Conceptual Structures for Modeling in CIM

Michel Wermelinger[1] and Alex Bejan[2]

[1] Dep. de Informática, Univ. Nova de Lisboa
2825 Monte da Caparica, PORTUGAL
E-mail: mw@fct.unl.pt
[2] IBM Corporation, 33HA/971
Neighborhood Rd, Kingston NY 12401, USA
E-mail: alge@kgnvmy.vnet.ibm.com

Abstract. The International Standards Organization (ISO) will release in 1993 the first version of the STEP standard, which is dedicated to the exchange of product model data, and is seen as the basis of the next generation of enterprise information modeling tools. Almost in the same time frame ANSI will release the Information Resource Dictionary System (IRDS) Conceptual Schema standard [1], which recommends the conceptual graphs (CGs) or other representation languages based on logic to be used for enterprise information modeling and integration. In this paper we develop the foundations for the utilization of conceptual structures (CS) in combination with *EXPRESS* and STEP Application Protocols in the field of Computer Integrated Manufacturing (CIM). The most important result described here is a mapping of *EXPRESS* into CGs. Around it we develop the architecture of a system able to analyze and translate some of the semantics of information models. Our overall strategy consists of representing the semantics of the language, including the informal meanings represented in the *EXPRESS* manual in plain English, in a systematic way in CS, and then use this block of knowledge, that *can* be processed by a machine, for the increasingly automatic analysis, translation and integration of enterprise information models. The work here described is one of the components of a prototype of a model management system under development at IBM, Kingston NY, coordinated by the CIM Architecture group.
Keywords: Computer Integrated Manufacturing, PDES/STEP, model management system, *EXPRESS* language, modular knowledge bases

1 Introduction

In the modern enterprise, engineers and other information modeling experts use a wide range of information models to manage the enterprise's static, dynamic and high-level aspects. These models may be very small or very complex. In the process of creating or manipulating information models the modelers need to compare their representations with previous work in the same or related areas. Also, especially when dealing with complex models, they need analytical tools to help them better understand the not-so-obvious characteristics of the models. A variety of information modeling technologies and tools are used to define and operate on the information models. But any of these technologies captures only a fraction of the meanings of the enterprise's products and processes. The rest of the semantics remain in such informal representations as comments in natural language, explanatory diagrams, or even in the engineers' unspoken minds. Knowledge engineers manage usually to understand each other's models due to their common background. The only disadvantage is that the task of manually analyzing a complex model, or comparing two models is very laborious, error-prone, and time consuming. Computer programs are not of much help either, because they have access only to a very limited amount of semantics of the enterprise. This is why, even the best current computer based model analyzers, which can be characterized as syntactic analyzers, are very inefficient and user unfriendly. The automation of information model analysis for enterprise models requires the machine to have access to significantly more knowledge about the enterprise than is currently included in the formal models. In this paper we do some of the required mappings for using conceptual structures in enterprise model management as a normative language able to support operations on models written in *EXPRESS*.

The *EXPRESS* language is intended as "a formal information requirements specification language" [4]. Among its goals are: to be parsable by computers as well as humans, to enable partitioning of the domain of discourse, to focus on the design of entities, and to avoid implementation views.

EXPRESS is emerging as an international standard language for enterprise modeling. Not only is it used more and more by manufacturing industries around the world to represent their businesses, but also it started to be used by economic units from the financial, mining, medical and other areas.

In this process, the semantic scope of the language is wontedly challenged by tentative representations of larger portions of the enterprise many of a kind not originally anticipated. Moreover, seduced by its large acceptability, enterprise knowledge engineers are trying to translate their models in *EXPRESS*, thus entitling it with a *de facto* normative property. These uses put a lot of pressure on the broadness of *EXPRESS*, asking for more expressive power.

However, *EXPRESS* cannot be extended beyond certain limits without loosing some of its desirable properties, such as ease of representing industrial designs and concepts, and its appeal to the enterprise information modelers. Also to be taken into account is that any modification to the language is not likely to take effect in the next few years.

Thus, it looks reasonable to provide another knowledge representation means as a normative language for operations on models of enterprises. Conceptual Graphs (CGs) are a good candidate for the task because they have the needed semantic power to represent any modeling language; also the possibility of translating CGs in structured English will provide the additional service of automatically producing comments, where semantics are not translatable into *EXPRESS*. The latter feature will significantly increase the understandability of the *EXPRESS* model.

