Chapter 4

The Specification and Proof of an EXPRESS to SQL "Compiler"

Juan Bicarregui and Brian Matthews

Summary

EXPRESS and SQL are two ISO standard languages for modelling data. However, EXPRESS is abstract in the sense that it is intended to be used to define application-oriented data types, whereas SQL is concrete in that all data must be modelled using relation tables. In this chapter, we specify and prove some properties of an EXPRESS to SQL "compiler" which implements the ISO standard STEP Data Access Interface (SDAI) for the storage and retreval of EXPRESS instance data.

The "compiler" is formalised as a refinement: an abstract model of the EXPRESS database is given which defines operations for the storage and retreval based on a model of the EXPRESS data types on VDM; then a specification of a relational database is given as a basis for a refinement of the EXPRESS database; and then concrete versions of the operations are defined on top of the relational model. These operations are, in effect, the semantic functions of the compiler. The equivalence of the abstract and concrete specifications is the justification of the correctness of the compiler.

We outline the specifications and prove an obligation concerning the refinement. A number of issues concerning modelling style arise in comparing the EXPRESS data types with those of VDM and concerning the structuring of the development to facilitate proofs. The specification was developed using the IFAD VDM-SL Toolbox and the proofs constructed by hand.

4.1 STEP and EXPRESS

EXPRESS is an "information modelling" language developed within the ISO *Standard for the Exchange of Product model data* (STEP) [1]. By defining standard languages for the representation of data models and data compliant with those models, and by standardising particular data models for specific application areas, STEP provides a vendor-neutral mechanism for the representation of product data and hence facilitates the open exchange of data between applications.

The STEP standard is divided into a number of parts. Some parts define "generic technologies" for the definition of models and data, others give particular data models (*Application Protocols*) for representation of data in specific application areas. For example the application protocol for geometric and topological representation, defines that a circle should be represented by its centre and radius, rather than say by a diameter or by three points on the circumference.

The generic technologies include EXPRESS (Part 11) which is a language for the definition of data models and EXPRESS-I (Part 12) for representation of data. Part 21 gives a syntax for a condensed form of EXPRESS-I and Parts 22-26 give a standard API for accessing an EXPRESS-based database, the SDAI (STEP Data Access Interface). An abstract definition of the SDAI is given based on a partial model of EXPRESS in EXPRESS and an informal definition of the operations to manipulate this model. Concrete versions of the SDAI are defined in a number of "language bindings" giving the data structures for implementations in particular languages.

4.1.1 The Context

The work described in the chapter was undertaken in the context of the ProcessBase project[1] which concerns the development of an Application Protocol for Process Plants (AP221 and AP227) and is also developing software supporting translation between the APs and some native CAD formats (AutoCAD and PID) for 2D schematics, 3D models and functional data.

The translation between native and STEP data formats is achieved by some application specific conversion software. Because data is structured radically differently in the standard and native models this software requires intensive access to all parts of the data and data models.

The implementation is achieved by interrogation of an SDAI database built on top of relational technology. Effectively two separate repositories are created, one loaded with each data model. Translation takes place by selecting from the source database the components required to construct each data item in the target database. The EXPRESS "shell" built on top of the relational database is the EXPRESS to SQL "compiler" referred to in the title of this chapter.

[1] ESPRIT Project 6212, ProcessBase

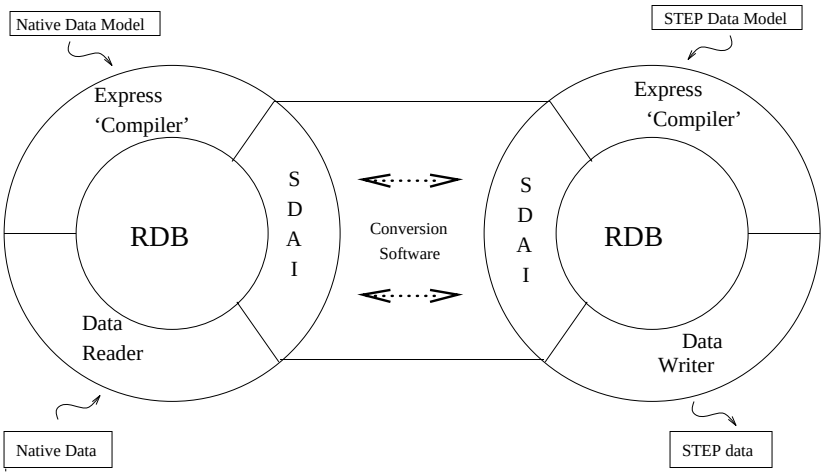

Figure 4.1: The conversion takes place between 2 virtual SDAIs.

