to do when purely restricted to logic — not impossible, but readability is likely to suffer with no increase in either precision or formality.

5 Conclusions and further work

There was little difficulty in transliterating Frame ontologies into *EXPRESS* models, and then further mapping these into idiomatic *EXPRESS* code. The transliteration process could be reasonably automated, but not the idiomatic mapping.

Further investigation into the use of instances in the Frame language and their mapping relationship to *EXPRESS* may be required. In the ontologies mapped so far, these have not caused any problems, but it is possible that they might do so in more complex ontologies.

It is unclear to me whether it is possible to map *EXPRESS* models into the Frame language without losing information. In order to shed some light on this problem I suggest that an experienced Frame (or KIF) modeler map the *EXPRESS* model given in the appendix (A) into the Frame language. The original scope statement for this model is given in ISO TR9007 [ISO87]. This ISO report describes requirements for conceptual languages and presents the model represented in a variety of generic modeling language types.

References

[GF92] M. R. Genesereth and R. E. Fikes. *Knowledge Interchange Format, Version 3.0*

Reference Manual. Report Logic-92-1, Computer Science Department, Stanford University, June 1992.

[Gin91] M. L. Ginsberg. Knowledge Interchange Format: The KIF of Death. *AI Magazine*, 12(3):57–63, Fall 1991.

[ISO87] ISO TR9007. *Information processing systems — Concepts and terminology for the conceptual schema and the information base*, 1987.

[ISO94] ISO 10303-11:1994. *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*, 1994.

[SW94] Douglas A. Schenck and Peter R. Wilson. *Information Modeling the EXPRESS Way.* Oxford University Press (ISBN 0-19-308714-3), 1994.

[Wil96a] Peter R. Wilson. *Bibliographic Ontology — An EXPRESS Rendition of a Frame Model*, February 1996. (Draft).

[Wil96b] Peter R. Wilson. *Component Modeling Ontologies — An EXPRESS Rendition of a Frame Model*, March 1996. (Draft).

[Wil96c] Peter R. Wilson. *EXPRESS and Set Theory*, June 1996. (Draft).

[Wil96d] Peter R. Wilson. *Frame Ontology — An EXPRESS Rendition of a Frame Model*, April 1996. (Draft).

[X395] X3T2 KIF AHG. *Knowledge Interchange Format Reference Manual*, March 1995.

A EXPRESS example model

This Appendix contains a complete and documented *EXPRESS* model together with an *EXPRESS-G* graphical version. The model is documented in a similar manner to the STEP models.

A.1 Scope

The model has to do with the registration of cars and is limited to the scope of interest of the Registration Authority. This Authority exists for the purpose of:

- Knowing who is or was the registered owner of a car at any time from construction to destruction of the car;
- To monitor laws regarding the transfer of ownership of cars;
- To monitor laws regarding the fuel consumption of cars;
- To monitor laws regarding manufacturers of cars.

A.2 Model overview

The model is described using both *EXPRESS* and *EXPRESS-G*. The *EXPRESS* definitions are primary and the *EXPRESS-G* diagrams are to assist in understanding the primary model. If there is any conflict between the *EXPRESS* and *EXPRESS-G*, then the *EXPRESS* takes precedence.

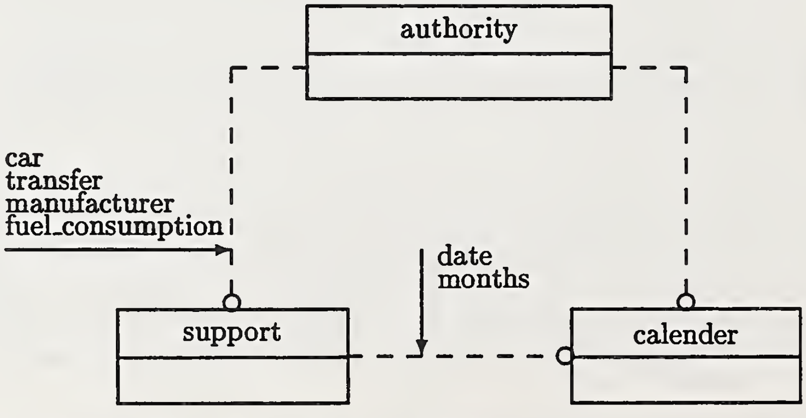

Figure 1: Complete schema-level model for Registration Authority example (Page 1 of 1).