EXPRESS has a formal, explicit syntax written as a context free grammar

(BNF). Its rules describe how to create well-formed expressions in *EXPRESS* using a certain vocabulary. Its semantics is described in the *EXPRESS* Manual draft, as well as other publications that deal with the intended use of the language. We would like to make this semantics explicit to the system, by rewriting it in Conceptual Structures.

In the language definition document [4], in addition to the syntactic definition of the language, the meaning of each construct, e.g. entity, is defined in plain English: [3] "An entity declaration creates an entity data type and declares an identifier to refer to it. An entity data type represents a class of objects which have common properties. [. . .]

```
entity_decl = entity_head entity_body END_ENTITY ';'
entity_head = ENTITY entity_id  [subexpr]  ';'
entity_body = {explicit_att}  [derive_clause]
              [inverse_clause]
              [unique_clause] [where_clause].
```

"

It appears from here that even the simplest fact about an entity, i.e. that it represents a class of objects which have common properties is not captured by the syntax. That is, the knowledge engineer modeling the enterprise always keeps in mind that an *EXPRESS* entity actually represents a class of objects that have common properties. The computer that holds the information model is never aware of this fact.

The difference between what is represented in the formal grammar (syntax of *EXPRESS*), and what is in plain English is simply that only the former can be processed by machine. Conceptual Graphs, though, can represent both the formal grammar and the English text in uniform constructs, and can be processed by machine entirely. With the current technology some other knowledge about the models is never made known to the machine, which limits the model analysis capabilities. Examples may include the habit of certain users to write names using the underscore, particular ways of representing relationships, the use of types, and many others. This is why it's important that all these pieces of knowledge be explicitly represented in Conceptual Graphs and processed thereafter as a uniform knowledge block, to confer meaning to the enterprise models in cause.

Another problem for the users of STEP is that there is no formal theory behind (the semantics of) *EXPRESS*. This is very understandable, as there is no formal semantics, and building a formal theory in a natural language is quite difficult. Thus, for one thing, representing the *EXPRESS* semantics in CGs would be a good step in the direction of formalizing them.

The representation of the syntax of *EXPRESS* as Conceptual Structures would be the first challenge. Even more effort will be required to rewrite its semantics in Conceptual Structures, and then to prove its consistency (find at least one true interpretation of this model).

[3] All text between quotation marks is taken verbatim from [4].

The result of writing all the syntax and large amounts of the semantics of *EXPRESS* in the same notation will be to provide a more complete, explicit model for this language. This will thereafter prove valuable for *EXPRESS* model analysis, translation to other modeling languages, unification and integration of models.

However, if we think about operations on CIM information models such as model analysis, translation, join, etc. this is only the first step. We need, in addition, a higher-order ontology (meta-model) that will help us represent the knowledge about *EXPRESS* and other languages in a systematic way. The best candidate for the job is the Semantic Unification Meta-Model (SUMM), also an emerging ISO standard [3]. SUMM uses a modal logic approach, which we will replace with CS, in order to ensure uniformity of all of the meta-knowledge.

It should be noted that using CS as the meta-language of our meta-model is different from using them as the target language of a translation from *EXPRESS*. It just happens that the two languages are the same.

With the CS and *EXPRESS* completely denoted in a common language, we can start the mapping of the expressions of the two languages in the appropriate way. *EXPRESS* syntax will always be possible to be translated CS, but only certain conceptual graphs to *EXPRESS*. However, as stated above, any CS can be translated to structured English, so we can avoid an absolute loss of semantics in moving from CS to *EXPRESS*.

More precisely, the translation from *EXPRESS* to CS consists of replacing the *EXPRESS* syntax with CS syntax, and appending all the formal semantics of an *EXPRESS* model to the new CS representation of the model. The new representation of the model is in this way independent from the knowledge of *EXPRESS* by the users, and it preserves the initial semantics. This is another way of saying that the semantics of the *EXPRESS* model are explicit in the CS format. In the opposite direction, if a model in CS contains at least a certain subset of the explicit semantics of any *EXPRESS* model (subset that we still have to define), it is guaranteed to be translatable to *EXPRESS*. In any model translatable to *EXPRESS* it is possible to exist some expressions that are not supported by *EXPRESS*. Those will be transformed into structured (natural) language (e.g. English). The translation from CS to *EXPRESS* involves checking that all the required *EXPRESS* semantics is explicitly stated in the model, then replacing the CS syntax with *EXPRESS* syntax for the translatable expressions, throwing away the expressions representing the explicit semantics of *EXPRESS*, and translating the rest of the expressions to structured language.