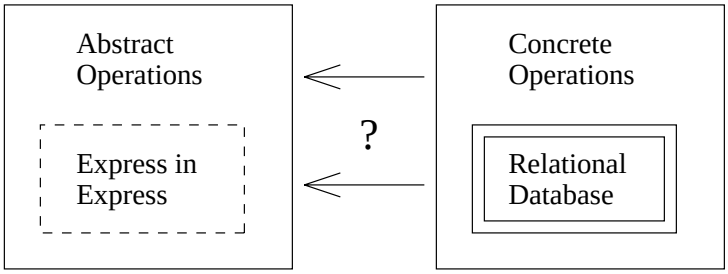

Figure 4.2: The compiler is formalised as a refinement

4.1.2 The Specifications

The abstract specification is conceptually a single module. Its major datatypes are an abstract syntax for the EXPRESS language itself. These are based on the definition of EXPRESS in EXPRESS given in the SDAI but are remodelled to account for some differences between the VDM and EXPRESS. The concrete specification is conceptually two modules. modules The first follows the definition of SQL in VDM given in [5]. This should be simple enough to be implemented on top of any relational database. The second builds on this to give an implementation of the SDAI. (In fact, some datatypes common to abstract and concrete specifications are given as a separate module.) The refinement relation between specifications is the justification of the correctness of the compiler (Figure 4.2).

4.1.3 Related Work

EXPRESS is independent of any particular database technology, however, several implementations of EXPRESS tools founded on relation database technology exist, for example, [15, 16, 18, 13, 17]. These were compared in a preliminary study to this work [9]. One implementation of EXPRESS in SQL was reverse engineered to define the translation and the compiler formalised as a refinement between specifications based upon an existing VDM definition of the semantics of SQL and the definition of EXPRESS in EXPRESS given in ISO 10303 Part 22.

Formal definitions exist for several aspects of this development including a VDM formalisation of the semantic of SQL [5], two EXPRESS models of EXPRESS itself [3, 7], and a VDM definition of part of the SDAI [8]. An interesting development which is trying to define a unifying semantic model for several languages is [4].

VDM is a mature notation for the definition of formal languages. (e.g. [20, 21, 22, 23, 24, 25]). It is undergoing standardisation by BSI and ISO[11]. Several support tools for VDM are available, the present work was undertaken using the IFAD VDM-SL Toolbox [19]

4.1.4 Overview

Section 2 gives an outline of some key concepts of the EXPRESS language and Section 3 the abstract specification of the EXPRESS database. Section 4 gives a simplified relational database and Section 5 defines the EXPRESS database operations on top of this. Section 6 discusses the approach taken.

4.2 An Outline of EXPRESS

EXPRESS is a language for defining data models. It has three levels of granularity. The coarsest level components are Schema which are akin to modules; modules the second and third are Entities and Attributes which loosely correspond to datatypes (or object classes) and fields (or instance variables) respectively.

We present some of the most important features of the EXPRESS language through a simple example based on one given in [6].

4.2.1 Entities

The EXPRESS entity *car* describes the components of a car. As in a VDM record type, each component is given a name and a type.

```
ENTITY car                          ;
        model_type : car_model      ;
          made_by : manufacturer    ;
          mnfg_no : STRING          ;
    registration_no : STRING        ;
  production_year : INTEGER         ;
         owned_by : owner           ;
END-ENTITY

ENTITY car_model              ;
          name : STRING       ;
       made_by : manufacturer ;
  consumption : REAL          ;
END-ENTITY
```

Where Rules.

It is clear that the *made_by* attribute should be the same for a car and for the model of that car. This can be formalised as a "where rule" given in the definition of car, for example

WHERE made_by :=: model_type.made_by

This constraint is akin to an invariant in VDM. Note that *:=:* is object identity (see later).

Derived Attributes.

An alternative way to model this constraint would be to give the *made_by* attribute as a *derived attribute* of car.i.e.

DERIVE made_by := model_type.made_by

This indicates that the maker of a car can be derived (in this case trivially) from the maker of the model of the car.

In VDM, the same information could be extracted by use of an auxiliary function.

Object Identifiers.