The model consists of three schemas, as shown in figure 1. The schema `authority` is the primary schema. It references items from the two ancilliary schemas, namely `support` and `calendar`. The `support` schema also references items from the `calendar` schema.

A.3 Authority schema

This schema is the primary one in the model and is principally concerned with the main functions of the Registration Authority.

The schema imports definitions from two sources, namely the `support` and the `calendar` schemas.

Figure 2 is an *EXPRESS-G* complete entity-level model for this schema.

EXPRESS specification:

```
*)
SCHEMA authority;
  REFERENCE FROM support (car,
                          transfer,
                          manufacturer,
                          fuel_consumption,
                          mnfg_average_consumption);
  REFERENCE FROM calendar (current_date);
(*
```

A.3.1 Entity definitions

Entity HISTORY

A `history` records the transfers of ownership of a `car` over its lifetime. A `history` must be kept for a certain period after the `car` is destroyed, after which the ownership records may be destroyed.

EXPRESS specification:

```
*)
ENTITY history;
  item : car;
  transfers : LIST [0:?] OF UNIQUE transfer;
DERIVE
  to_be_deleted : BOOLEAN := too_old(SELF);
UNIQUE
  un1 : item;
WHERE
  one_car  : single_car(SELF);
```

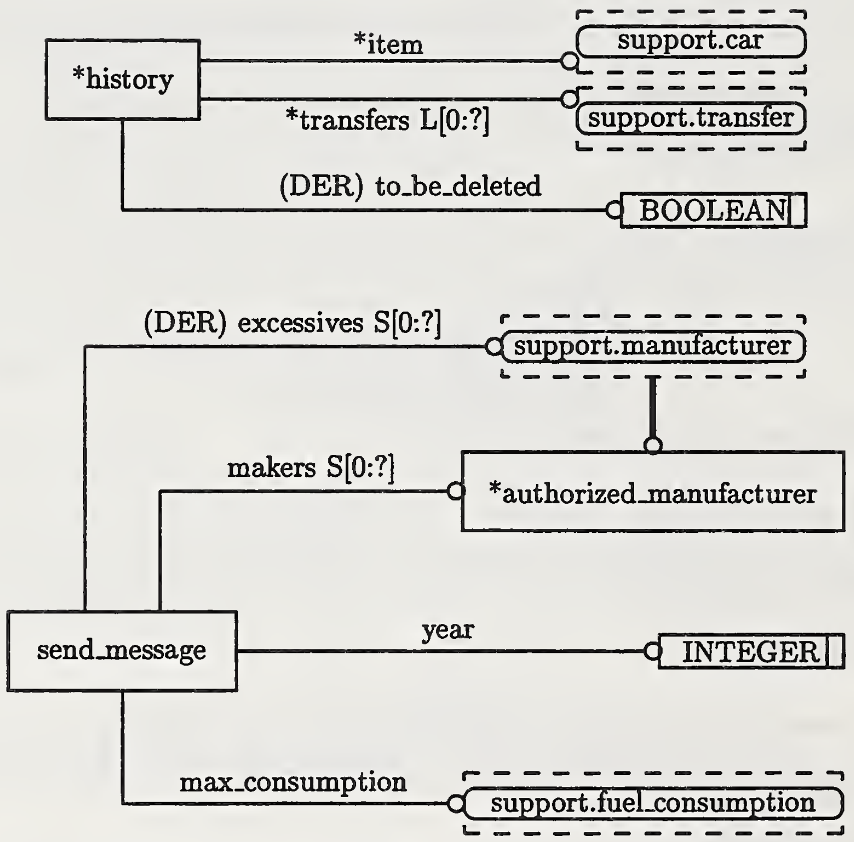

Figure 2: Complete entity-level model of the Authority schema (Page 1 of 1).

```
  ordering : exchange_ok(transfers);
END_ENTITY;
(*
```

Attribute definitions:

item: The car whose ownership history is being tracked.

transfers: The ownership transfer records of the item.

to_be_deleted: A flag which indicates that this history record may be deleted because the item has been destroyed (TRUE), or that the record shall not be deleted (FALSE).