If only *EXPRESS* models are considered model analysis and syntactic checking can be done in the way described above. If more than one model is involved, comparative model analysis, and model unification and integration can also be performed.

More languages can be included in the picture, as well. In that case we can talk about a source language (or languages), and one or more target languages. In those translations the meta-model defines the expressions in each source language that can be translated to the target language. Translation grammars will be built around these rules, similar to Definite Clause Translation Grammars or Definite Feature Grammars. They will define the series of CS operations that will be applied to the CS representation of the model in the source language in order to transform it in the target language.

To be more explicit, we deal with a translation in 3 steps:

1. a syntactic translation of a model in a source language to CGs + addition of explicit semantics specific to the source language;
2. a transformation of the CG representation of the model so that the semantics is preserved, but the syntax is modified to that of the target language;
3. a syntactic translation of CGs to the target language + discarding the explicit semantics of the source language + a translation of non (target language) translatable CGs to structured language (English).

The remainder of this paper is organized as follows: Sec. 2 presents an extension to the CS theory, and introduces the notation. Sections 3 to 6 describe the mapping of *EXPRESS* to CS (a more complete account is given in [5]): Sec. 4 deals with expressions, Sec. 5 with data types, and Sec. 6 handles the *EXPRESS* schema construct. Finally, we present in the conclusions some of the advantages of our work for related fields.

2 Conceptual Structures Revisited

This section points out some notational conventions and introduces a new notion to be added to CS theory: knowledge packet. This will enable us to formally define knowledge bases and to represent *EXPRESS* schemata.

2.1 Modularizing Knowledge Bases

The main building block of knowledge bases in basic CS theory is the canon [2, Assumption 3.4.5]. However, there is no provision for structuring knowledge bases, i.e. there is no artifact corresponding to the notion of module provided by conventional programming languages. The following provides a first step in that direction.

We first start by recalling that a canon is a self-contained knowledge base: all the markers and types used in the canonical basis and in the conformity relation are part of the canon. Quasi-canons are obtained by relaxing one of the constraints.

Definition 1. A *canon* is a tuple $\langle I, T, ::, B\rangle$ where I is a set of individual markers, T is a type hierarchy, :: is a conformity relation relating labels in T with markers in I, and B is a finite set of graphs with all referents either generic or markers in I and all type labels in T.

Definition 2. A *quasi-canon* is a canon $\langle I, T, ::, B\rangle$ where the canonical basis B may use type labels that are not contained in T and markers that are not contained in I.

As I, B, and $::$ are sets (the latter of ordered pairs), the subset relationship as well as union are defined for them. The following definitions are needed in order to have inclusion and union defined for all the four components of a canon:

Definition 3. Two type hierarchies T and T' conflict if there are two *different* types t_1 and t_2 such that $t_1 \leq_T t_2$ and $t_2 \leq_{T'} t_1$.

Definition 4. Given two type hierarchies T_1 and T_2, T_1 is a *sub-hierarchy* of T_2 (written $T_1 \subseteq T_2$) if and only if the following conditions hold:

1. $\forall t \in T_1 : t \in T_2$
2. $\forall t_1, t_2 \in T_1 : t_1 \leq_{T_1} t_2 \Rightarrow t_1 \leq_{T_2} t_2$

Definition 5. The *union* of two type hierarchies T_1 and T_2 is written as $T_1 \cup T_2$ and it is defined as the smallest hierarchy such that $T_1 \subseteq T_1 \cup T_2$ and $T_2 \subseteq T_1 \cup T_2$.

Notice that if two type hierarchies conflict, their union is not defined (because it wouldn't be a proper hierarchy). We will therefore assume that two different canons have no conflicting type hierarchies and no common markers.

Now we can build the union of canons as the union of the four components, and inclusion over canons is defined as inclusion of the four components. The bottom element of the inclusion relation is the same as the neutral element of union, namely the empty canon (denoted by $\emptyset$) which includes only the universal and the absurd types.