A fundamental difference between an EXPRESS entities and record types in say VDM, is that an EXPRESS entity declaration indicates an implicit indirection in the data. Following the object-oriented style, each instance of an entity corresponds to a identified object and an environment is assumed which dereferences object identifiers. For example, an instance of a car might be

#1 = car(#2,#3,"VW233445", "N123PQR", 1995, #4)

#2 = car_model("GOLF", #3, 2.6)

#3 = manufacturer(...)

...

where the *#i* represent the object identifiers. Note that the same manufacturer appear in both the *car* and *car_model* instances.

Uniqueness Constraints.

A number of constraints can be given which restrict the universe of instances. Uniqueness of attributes or combinations of attributes can be specified across all instances of an entity. For example, the rule *single* states that the registration number should be globally unique over all cars and *joint* states that, together, the *made_by* and *mnfg_no* fields are also globally unique.

UNIQUE single : registration_no ;
joint : made_by, mnfg_no ;

Existence Constraints.

It is also possible to give constraints which describe the necessity of the existence of some instance. For example, if *owner* is defined by

ENTITY *owner* ;
owns : SET OF car ;
END-ENTITY

then, in the entity definition for cars, one can state that every car must have an owner by giving *owned_by* as an "inverse attribute" as follows:

INVERSE owned_by : owner FOR owns

This stipulates that every car must appear uniquely in the set of cars owned by some owner. This is equivalent to a clause in the invariant

inv-car(c) $\triangleq$ $c \in c.owned\_by.owns$

By defining specific pieces of syntax for many common modelling situations, EXPRESS entities give a means to specify certain types of relational constraints concisely. This approach can be more "engineer friendly" than the explicit formalisation of such constraints in the general logic of invariant predicates. However the interpretation of these constructs can be somewhat confusing to the newcomer.

4.2.2 Other Type Constructors

EXPRESS also includes a number of other type constructors which can be used without the overhead of object indirection. Type expressions using these construc-

tors are named in defined types. For example there are arrays, lists, sets, bags, enumerated types and select types (unions). However, there are no map or function types and tuples can only be constructed through the use of entities.

4.2.3 Subtypes

The object-flavoured nature of EXPRESS is reinforced by its use of subtypes. A subtype inherits and extends the set of attributes of its supertype. There are three ways in which inheritance can be employed.

Simple inheritance is declared in both supertype and subtype, for example by the combination

ENTITY C SUPERTYPE OF ONEOF (A,B) ....

and

ENTITY A SUBTYPE of C ....
ENTITY B SUBTYPE of C ....

Here A inherits from C and so does B.

A form of multiple inheritance is declared by

ENTITY C SUPERTYPE OF AND (A,B) ....

which allows instances with the fields of A *and* B and C

The third form *ANDOR(A,B)* is equivalent to *ONEOF(A, B, AND(A,B))*.

A supertype can be declared as *abstract* in which case it can only be instantiated through its subtypes.

The *SUPERTYPE* clause can be omitted and inferred from the subtypes declarations. However, the use of *ANDOR* as the default combinator for the subtypes when no supertype clause is given makes its omission highly non-compositional. explicitly.

Subtyping provides a way to condense a data model definition by avoiding repetition of combinations of attributes which are common to several entity definitions. However, it is possible to expand out the subtype hierarchy to give a model with an equivalent set of valid instances. For present purposes, we assume that all subtyping has been expanded out.

Thus EXPRESS provides many (though not all) of the forms of type construction available in VDM and includes a form of subtyping and object identity not available there.

4.3 The Abstract EXPRESS Database

The SDAI standard defines three components in the abstract definition of the interface. Firstly, the "data dictionary" defines the types used. This is the abstract syntax for the data models written in EXPRESS (the meta-data) the instance data written in EXPRESS-I (the actual data). The second component is the "session" which represents the components manipulated during execution (the state). These two components are given as formal models in EXPRESS. Thirdly, there are the operations which, although described in terms of the formal models are given informally in the standard.

The abstract VDM specification is built from two modules. modules One module defines the types used as parameters to the operations, the other defines the abstract state and operations.

Rather than give a VDM model which is as close as possible to the EXPRESS model in the standard (the approach which is followed in [8]), we make certain modelling decisions in "translating" from EXPRESS to VDM. The new model makes use of the facilities available in VDM to replace some of the features of the EXPRESS model which could not be modelled directly.

In an informal translation such as this, there remains a danger that the model we construct does not correspond to that in the standard. However, this approach was adopted after early attempts to follow the EXPRESS style in VDM encountered difficulties with handling object identifiers and the environment to dereference them. This confirmed previous experience [10] that trying to translate directly from one language to an apparently similar one can lead to problems.