Formal propositions:

un1: The value of item shall be unique across all instances of history.

one_car: Each transfer collected in a history shall be of the same car.

ordering: The list of transfer shall be in increasing historical order.

Entity AUTHORIZED MANUFACTURER

An authorized manufacturer is a manufacturer who has been given permission by the Registration Authority to make cars.

EXPRESS specification:

```
*)
ENTITY authorized_manufacturer
  SUBTYPE OF (manufacturer);
END_ENTITY;
(*
```

Rule MAX NUMBER

No more than five authorized manufacturers are permitted at any one time.

EXPRESS specification:

```
*)
RULE max_number FOR (authorized_manufacturer);
WHERE
  max_of_5 : SIZEOF(authorized_manufacturer) <= 5;
END_RULE;
(*
```

Formal propositions:

max_of_5: The rule is violated if there are more than five authorized manufacturers at any time.

Entity SEND MESSAGE

In January each year the Registration Authority shall send a message to each manufacturer whose cars' average fuel consumption exceeds a certain limit, which may vary from year to year.

EXPRESS specification:

```
*)
ENTITY send_message;
  max_consumption : fuel_consumption;
  year            : INTEGER;
  makers          : SET [0:?] OF authorized_manufacturer;
DERIVE
  excessives : SET [0:?] OF manufacturer := guzzlers(SELF);
END_ENTITY;
(*
```

Attribute definitions:

max_consumption: The legal maximum average fuel consumption.

year: The year for which the max consumption value applies.

makers: The authorized manufacturers operating during the year.

excessives: The manufacturers whose cars exceed the consumption limit.

A.3.2 Function and procedure definitions

Function GUZZLERS

This function returns the set of manufacturers whose cars exceed an average fuel consumption limit.

Argument definitions:

par: An instance of a send message entity.

RESULT: A set of instances of manufacturer whose cars' average fuel consumption is excessive.

EXPRESS specification:

```
*)
FUNCTION guzzlers(par : send_message) : SET OF manufacturer;
LOCAL
  result : SET OF manufacturer := [];
  mnfs   : SET OF manufacturer := par.makers;
  limit  : fuel_consumption := par.max_consumption;
  time   : INTEGER := par.year;
END_LOCAL;
  REPEAT i := 1 TO SIZEOF(mnfs);
    IF (mnfg_average_consumption(mnfs[i],time) > limit) THEN
      result := result + mnfs[i];
    END_IF;
  END_REPEAT;
RETURN(result);
END_FUNCTION;
(*
```

Function TOO OLD

This function calculates whether the car in a history was destroyed more than two years ago.

Argument definitions:

par: An instance of a history.

RESULT: A Boolean value. TRUE if the car in the input history was destroyed two or more years ago; otherwise FALSE.

EXPRESS specification:

```
*)
FUNCTION too_old(par : history) : BOOLEAN;
  (* The function returns TRUE if the input history is
     outdated. That is, if it is of an item that was destroyed
     more than 2 years ago. *)
  IF ('SUPPORT.DESTROYED_CAR' IN par.item) THEN
    IF (current_date.year-par.item.destroyed_on.year >= 2) THEN
      RETURN(TRUE);
    END_IF;
  END_IF;
  RETURN(FALSE);
END_FUNCTION;
(*
```

Function EXCHANGE OK

This function checks whether or not the transfers in a list are ordered.

Argument definitions:

par A list of transfer instances.

RESULT A Boolean value. TRUE if the recipient in the N^{th} transfer is the same as the giver in the $(N+1)^{th}$ transfer.

EXPRESS specification:

```
*)
FUNCTION exchange_ok(par : LIST OF transfer) : BOOLEAN;
  (* returns TRUE if the "to owner" in the N'th transfer of a
     car is the "from owner" in the N+1'th transfer *)
  REPEAT i := 1 TO (SIZEOF(par) - 1);
    IF (par[i].new :<>: par[i+1].prior) THEN
      RETURN (FALSE);
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;
(*
```

Function SINGLE CAR

This function checks whether or not the car in a transfer history is the same car specified in each individual transfer.

Argument definitions:

par: A history instance.

RESULT: A Boolean value. TRUE if the history and all its transfers are of the same car, otherwise FALSE.