Let $\mathcal{K}$ be the set of all knowledge packets (to be defined below) and $\mathcal{C}$ the set of all canons. The following functions exist:

Definition 6. The function $\mathit{available} : \mathcal{K} \rightarrow \mathcal{C}$ returns for each knowledge packet the information contained in it. The function $\mathit{exports} : \mathcal{K} \rightarrow \mathcal{C}$ returns for each knowledge packet the information made visible to the other knowledge packets, whereby $\forall K \in \mathcal{K} : \mathit{exports}(K) \subseteq \mathit{available}(K)$.

Definition 7. An *import function* for a knowledge packet K is a total function $\mathit{imports}_K : \mathcal{K} \rightarrow \mathcal{C}$ such that

1. $\mathit{imports}_K(K) = \emptyset$
2. $\mathit{imports}_K(K') \subseteq \mathit{exports}(K')$

We can now define a knowledge packet as a canon with an import-export interface: it imports part of the information exported by other knowledge packets, it defines its own information (possibly) based on the imported one, and it exports part of all that available information.

Definition 8. A *knowledge packet* K is a tuple $\langle \mathit{Exp}, \mathit{Imp}_K, \mathit{Def} \rangle$ such that Exp is a canon, Def a quasi-canon, and Imp_K an import function satisfying

1. $\mathit{exports}(K) = \mathit{Exp}$
2. $\mathit{available}(K) = \bigcup_{K' \in \mathcal{K}} \mathit{Imp}_K(K') \cup \mathit{Def}$

Definition 9. Knowledge packet K *uses* knowledge packet K' if and only if $imports_K(K') \neq \emptyset$. The relation $\prec \subseteq \mathcal{K} \times \mathcal{K}$ is the transitive closure of the "uses" relation.

Notice that if knowledge packet K defines a type `A` explicitly in terms of type `B` but only exports the type label of `A`, any packet that uses K isn't aware of `B` nor of `A`'s definition. However, the system must know both to properly make inferences and check for canonicity. Therefore the notion of visibility is only relevant for the user, but the system must use all the information that is (indirectly) used by a knowledge packet.

Definition 10. A knowledge packet K is consistent if the set of graphs contained in $\bigcup_{K \prec K'} available(K') \cup available(K)$ is consistent.

Definition 11. A *knowledge base* $\mathcal{KB}$ is a set of consistent knowledge packets such that $\forall K \in \mathcal{KB}\ K \not\prec K$.

2.2 Notation

We will use the linear notation for conceptual graphs as defined in [2], extended with `[TYPE:~ref]` as an abbreviation for `[TYPE] -> (DFFR) -> [TYPE:ref]`. Furthermore, `A < B` means that `A` is a direct subtype of `B`, and `[IF: G1 [THEN: G2]]` is a more readable version of `~[ G1 ~[ G2 ]]` where `G1` and `G2` are sets of graphs.

3 Translating *EXPRESS* to Conceptual Structures

The three main constructs of the *EXPRESS* language are: expressions, types, and schemata. They will be translated in a principled way to dataflow graphs, concept types, and knowledge packets, respectively. Due to space limitations we will mainly concentrate on the most important data types.

In the next sections we will show how the knowledge packet EKP representing the semantics of *EXPRESS* should look like, assuming there is a core knowledge packet CKP with the most basic primitive concepts of CS theory. Having these, every *EXPRESS* model M (a set of schemata) translates to a CS knowledge base $\mathcal{KB}$ containing EKP and CKP with $EKP \prec CKP$. Furthermore, for every schema in M there is a corresponding knowledge packet $K \in \mathcal{KB}$ such that $K \prec EKP$.

3.1 The Core Knowledge Packet

In order to have a well founded translation scheme for *EXPRESS*, we will make several assumptions about CKP we believe are reasonable for any practical CS system. Namely, that it includes primitive types and relations like `ENTITY` and `ATTR/2` (given in [2]) and the boolean connectives between propositions (e.g. XOR/2). Furthermore, we must have WORD < STRING and INTEGER < REAL. This implies that the CS processor is able to parse integer and real values as well as strings in the referent field. Furthermore, we assume that some primitive operations on those types are pre-defined (like comparison and addition). Finally, we will use the Unique Name Assumption for reals, integers and strings, i.e. the same name denotes always the same individual and therefore equal looking concepts are coreferent (and this enables the inference rules to be applied).

4 Expressions

An expression will be translated into a dataflow graph, where operators are actors and the result is given by the final (and single) sink concept. The translation scheme is recursive: if the expression is of the form *operand op operand* then translate first both operands and then make their sink concepts the source concepts of the actor corresponding to the operator.