4.3.1 The EXPRESS and EXPRESS-I Abstract Syntax

The first module modules defines the abstract syntax for the EXPRESS Types, that is, the meta-data and instance data stored by the database. This follows the structure of the data dictionary in the SDAI. It will be shared by both abstract and concrete specifications.

module *ET*

exports all

The following sections do not attempt to be comprehensive in their coverage of the EXPRESS language, but rather describe those aspects which lead to interesting modelling issues.

Schema Definitions.

We begin by giving a model for schemas. A schema definition has a set of entities which it declares, a set of types it declares, and some global rules.

$$\begin{aligned} \textit{schema-def} :: & \textit{exp-types} : \textit{type-name} \xrightarrow{m} \textit{type-def} \\ & \textit{entities} : \textit{entity-name} \xrightarrow{m} \textit{entity-def} \\ & \textit{global-rules} : \textit{global-rule}\text{-set}; \end{aligned}$$

In keeping with VDM style, we exploit the fact that the *type-name*s and *entity-name*s are unique and use these as the indices in map types.

In the EXPRESS model, these names are given as components of the entity and type definitions, sets of which are components of schema definitions. Constraints are then given which state that the names are unique within the parent object. For example, the model of entity definition ensures that the entity names are unique within the parent schema by the following combination of an inverse attribute and a uniqueness rule:

INVERSE parent-schema : schema-definition FOR entities

UNIQUE UR1: entity-name, parent-schema

This construction is rather common in EXPRESS.

Type Definitions.

Entity names and a number of other type constructors can be used in type definitions. For brevity we do not go into this further here.

$$\textit{type-def} = \textit{entity-name} \mid \ldots;$$

Entity Definitions.

Freed from its name, an entity definition is simply a map from attribute names to attribute definitions.

$$\textit{entity-def} = \textit{attribute-name} \xrightarrow{m} \textit{attribute-def};$$

Attribute Definitions.

Attribute definitions can be of one of three forms, defined in EXPRESS using a subtype/supertype relationship:

ENTITY attribute ABSTRACT SUPERTYPE OF (

ONEOF(derived_attribute, explicit_attribute, inverse_attribute))

For brevity, we only model explicit attributes here[2].

[2] One way in which it is possible to model the subtype/supertypes relationship in VDM is to expand out the supertype's attributes into each subtype. Another is to define the supertype's attributes as a composit type and explicitly include this an attribute in the subtypes. Note that this latter approach can be taken without the cost in complexity that the extra level of indirection would have introduced if using EXPRESS entities.

$$\begin{aligned} \textit{attribute-def} = {} & \textit{entity-name} \mid \\ & \textit{type-name} \mid \\ & \textit{aggregate-def} \mid \\ & \ldots; \end{aligned}$$

An attribute definition is the type of the attribute. It can either be an entity type, indicated by the entity name, or any named type, or some particular forms of type expressions, such as aggregate types which generalise sets, bags, lists and arrays. It is unclear why not all forms of type expression are permitted as the types of attributes.

Models.

A *model* is the set of instances of the entities of a schema. For consistency with the use of "schema definition" above, here we call it a "data definition"

$$\textit{data-def} = \textit{instance-ref} \xrightarrow{m} \textit{instance-def};$$

The SDAI states that "the *underlying-schema* of the model must be the schema that defines the structure of the data that appears in the model", but does not formalise this. Formalising this requirement includes ensuring that each instance in the model is an instance of an entity in the underlying schema. This constraint emerges here as part of the invariant when instance data and meta data are brought together.

Instances.

The SDAI defines a hierarchy of kinds of instance: an *application-instance* is an instance of an entity defined in an application schema and an *sdai-instance* is an instance of an entity defined in one of the schemas of the SDAI definition itself. The latter are divided into *dictionary-instance*s and *session_instance*s. The different classes are then used to distinguish types of instances when required. However, all of these instances have the same attributes so here we do not need to model them separately but rather can define functions that distinguish them if necessary.

$$\begin{aligned} \textit{instance-def} :: {} & \textit{entity} : \textit{entity-name} \\ & \textit{attributes} : \textit{attribute-name} \xrightarrow{m} \textit{attribute-value}; \end{aligned}$$

$$\begin{aligned} \textit{attribute-value} = {} & \textit{instance-ref} \mid \\ & \textit{expression} \mid \\ & \textit{aggregate-value} \mid \\ & \ldots; \end{aligned}$$