EXPRESS specification:

```
*)
FUNCTION single_car(par : history) : BOOLEAN;
  (* returns TRUE if a history is of a single car *)
  REPEAT i := 1 TO SIZEOF(par.transfers);
    IF (par.item :<>: par.transfers[i].item) THEN
      RETURN (FALSE);
```

```
    END_IF;
  END_REPEAT;
  RETURN (TRUE);
END_FUNCTION;
(*
```

A.3.3 Entity classification structure

The following indented listing shows the entity classification structure. Entities in upper case characters are defined in this schema. Entities in lower case characters are defined in other schemas.

```
HISTORY
manufacturer (in schema support)
  AUTHORIZED_MANUFACTURER
SEND_MESSAGE


*)
END_SCHEMA;  -- end of authority schema
(*
```

A.4 Support schema

This schema contains supporting definitions for the primary `authority` schema.

An *EXPRESS-G* model of the contents of this schema is given in figure 3 and in figure 4.

The schema imports definitions from the `calendar` schema.

EXPRESS specification:

```
*)
SCHEMA support;
  REFERENCE FROM calendar (date, months, days_between);
(*
```

A.4.1 Type definitions

Type NAME

The 'name' of something. A human interpretable name which may identify some object, thing or person, etc. For example, `Widget Company, Inc.`.

EXPRESS specification:

```
*)
TYPE name = STRING;
END_TYPE;
(*
```

Type IDENTIFICATION NO

A character string which may be used as the 'identification number' for a particular instance of some object. This is typically a mixture of alphanumeric characters and other symbols. For example, D20-736597WP23.

EXPRESS specification:

```
*)
TYPE identification_no = STRING;
END_TYPE;
(*
```

Type FUEL CONSUMPTION

A measure of the fuel consumption of some powered device.

EXPRESS specification:

```
*)
TYPE fuel_consumption = REAL;
WHERE
  range : {4.0 <= SELF <= 25.0};
END_TYPE;
(*
```

Formal propositions:

range: The value is limited to lie in the range 4 to 25 inclusive.

A.4.2 Entity definitions

Entity TRANSFER

A record of a transfer of a car from one owner to a new owner.

EXPRESS specification:

```
*)
ENTITY transfer;
```

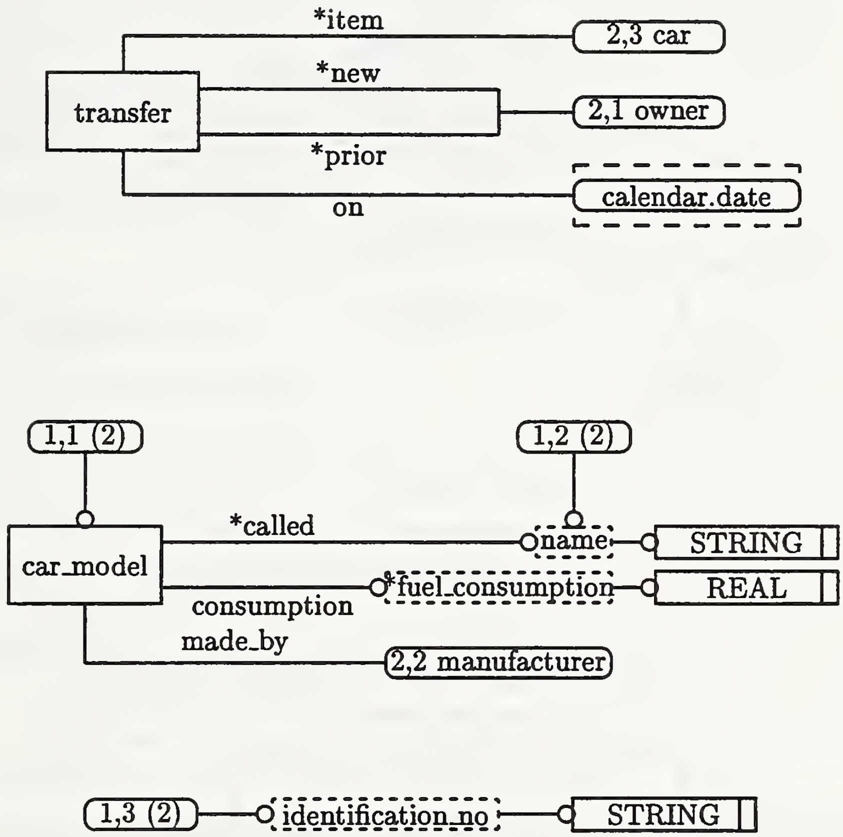

Figure 3: Complete entity-level model of the Support schema (Page 1 of 2).