Several operators are overloaded: for instance, the + sign is used to add numbers and concatenate binaries (which, as seen in Sec. 5.3, can be understood as sequences of bits). We can translate this into the following kind of definition

```
actor +(in: x, y; out: z) is
        [T:*x] -
                -> <IDENT> -> [NUMBER] -> <ADD> -
                        <- [NUMBER] <- <IDENT> <- [T:*y]
                        -> [NUMBER] -> <IDENT> -> [T:*z],
                -> <IDENT> -> [BINARY] -> <CONCAT> -
                        <- [BINARY] <- <IDENT> <- [T:*y]
                        -> [BINARY] -> <IDENT> -> [T:*z].
```

Static type checking can be done in the following way: given canonical graphs for the operators (interpreted as relations, not actors) translate the expression into a graph and see if it is canonical. For the above mentioned operator you might have the canonical graph

```
[ [NUMBER] <- (+) -
        2 <- [NUMBER]
        1 <- [NUMBER]
] -> (OR) -> [
  [BINARY] <- (+) -
        2 <- [BINARY]
        1 <- [BINARY]
].
```

stating that the operands of + must both be numbers or both be binaries, and the result is a number or a binary, respectively. Notice that such a canonical graph represents in essence the signature of an operator.

In *EXPRESS*, the special constant '?' represents an indeterminate value. We will assume that *EKP* contains an individual marker named `undefined` which conforms to any of the data types listed in the next section. The constant '?' will therefore be mapped into `[GENERIC:undefined]`.

The above means that expressions may return *null* (which is also represented by '?') if some operand hasn't any value (e.g. it might be an optional attribute, see Sec. 5.1). To cope with this, the definition of actors must handle the `undefined` marker properly. For example:

```
actor ADD(in x,y; out z) is
        [NUMBER:*z] <- <IDENT> <- [NUMBER:undefined] -
                <- <IDENT> <- [NUMBER:*x]
                <- <IDENT> <- [NUMBER:*y]
        ... ;; rest of definition
```

5 Data Types

EXPRESS provides the following data types:

Simple These are the basic types: INTEGER, REAL, NUMBER, BOOLEAN, LOGICAL, STRING and BINARY.
Aggregation These types represent collections.
Entity These are record-like types.
Defined Defined types are just aliases for other types.
Named Defined types and entity types are called named types because they must have a name.
Select These act like an abstract supertype (see 5.1) for a given collection of named types.
Base Simple types, aggregation types, and named types are collectively called base types because they are the only types allowed for aggregation elements.
Generic This is the set of all types.

As the NUMBER type is an abstract supertype for integer and real values, and as BOOLEAN is a subtype of LOGICAL we will have in *EKP*'s type hierarchy

```
INTEGER < REAL     REAL < NUMBER      NUMBER  < SIMPLE
BOOLEAN < LOGICAL  LOGICAL < SIMPLE   STRING < SIMPLE
BINARY < SIMPLE    E-ENTITY < NAMED   DEFINED < NAMED
NAMED < BASE       AGGREGATE < BASE   SIMPLE < BASE
BASE < GENERIC     SELECT < GENERIC   ENUMERATION < GENERIC
```

We call the entity type `E-ENTITY` because `ENTITY` is already in *CKP*.

5.1 Entity Data Types

Entity types correspond to the classes of object-oriented programming languages. An entity is defined through a set of functional attributes upon which several kinds of constraints may be imposed using domain rules. Entity types may form a hierarchy thereby enabling inheritance of attributes. We will translate entity types into concept types while maintaining the subtype relationship, thereby making use of the inheritance mechanism provided by CS theory.

Attributes. According to *EXPRESS*, attributes are partial functions if the attribute is declared to be optional, otherwise it is a total function. To make these semantics explicit with CS we will use a total functional relation `ATTR/3`[4] and make sure that the default value for optional attributes is `undefined`. The canonical graph in *EKP* is

```
(ATTR) -
    1 <- [GENERIC:@every]
    2 <- [WORD:@every]
    -> [BASE:@1]
```

meaning that any generic type may have a base type as a named attribute. To simplify reading and to use fewer lines, we will write

```
[G] -> (ATTR:name) -> [B]
```

as an *abbreviation*[5] for

```
(ATTR) -
    1 <- [G]
    2 <- [WORD:"name"]
    -> [B].
```

An attribute `a` of type `B` will be denoted as `[E-ENTITY] -> (ATTR:a) -> [B]` and the graphs for all attributes are joined on the `E-ENTITY` concept. For each optional attribute `a` of entity `E` we will additionally have

```
schema for E(x) is [E:*x] -> (ATTR:a) -> [BASE: undefined].
```

There is no need to repeat the attribute's type because it will be automatically obtained when doing the schematic join.