The *entity* field of an instance definition is the entity-definition of which this is an instance. This "cross-linking" is required because this schema-centered model does not tie instances to entities in any other way. (Some alternative approaches are

discussed in 4.3.3.) In general, this form of duplication is undesirable as it leads to many consistency checks in invariants and operations. belongs"

4.3.2 The State and Operations

The second module modules of the abstract specification defines the state and operations. It imports the EXPRESS types from the last module.

module *AEDB*

imports

from *ET* all

In an SDAI "session"[3]. a database has a number of *db*s indexed by schema names Also, if there are several models for the same schema, these appear as separate *db*s, each with its own copy of the schema definition. (The alternative of composing each schema definition with a set of models is also discussed in Section 4.3.3.)

However, for simplicity, we specify a database described by a single schema. The state brings together the schema and data in a database segment (a real database may incorporate a number of these). The database is initially empty[4].

state *db* of
  schema : *schema-def*
  model : *data-def*
  inv mk-*db* $(s, im) \triangleq$
    $\forall\, ir \in \mathsf{dom}\ im \cdot$
      $\exists\, en \in \mathsf{dom}\ s.entities \cdot$
        isa-instance-of $(ir, en, \mathsf{mk}\text{-}db\,(s, im))$;
  init $db \triangleq db = \mathsf{mk}\text{-}db\,(\mathsf{mk}\text{-}schema\text{-}def\,(\{\mapsto\}, \{\mapsto\}, \{\,\}), \{\mapsto\})$
end

The instance and meta data are tied together via the *entity* field in the instance definitions. The invariant states that all the instances must be instances of some entity definition in the schema by using the function *isa-instance-of*. This function takes as arguments the instance reference, the entity name, and the whole database. The last is required in order that the first two can be dereferenced.

[3] In the SDAI, a session supports a number of repositories, here (like most of the implementations) we assume the EXPRESS database contains a single repository.

[4] The syntax *ET*' is used for referring to constructs from the imported module.

$$isa\text{-}instance\text{-}of : instance\text{-}ref \times entity\text{-}name \times db \to \mathbb{B}$$

$$\begin{array}{l} isa\text{-}instance\text{-}of\ (ir, en, d) \triangleq \\ \quad \mathsf{let}\ id = d.instance\text{-}map\,(ir), \\ \qquad em = d.schema.entities\ \mathsf{in} \\ \quad names\text{-}match\,(id.entity, en) \wedge \\ \quad doms\text{-}equal\,(id.attributes, em\,(en)) \wedge \\ \quad \forall\, an \in \mathsf{dom}\ id.attributes \cdot \\ \qquad isa\,(id.attributes\,(an), em\,(en)\,(an), d); \end{array}$$

Three conditions are required to be satisfied for an instance to be a valid instance of an entity. Firstly, the names of the instance and the entity must match, secondly, the attributes must have the same names in instance and entity definition, and lastly each attribute value must be of type of the corresponding attribute definition. This last condition is formalised by the function *isa* which is a general type checking function which can be applied to any value/type pair and recursively recursively calls *isa-instance-of* in the case that the pair is an instance/entity.

$$names\text{-}match : entity\text{-}name \times entity\text{-}name \to \mathbb{B}$$

$$\begin{array}{l} names\text{-}match\,(en1, en2) \triangleq \\ \quad en1 = en2; \end{array}$$

$$doms\text{-}equal : (@A \xrightarrow{m} @B) \times (@A \xrightarrow{m} @C) \to \mathbb{B}$$

$$\begin{array}{l} doms\text{-}equal\,(m1, m2) \triangleq \\ \quad \mathsf{dom}\ m1 = \mathsf{dom}\ m2; \end{array}$$

$$isa : attribute\text{-}value \times attribute\text{-}def \times db \to \mathbb{B}$$

$$\begin{array}{l} isa\,(av, ad, d) \triangleq \\ \quad \mathsf{cases}\ \mathsf{mk}\text{-}(av, ad) : \\ \qquad \mathsf{mk}\text{-}(\mathsf{mk}\text{-}instance\text{-}ref\,(\text{-}), \mathsf{mk}\text{-}entity\text{-}name\,(\text{-})) \to \\ \qquad\quad isa\text{-}instance\text{-}of\,(av, ad, d), \\ \qquad \mathsf{mk}\text{-}(\mathsf{mk}\text{-}expression\,(\text{-}), \mathsf{mk}\text{-}type\text{-}name\,(\text{-})) \to \\ \qquad\quad \ldots\ldots \\ \quad \mathsf{end}; \end{array}$$