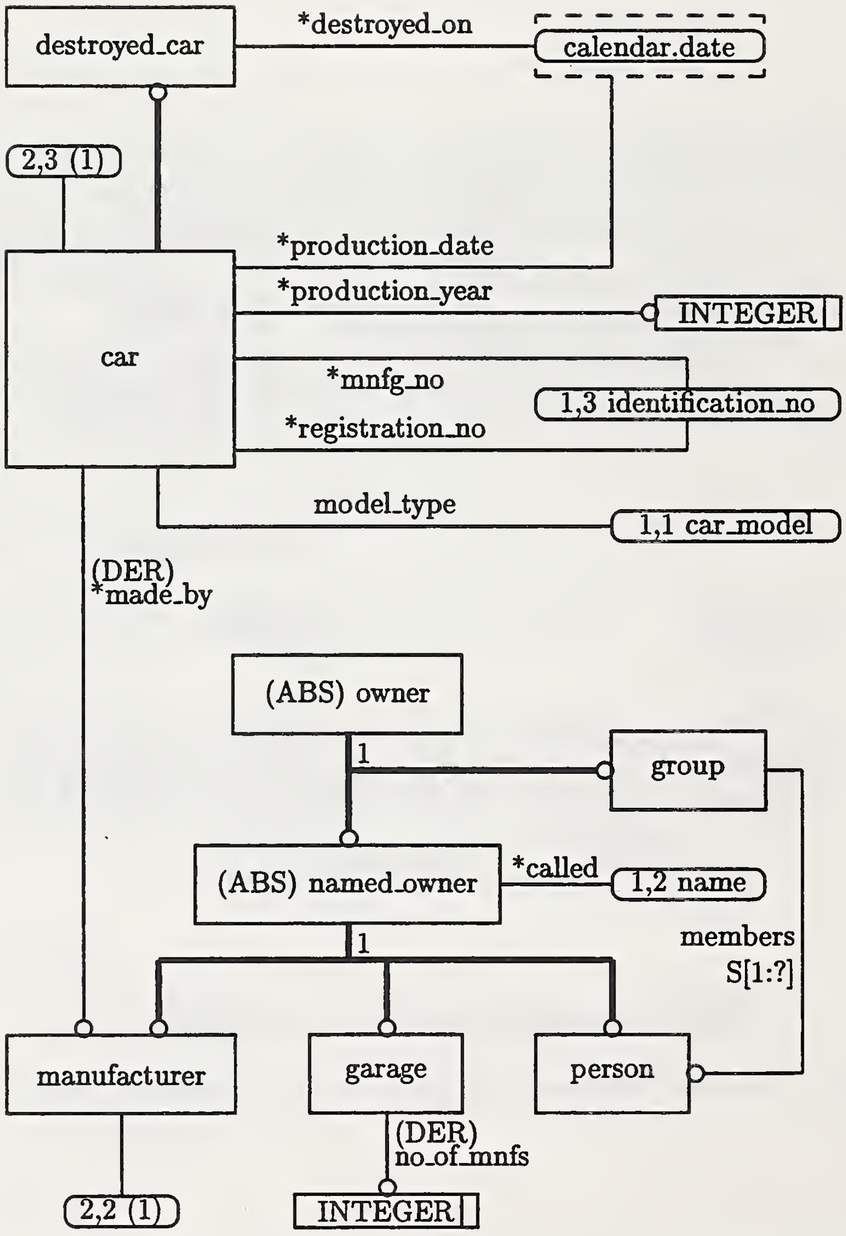

Figure 4: Complete entity-level model of the Support schema (Page 2 of 2).

```
  item  : car;
  prior : owner;
  new   : owner;
  on    : date;
WHERE
  wr1 : NOT ('SUPPORT.MANUFACTURER' IN TYPEOF(new));
  wr2 : (NOT ('SUPPORT.MANUFACTURER' IN TYPEOF(prior))) XOR
     (('SUPPORT.MANUFACTURER' IN TYPEOF(prior)) AND
     ('SUPPORT.GARAGE' IN TYPEOF (new)));
  wr3 : (NOT ('SUPPORT.GARAGE' IN TYPEOF(prior))) XOR
     (('SUPPORT.GARAGE' IN TYPEOF(prior)) AND
     (('SUPPORT.PERSON' IN TYPEOF(new)) XOR
     ('SUPPORT.GROUP' IN TYPEOF(new))));
  wr4 : (NOT ('SUPPORT.DESTROYED_CAR' IN TYPEOF(item)) XOR
     (('SUPPORT.DESTROYED_CAR' IN TYPEOF(item)) AND
      (days_between(on, item\destroyed_car.destroyed_on) > 0)));
END_ENTITY;
(*
```

Attribute definitions:

item: The car being transferred.

prior: The prior owner of the `item`.

new: The new owner of the `item`.

on: The `date` of the `transfer`.

Formal propositions:

wr1: A car cannot be transferred to a `manufacturer`.

wr2: A `manufacturer` can only transfer a car to a garage.

wr3: A garage can only transfer a car to either a person of a group of people.

wr4: A car which has been destroyed cannot be transferred.

Entity CAR

A car.

EXPRESS specification:

```
*)
ENTITY car;
```

```
  model_type      : car_model;
  mnfg_no         : identification_no;
  registration_no : identification_no;
  production_date : date;
  production_year : INTEGER;
DERIVE
  made_by : manufacturer := model_type.made_by;
UNIQUE
  joint  : made_by, mnfg_no;
  single : registration_no;
WHERE
  jan_prod : (production_year = production_date.year) XOR
             ((production_date.month = months.January) AND
              (production_year = production_date.year - 1));
END_ENTITY;
(*
```

Attribute definitions:

model_type: The car model.

mnfg_no: An identification number of the car assigned by the car's manufacturer.

registration_no: An identification number for the car assigned by the Registration Authority.

production_date: The date on which the car was produced.

production_year: The registered year of production of the car.

made_by: The manufacturer of the car.

Formal propositions:

joint: The mnfg no given to a car is unique for the given car manufacturer.

single: Each car is given a unique registration no by the Registration Authority.

jan_prod: The registered production year is the same as the year in which the car was produced, except that cars produced in January may be registered as having been produced in the previous year.

Entity DESTROYED CAR

A car may be destroyed, in which case its date of destruction is recorded.

EXPRESS specification:

```
*)
ENTITY destroyed_car
  SUBTYPE OF (car);
  destroyed_on : date;
WHERE
  dates_ok : days_between(production_date, destroyed_on) >= 0;
END_ENTITY;
(*
```

Attribute definitions:

destroyed_on: The date on which the car was destroyed.

Formal propositions:

dates_ok: A car cannot be destroyed before it has been made.

Entity CAR MODEL

A particular type of car.

EXPRESS specification:

```
*)
ENTITY car_model;
  called      : name;
  made_by     : manufacturer;
  consumption : fuel_consumption;
UNIQUE
  un1 : called;
END_ENTITY;
(*
```

Attribute definitions:

called: The name of the model.

made_by: The manufacturer of the model.

consumption: The average fuel consumption of all cars of this model type.

Formal propositions:

un1: Each car model has a distinct name.

Entity OWNER

An owner of a car. Owners are categorized into named owner and group.
EXPRESS specification:

```
*)
ENTITY owner
  ABSTRACT SUPERTYPE OF (ONEOF(named_owner,
                               group));
END_ENTITY;
(*
```

Entity NAMED OWNER

An owner who has a name. These are categorized into manufacturer, garage and person.
EXPRESS specification:

```
*)
ENTITY named_owner
  ABSTRACT SUPERTYPE OF (ONEOF(manufacturer,
                               garage,
                               person))
  SUBTYPE OF (owner);
  called : name;
UNIQUE
  un1 : called;
END_ENTITY;
(*
```

Attribute definitions:

called: The name of the owner.

Formal propositions:

un1: Owner's names are unique.

Entity MANUFACTURER

A type of named car owner. Manufacturers may also manufacture cars.
EXPRESS specification:

```
*)
ENTITY manufacturer
```

```
  SUBTYPE OF (named_owner);
END_ENTITY;
(*
```

Entity GARAGE

A type of named car owner.