There are several reasons for using a generic attribute relation instead of a different conceptual relation for each attribute. First, notice that an attribute name has no semantic value whatsoever. From the computer's perspective it is just an identifier like any other, only the user will assign any meaning to it. Furthermore, the approach we propose makes it easier to compare models where their creators had the same entity in mind but chose different names for the attributes. By making a preprocessor that generalizes all `[WORD: "name"]`

[4] We assume that relations may have the same name as long as their arity differs. This means that `ATTR/2`, already existing in the core knowledge packet, is a completely different relation.

[5] This means that `(ATTR:name)` should *not* be interpreted as a relation with a referent field.

concepts simply to [WORD], we can compare entities using any basic conceptual graph matching algorithm. A third reason is that the same attribute name can appear in several, non-related entity types. So what would be the semantics (i.e. the canonical graph) of the corresponding conceptual relation? Finally, a more pragmatic reason is the fact that the other approach would lead to a proliferation of relations.

Domain rules. "Domain rules are used to specify conditions that constrain the value of individual attributes or a combination of attributes for every entity instance." Each rule is represented by a logical expression which must not be violated (i.e. made false) by any entity instance in the information base.

Domain rule expressions are therefore translated to dataflow graphs with final output concept `[LOGICAL:~false]`. Each occurrence of an attribute name `a` is translated as `[E:*x] -> (ATTR:a) -> [BASE]`, where `E` is the entity type being characterized and `*x` is the same variable for all attributes in order to show that the rule applies to the same instance.

Supertypes and Subtypes. An entity type is required to declare all its supertypes (if any) and is allowed to specify constraints on the relationships between its subtypes. Additionally, an entity data type may only exist for classification purposes and therefore it is never directly instantiated. In this case it is declared as an abstract supertype, i.e. all its instances are instances of some of its subtypes.

The declaration e `SUBTYPE OF (`t_1`,` t_2`, ...)`, where e is the name of the entity type being defined, gets translated to `type` e`(x) is [`t_1`:*x] [`t_2`:*x] ... [E-ENTITY:*x]` and the graphs for the attributes are joined to the `E-ENTITY` concept. No further constraint is imposed by the declaration e `SUPERTYPE OF (`$expr$`)`, where $expr$ is an expression denoting the relationships between e's subtypes.

On the other hand, e `ABSTRACT SUPERTYPE OF (`$expr$`)` implies one can't have an instance of e without having one of its subtypes. This is expressed by `[IF: [`e`:*x] [THEN:` g_{expr} `]]`, where g_{expr} is the set of graphs obtained by translating the subtype expression $expr$ recursively according to the following rules:

EXPRESS	Conceptual Graphs
ONEOF($expr_1$, $expr_2$, ...)	[g_{expr_1}]->(XOR)->[g_{expr_2}]->(XOR)->...
$expr_1$ AND $expr_2$	g_{expr_1} ; g_{expr_2}
$expr_1$ ANDOR $expr_2$	[g_{expr_1}] -> (OR) -> [g_{expr_2}]
entity type name E	[E:*x]

Notice that for the last case the same variable must be used for the whole expression. An example of this translation scheme is given in the next section.

5.2 Aggregation Data Types

Aggregation types are used to represent collections of elements of some base type. Depending on the specific data type of the aggregation, these collections can have fixed or varying size, and duplicate elements may be allowed or not:

Aggregate	Length	Duplicates
ARRAY	fixed	user defined
BAG	varying	yes
LIST	varying	user defined
SET	varying	no