A number of auxiliary function are also provided for convenience in the operation definitions that follow. For example:

$$delete\text{-}instance : db \times instance\text{-}ref \to db$$

$$\begin{array}{l} delete\text{-}instance\,(d, ir) \triangleq \\ \quad \mu\,(d, instance\text{-}map \mapsto \{ir\} \lhd d.instance\text{-}map) \\ \mathsf{pre}\ \ ir \in \mathsf{dom}\ d.instance\text{-}map\ ; \end{array}$$

$$
\begin{array}{l}
\mathit{add\text{-}instance} : \mathit{instance\text{-}def} \times \mathit{db} \to \mathit{db} \times \mathit{instance\text{-}ref} \\
\mathit{add\text{-}instance}\,(i, d) \triangleq \\
\quad \mathsf{let}\ \mathit{nir} = \mathit{new\text{-}ir}\,(d), \\
\qquad \mathit{ndb} = \mu\,(d, \mathit{instance\text{-}map} \mapsto d.\mathit{instance\text{-}map} \uplus \{\mathit{nir} \mapsto i\})\ \mathsf{in} \\
\quad \mathsf{mk\text{-}}(\mathit{ndb}, \mathit{nir}) \\
\mathsf{pre}\ \ i.\mathit{entity} \in \mathsf{dom}\ d.\mathit{schema.entities} \wedge \\
\qquad \mathsf{dom}\ i.\mathit{attributes} = \mathsf{dom}\ d.\mathit{schema.entities}\,(i.\mathit{entity}) \wedge \\
\qquad \forall\, \mathit{an} \in \mathsf{dom}\ i.\mathit{attributes} \cdot \\
\qquad\quad \mathit{isof\text{-}type}\,(i.\mathit{attributes}\,(\mathit{an}), d.\mathit{schema.entities}\,(i.\mathit{entity})\,(\mathit{an}), d);
\end{array}
$$

$$
\begin{array}{l}
\mathit{new\text{-}ir} : \mathit{db} \to \mathit{instance\text{-}ref} \\
\mathit{new\text{-}ir}\,(d) \triangleq \\
\quad \mathsf{let}\ \mathit{ir} : \mathit{instance\text{-}ref}\ \mathsf{be\ st}\ \mathit{ir} \notin \mathsf{dom}\ d.\mathit{instance\text{-}map}\ \mathsf{in} \\
\quad \mathit{ir}
\end{array}
$$

The Operations.

The SDAI defines some 50 operations, not all of which were formalised in VDM. There are also some operations required to complete the description of the EXPRESS database which are not defined in the SDAI. These concern instantiating the database by reading EXPRESS data models and instances from file.

Here we give a few example a pair of operations which add and delete entities within the database and a pair which add and delete instances.

$$
\begin{array}{l}
\mathit{add\text{-}entity}\,(\mathit{en} : \mathit{ET}\text{`}\mathit{entity\text{-}name}, \mathit{ed} : \mathit{ET}\text{`}\mathit{entity\text{-}def}) \\
\mathsf{ext\ wr}\ \mathit{schema} : \mathit{ET}\text{`}\mathit{schema\text{-}def} \\
\mathsf{pre}\ \ \mathit{en} \notin \mathsf{dom}\ \mathit{schema.entities} \\
\mathsf{post}\ \mathit{schema.entities} = \overleftarrow{\mathit{schema}}.\mathit{entities} \dagger \{\mathit{en} \mapsto \mathit{ed}\} \wedge \\
\qquad \mathit{schema.exp\text{-}types} = \overleftarrow{\mathit{schema}}.\mathit{exp\text{-}types} \wedge \\
\qquad \mathit{schema.global\text{-}rules} = \overleftarrow{\mathit{schema}}.\mathit{global\text{-}rules}
\end{array}
$$

$$
\begin{array}{l}
\mathit{delete\text{-}entity}\,(\mathit{en} : \mathit{ET}\text{`}\mathit{entity\text{-}name}) \\
\mathsf{ext\ rd}\ \ \mathit{model} : \mathit{ET}\text{`}\mathit{data\text{-}def} \\
\qquad \mathsf{wr}\ \mathit{schema} : \mathit{ET}\text{`}\mathit{schema\text{-}def} \\
\mathsf{pre}\ \ \mathit{en} \in \mathsf{dom}\ \mathit{schema.entities} \wedge \\
\qquad \forall\, i \in \mathsf{rng}\ \mathit{model} \cdot i.\mathit{entity} \neq \mathit{en} \\
\mathsf{post}\ \mathit{schema.entities} = \{\mathit{en}\} \mathbin{\lhd} \overleftarrow{\mathit{schema}}.\mathit{entities} \wedge \\
\qquad \mathit{schema.exp\text{-}types} = \overleftarrow{\mathit{schema}}.\mathit{exp\text{-}types} \wedge \\
\qquad \mathit{schema.global\text{-}rules} = \overleftarrow{\mathit{schema}}.\mathit{global\text{-}rules}
\end{array}
$$