EXPRESS specification:

```
*)
ENTITY garage
  SUBTYPE OF (named_owner);
DERIVE
  no_of_mnfs : INTEGER := dealer_for_mnfs(SELF);
WHERE
  wr1 : {1 <= no_of_mnfs <= 3};
END_ENTITY;
(*
```

Attribute definitions:

no_of_mnfs: The number of different manufacturers of the cars owned by the garage.

Formal propositions:

wr1: At any particular time, a garage shall not own cars made by more than three manufacturers.

Entity PERSON

A type of named car owner.

EXPRESS specification:

```
*)
ENTITY person
  SUBTYPE OF (named_owner);
END_ENTITY;
(*
```

Entity GROUP

A type of car owner consisting of a group of people.

EXPRESS specification:

```
*)
ENTITY group
  SUBTYPE OF (owner);
  members : SET [1:?] OF person;
END_ENTITY;
(*
```

Attribute definitions:

members: The people who form the group.

A.4.3 Function and procedure definitions

Function DEALER FOR MNFS

This function calculates the total number of distinct manufacturers of cars owned by a garage.

Argument definitions:

dealer: An instance of a garage.

RESULT: The number of distinct manufacturers of the cars owned by the garage.

EXPRESS specification:

```
*)
FUNCTION dealer_for_mnfs(dealer : garage) : INTEGER;
  LOCAL
    cars : SET OF car := [];
    transfers : SET OF transfer := [];
    makers : SET OF manufacturer := [];
  END_LOCAL;
  transfers := USEDIN(dealer, 'TRANSFER.NEW');
  REPEAT i := 1 TO SIZEOF(transfers);
    cars := cars + transfers[i].item;
  END_REPEAT;
  transfers := USEDIN(dealer, 'TRANSFER.PRIOR');
  REPEAT i := 1 TO SIZEOF(transfers);
    cars := cars - transfers[i].item;
  END_REPEAT;
  REPEAT i := 1 TO SIZEOF(cars);
    makers := makers + cars[i].model_type.made_by;
  END_REPEAT;
  RETURN (SIZEOF(makers));
END_FUNCTION;
(*
```

Function MNFG AVERAGE CONSUMPTION

This function calculates the average fuel consumption in a given year of all the cars made by a particular manufacturer.

Argument definitions:

mnfg: A manufacturer.

when: An INTEGER representing a particular year.

RESULT: A REAL giving the average fuel consumption of the manufacturer's cars during a particular year.

EXPRESS specification:

```
*)
FUNCTION mnfg_average_consumption(mnfg : manufacturer;
                                  when : INTEGER) : REAL;
  (* returns the average fuel consumption of the given
     manufacturer's cars produced in the given year *)
  LOCAL
    models : SET OF car_model := [];
    cars   : SET OF car := [];
    num    : INTEGER := 0;
    tot    : INTEGER := 0;
    fuel   : REAL := 0;
    result : REAL := 0.0;
  END_LOCAL;
     -- set of mnfg's models
  models := USEDIN(mnfg, 'MODEL.MADE_BY');
  REPEAT i := 1 TO SIZEOF(models);
     -- cars of particular model year
    cars := QUERY(temp <* USEDIN(models[i], 'CAR.MODEL_TYPE')
           | temp.production_year = when);
    num := SIZEOF(cars);
    fuel := fuel + num*models[i].consumption;
    tot := tot + num;
  END_REPEAT;
  IF tot > 0.0 THEN
    result := fuel/tot;
  END_IF;
  RETURN (result);
END_FUNCTION;
(*
```

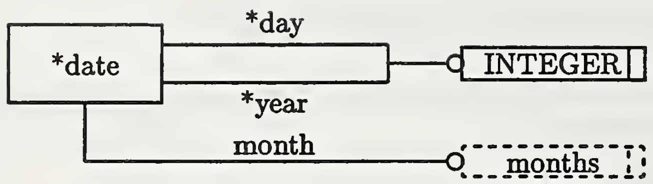

Figure 5: Complete entity-level model of Calendar schema (Page 1 of 1).