Indexes are used to access individual elements of an aggregation, and the cardinality of a collection is given by its bounds. The upper bound of varying-sized aggregations may be undefined. If we want to represent the semantics of aggregations in an entity-like fashion we would have:[6]

```
ENTITY aggregate
ABSTRACT SUPERTYPE OF (ONEOF(array, list, bag)));
SUBTYPE OF generic;
    low_index, low_bound: INTEGER;
    high_index, high_bound: OPTIONAL INTEGER;
    base_type:  BASE;
WHERE   high_index >= low_index AND high_bound >= low_bound;
END_ENTITY;

ENTITY array SUBTYPE OF (aggregate);
WHERE   high_bound <> '?';
        high_index = high_bound AND low_index = low_bound;
END_ENTITY;

ENTITY bag SUBTYPE OF (aggregate);
WHERE low_bound >= 0; low_index = 1;
END_ENTITY;

ENTITY set SUBTYPE OF (bag); END_ENTITY;

ENTITY list SUBTYPE OF (aggregate);
WHERE low_bound >= 0; low_index = 1;
END_ENTITY;
```

[6] The following takes other information contained in [4] into account.

Using the translation scheme developed for entities, we will have in the *EKP* for the "aggregate entity"

```
type AGGREGATE(x) is
    [GENERIC:*x] -
        (ATTR:low_index) -> [INTEGER]
        (ATTR:low_bound) -> [INTEGER]
        (ATTR:high_index) -> [INTEGER]
        (ATTR:high_bound) -> [INTEGER].
        (ATTR:base_type) -> [BASE].

schema for AGGREGATE(x) is
    [AGGREGATE:*x] -> (ATTR:high_index) -> [BASE:undefined].
schema for AGGREGATE(x) is
    [AGGREGATE:*x] -> (ATTR:high_bound) -> [BASE:undefined].

[LOGICAL:~false] <- <AND> -
    1 <- [LOGICAL] <- <GE> -
        1 <- [BASE] <- (ATTR:high_index) <- [AGGREGATE:*x]
        2 <- [BASE] <- (ATTR:low_index) <- [AGGREGATE:*x],
    2 <- [LOGICAL] <- <GE> -
        1 <- [BASE] <- (ATTR:high_bound) <- [AGGREGATE:*x]
        2 <- [BASE] <- (ATTR:low_bound) <- [AGGREGATE:*x].

[IF: [AGGREGATE:*x]
[THEN: [ [BAG:*x] ] -> (XOR) -> [ [LIST:*x] ] -
       -> (XOR) -> [ [ARRAY:*x] ]
]].
```

To access the elements of an aggregation we need to have in *EKP*

```
relation ELEM(x, y, z) is
    [AGGREGATE: @every *x] -
        (ATTR:base_type) -> [BASE: @every *z]
        (ATTR:low_index) -> [INTEGER] <- (>=) <-[INTEGER:@1 *y]
        (ATTR:high_index)-> [INTEGER] -> (>=) ->[INTEGER:*y].
```

If an aggregation type `A` doesn't allow duplicate elements (e.g. `A` is a set type) we must add the graph

```
[IF: [A:*x] 1 -> (ELEM) -
        2 <- [INTEGER:*y]
        -> [BASE:*z] ;
     [A:*x] -> (ELEM) -
        <- [INTEGER:*w] -> (DFFR) -> [INTEGER:*y]
        -> [BASE:*v]
[THEN: [BASE:*y] -> (DFFR) -> [BASE:*v] ]].
```

5.3 Binary Data Type

The *EXPRESS* reference manual itself considers the BINARY data type to be a non-empty sequence of bits. Using the semantics developed above we will have in the *EKP*

```
type BIT(x) is
    [INTEGER:2] -> (>) -> [INTEGER:*x] -> (>) -> [INTEGER:-1].
type BINARY(x) is
    [LIST:*x] -
        (ATTR:low_bound) -> [INTEGER: 1]
        (ATTR:base_type) -> [BIT].
```

and therefore the binary literal `%10` is represented as

```
[BINARY] -
    1 -> (ELEM) -
         2 <- [INTEGER: 1]
         3 -> [BIT: 1],
    1 -> (ELEM) -
         2 <- [INTEGER: 2]
         3 -> [BIT: 0].
```

5.4 Enumeration Data Type

The ENUMERATION data type provided by *EXPRESS* is the usual ordered set of names. We must have in *EKP* the canonical graph for the "less-than" relation

```
    [ENUMERATION:*x] -> (LT) -> [ENUMERATION:*y].
```

together with the information that it is a transitive relation

```
[IF: [ENUMERATION:*x] -> (LT) -> [ENUMERATION:*y] ;
     [ENUMERATION:*y] -> (LT) -> [ENUMERATION:*z]
[THEN: [ENUMERATION:*x] -> (LT) -> [ENUMERATION:*z] ]].
```