$add\text{-}instance\,(i : ET`instance\text{-}def, en : ET`entity\text{-}name)\ ir : ET`instance\text{-}ref$
ext rd $schema : ET`schema\text{-}def$
wr $model : ET`data_def$
pre $en \in \mathsf{dom}\ schema.entities \wedge$
$ET`names\text{-}match\,(i.entity, en) \wedge$
$\mathsf{dom}\ i.attributes = \mathsf{dom}\ schema.entities\,(en) \wedge$
$\forall\, an \in \mathsf{dom}\ i.attributes \cdot$
$ET`isa\,(i.attributes\,(an),$
$schema.entities\,(en)\,(an),$
$\mathsf{mk}\text{-}db\,(schema, model))$
post let $\mathsf{mk}\text{-}(ndb, nir) = ET`add\text{-}instance\,(i, \mathsf{mk}\text{-}db\,(\overleftarrow{schema}, \overleftarrow{model}))$ in
$\mathsf{mk}\text{-}db\,(schema, model) = ndb \wedge ir = nir$;

$delete\text{-}instance\,(ir : ET`instance\text{-}ref)$
ext rd $schema : ET`schema\text{-}def$
wr $model : ET`data_def$
pre $ir \in \mathsf{dom}\ data\text{-}def$
post $\mathsf{mk}\text{-}db\,(schema, model) =$
$ET`delete\text{-}instance\,(\mathsf{mk}\text{-}db\,(\overleftarrow{schema}, \overleftarrow{model}), ir)$

4.3.3 Reflections on the Abstract Specification

The most fundamental difference in style between EXPRESS and VDM arises from the dereferencing implicit in instances of EXPRESS entities. In VDM, when indirection is required it must be made explicit by use fo a map type; in EXPRESS, each entity definition indicates an indirection in the corresponding instance data.

No uniform approach was taken here in determining which EXPRESS entities should be given object identifiers in the VDM model. However, often a strong indication that a map type was required came from the use of the combination of inverse attribute and uniqueness rule described above.

The data model given in [2] is rather concrete and includes a great proliferation of cross-references and repetition of data which makes it difficult to give a sufficiently strong invariant and operation definitions. The ubiquity of object identifiers in EXPRESS models can lead to a blurring of the roles of composit types as tuple constructors and to indicate indirection in the data.

The treatment here has removed many of these cross-references but there is still some "redundancy" remaining. The model given here can be seen as schema-centred in that the meta data and instance data are separated at the level of schema. The overall model of *db* is

$$
\begin{array}{l}
\mathit{entity\text{-}name} \xrightarrow{m} (\mathit{attribute\text{-}name} \xrightarrow{m} \mathit{attribute\text{-}def}) \\
\times \\
\mathit{instance\text{-}ref} \xrightarrow{m} \mathit{entity\text{-}name} \\
\qquad\qquad\qquad\quad \times \\
\qquad\qquad\qquad\quad (\mathit{attribute\text{-}name} \xrightarrow{m} \mathit{attribute\text{-}val})
\end{array}
$$

This makes instance references unique throughout the instances of the schema and requires the three consistency clauses given above.

An alternative, class-centered, model would bring the instance references within the entity definitions to give

$$
\begin{array}{l}
\mathit{entity\text{-}name} \xrightarrow{m} (\mathit{attribute\text{-}name} \xrightarrow{m} \mathit{attribute\text{-}def}) \\
\qquad\qquad\qquad\quad \times \\
\qquad\qquad\qquad\quad \mathit{instance\text{-}ref} \xrightarrow{m} (\mathit{attribute\text{-}name} \xrightarrow{m} \mathit{attribute\text{-}val})
\end{array}
$$

Here instance references are only unique within the instances of a class and only the *isa* and *doms-equal* clauses are required.