A.4.4 Entity classification structure

The following indented listing shows the entity classification structure. Entities in upper case characters are defined in this schema. Entities in lower case characters are defined in other schemas.

```
CAR
    DESTROYED_CAR
CAR_MODEL
OWNER
    GROUP
    NAMED_OWNER
        GARAGE
        MANUFACTURER
        PERSON
TRANSFER

*)
END_SCHEMA;  -- end of support schema
(*
```

A.5 Calendar schema

This schema contains definitions related to dates and other calendrical items.

Figure 5 is an *EXPRESS-G* model showing the contents of this schema.

EXPRESS specification:

```
*)
SCHEMA calendar;
(*
```

A.5.1 Type definitions

Type MONTHS

An enumeration of the months of the year. January is the first month in a year and December is the last month in a year.

EXPRESS specification:

```
*)
TYPE months = ENUMERATION OF
    (January, February, March,
     April,    May,      June,
     July,     August,   September,
     October, November, December);
END_TYPE;
(*
```

A.5.2 Entity definitions

Entity DATE

A date AD in the Gregorian calendar.

EXPRESS specification:

```
*)
ENTITY date;
  day   : INTEGER;
  month : months;
  year  : INTEGER;
WHERE
  days_ok : {1 <= day <= 31};
  year_ok : year > 0;
  date_ok : valid_date(SELF);
END_ENTITY;
(*
```

Attribute definitions:

day: The day of the month.

month: The month of the year

year: The year.

Formal propositions:

days_ok: The day shall be numbered between 1 and 31 inclusive.

year_ok: The year shall be greater than zero.

date_ok: The combination of day, month and year shall form a valid date, taking into account the differing numbers of days in particular months, and also the effect of leap years.

A.5.3 Function and procedure definitions

Function VALID DATE

This function checks a date for valid day, month, year combinations.

Argument definitions:

par: A date.

RESULT: A Boolean. TRUE if the date has a valid day, month, year combination, FALSE otherwise.

EXPRESS specification:

```
*)
FUNCTION valid_date (par : date) : BOOLEAN;
  (* returns FALSE if its input is not a valid date *)
  CASE par.month OF
    April     : RETURN (par.day <= 30);
    June      : RETURN (par.day <= 30);
    September : RETURN (par.day <= 30);
    November  : RETURN (par.day <= 30);
    February  : IF (leap_year(par.year)) THEN
                  RETURN (par.day <= 29);
                ELSE
                  RETURN (par.day <= 28);
                END_IF;
    OTHERWISE : RETURN (TRUE);
  END_CASE;
END_FUNCTION;
(*
```

Function LEAP YEAR

This function checks whether a given integer could represent a leap year.

Argument definitions:

year: An INTEGER.

RESULT: A Boolean. TRUE if year is a leap year, otherwise FALSE.

EXPRESS specification:

```
*)
FUNCTION leap_year(year : INTEGER) : BOOLEAN;
  (* returns TRUE if its input is a leap year *)
  IF ((((year MOD 4) = 0) AND ((year MOD 100) <> 0)) OR
      ((year MOD 400) = 0)) THEN
    RETURN (TRUE);
  ELSE
    RETURN (FALSE);
  END_IF;
END_FUNCTION;
(*
```

Function CURRENT DATE

This function returns the current date.

Argument definitions:

RESULT: The current date.

EXPRESS specification:

```
*)
FUNCTION current_date : date;
  (* This function returns the date when it is called.
     Typically, it will be implemented via a system provided
     procedure within the information base *)
END_FUNCTION;
(*
```

Function DAYS BETWEEN

This function returns the number of days between any two dates.

Argument definitions:

d1: A date.

d2: A date.

RESULT: An Integer. The number of days between the two input dates. If d1 is earlier than d2 a positive integer is returned; if d1 is later than d2 a negative integer is returned; otherwise zero is returned.

EXPRESS specification:

```
*)
FUNCTION days_between(d1, d2 : date) : INTEGER;
  (* returns the number of days between two input dates. If d1
     is earlier than d2, a positive number is returned. *)
END_FUNCTION;
(*
```

A.5.4 Entity classification structure

The following indented listing shows the entity classification structure. Entities in upper case characters are defined in this schema. Entities in lower case characters are defined in other schemas.

```
DATE
```

```
*)
END_SCHEMA; -- end of calendar schema
(*
```