The declaration

`TYPE` *type_id* `= ENUMERATION OF (`$name_1, \ldots, name_n$`); END_TYPE;`

is processed in the following way:

1. Create a new type *type_id* < `ENUMERATION`.
2. For each $i = 1, \ldots, n$ create a new individual marker e_i such that *type_id*: :e_i and add the graph `[`*type_id*:e_i`] -> (NAME) -> [WORD:`$name_i$`]` to the canonical basis.
3. For each $i \in \{1, \ldots, n-1\}$ add `[`*type_id*:e_i`] -> (LT) -> [`*type_id*:e_{i+1}`]` to the canonical basis.

5.5 Logical Data Type

"A LOGICAL data type represents a TRUE, FALSE or UNKNOWN value. The following condition holds for logical data types: FALSE $<$ UNKNOWN $<$ TRUE." Fortunately we can represent that in *EXPRESS* itself

```
TYPE LOGICAL = ENUMERATION OF (FALSE, UNKNOWN, TRUE);
END_TYPE;
```

and therefore apply the translation scheme presented in the previous section:

```
LOGICAL < ENUMERATION.
LOGICAL :: #123456.
[LOGICAL:#123456] -> (NAME) -> [WORD: "false"].
LOGICAL :: #123457.
[LOGICAL:#123457] -> (NAME) -> [WORD: "unknown"].
LOGICAL :: #123458.
[LOGICAL:#123458] -> (NAME) -> [WORD: "true"].
[LOGICAL:#123456] -> (LT) -> [LOGICAL:#123457].
[LOGICAL:#123457] -> (LT) -> [LOGICAL:#123458].
```

6 Schemata

The *EXPRESS* construct used to partition information bases is the schema, which contains a collection of type declarations (among other things not covered in this paper). There is no exporting mechanism: if a schema wants to refer to a type defined in another schema, it must use the USE or the REFERENCE interface specification, stating what types are imported from what schemata. In both cases, the importing schema can only use the imported name but not its definition (e.g. in the case of an entity type, one can't refer to its attributes). The difference is that USEd types are implicitly made visible to other modules. As an example, let's suppose that schema s_1 defines type `A` and s_2 defines `B` in terms of `A` and s_3 defines `C` in terms of `A` and `B`. If s_2 USEs `A` from s_1 then s_3 needs only to refer to s_2. However, if s_2 REFERENCEs `A` then s_3 must import `A` from s_1.

A schema is translated into a knowledge packet as defined in section 2.1 where *all* the available type labels are exported, except for those imported using the REFERENCE interface specification. This makes sure that no graph can refer to types used for defining an imported type and it also makes sure the REFERENCEd names aren't visible outside the importing module.

7 Conclusions

This paper provided a first step for mapping *EXPRESS* into Conceptual Structures. For that purpose a means to modularize CS knowledge bases was defined.

The approach shown needs further refinement but is enough to denote *EXPRESS* schemata.

We have also shown how to reconstruct the semantics of *EXPRESS* using CS theory thereby providing a unifying framework to check for semantic correctness and to compare models using the inference rules and the canonical formation rules.

Another advantage of this approach is the possibility to compare or integrate models built with different frameworks (e.g. NIAM, E-R) provided that translation schemes exist for those formalisms. But as *EXPRESS* is more expressive than many other modeling languages, having a translation scheme for it facilitates enormously the task for the other languages.

References

1. ISO/IEC JTC1/SC21/WG3. The IRDS conceptual schema. Working paper, International Organisation for Standardization, 1992.
2. John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine.* The System Programming Series. Addison-Wesley Publishing Company, 1984.
3. ISO TC184/SC4/WG3. Semantic unification meta model—volume 1: Semantic unification of static models. Technical report, International Organisation for Standardization, October 1992. Prepared by the Dictionary/Methodology Committee of the IGES/PDES Organization.
4. ISO TC184/SC4/WG5. The EXPRESS language reference manual. Committee Draft 10303 - 11, International Organisation for Standardization, May 1991.
5. Michel Wermelinger. A reconstruction of *EXPRESS* using conceptual structures. Technical report, Departamento de Informática, Universidade Nova de Lisboa, May 1993.

This article was processed using the LaTeX macro package with LLNCS style