A third possible model would be instance-centered.

$$
\begin{array}{l}
\mathit{instance\text{-}ref} \xrightarrow{m} \mathit{entity\text{-}name} \\
\qquad\qquad\qquad\quad \times \\
\qquad\qquad\qquad\quad (\mathit{attribute\text{-}name} \xrightarrow{m} (\mathit{attribute\text{-}def} \\
\qquad\qquad\qquad\qquad\qquad\quad \times \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{attribute\text{-}val}))
\end{array}
$$

Now, only the *isa* constraint is required in the invariant although a new one is now needed which states that all instances of the same entity have the same structure.

This last model highlights the fact that the entity names could be considered to be redundant, entities with the same structure being equivalent. However, this value-based rather than an instance-based approach does diminish the potential for the classification of instances. It is a matter for debate which of these models is more abstract or more convenient in use. Which model is preferred is a matter of choice but experience shows that it is often convenient to use whichever model has simplest invariant.

4.4 A Relational Database

The basis for the concrete specification of the EXPRESS database is a VDM definition of an idealised relational database given in [5].

4.4.1 Signature

We are careful only to export the types and operations which we want to make available to the implementation. Note that the structure of the types is exported meaning

that the default constructors and selectors are also exported. This is considerably more concise than defining and exporting each function explicitly.

module *RDB*

exports

types struct *Relation*,
struct *RelationName*,
struct *Field*,
struct *Fields*,
...

There are just seven operations in the idealised database.

operations $Create : RelationName \times Fields \xrightarrow{o} ()$,
$Expand : RelationName \times Field \xrightarrow{o} ()$,
$Drop : RelationName \xrightarrow{o} ()$,
$Insert : Ins \mid InsSel \xrightarrow{o} ()$,
$Update : Upd \mid UpdCond \xrightarrow{o} ()$,
$Delete : Del \mid DelCond \xrightarrow{o} ()$,
$Select : (Sel \mid SelCond) \times Environment \xrightarrow{o} Relation$

4.4.2 Datatypes

A number of type declarations are required to support the definition of the operations. [5] defines some "syntactic types" which are the types which are used in the parameters and results of the operations, and "semantic types" which are used to model the internal structures of the database. Here we give a top-down presentation of some of the types without distinguishing the two classes.

$Database = RelationName \xrightarrow{m} Relation;$

$Relation :: scheme : Scheme$
$\quad tuples : Tuple\text{-set}$

inv mk-$Relation\,(s, ts) \triangleq$
$\forall\, t \in ts \cdot$
$\quad \mathsf{dom}\ t = \mathsf{dom}\ s \wedge$
$\quad \forall\, att \in \mathsf{dom}\ t \cdot Compatible\text{-}type\,(t\,(att), s\,(att));$

$Scheme = AttributeName \xrightarrow{m} Attribute;$

$Tuple = AttributeName \xrightarrow{m} Value;$

$Attribute = DataType$

$Fields = Field\text{-set};$

$Field :: name : AttributeName$
$\quad type : DataType;$

$DataType = \text{INTEGER} \mid \text{STRING} \mid RelationName$

...

There are many more syntactic domains for queries in the full specification which are used for selections, conditions, comparisons, boolean operations, arithmetic operations, etc.

4.4.3 The State and Operations

The state of the module is a single database. It is initially empty.

state rdb of
  $db : Database$
  init $rdb \triangleq db = \mathsf{mk}\text{-}rdb(\{\mapsto\})$
end

Note that this model only supports a single schema.

A number of auxiliary functions are used in the definition of the operations. One which is necessary in the following given here.

$Make\text{-}Scheme : Fields \to Scheme$

$Make\text{-}Scheme(fs) \triangleq$
  $\{f.name \mapsto f.type \mid f \in fs\};$

The seven operation given in the signature are sufficient to implement the abstract operations given above. Just one example is given here.

$Create(r : RelationName, fs : Fields)$
ext wr $db : Database$
pre $r \notin \mathsf{dom}\ db \wedge$
  $\forall fi, fj \in fs \cdot$
    $fi \neq fj \Rightarrow fi.name \neq fj.name$
post let $rel = \mathsf{mk}\text{-}Relation(Make\text{-}Scheme(fs), \{\})$ in
  $db = db \dagger \{r \mapsto rel\}$

4.4.4 Reflections on the Relational Database Specification

The full specification of the relational database runs to some 10 pages of VDM. Although this was taken directly from a published specification, some significant

effort was required to fully formalise the specification and put it into the support tool. It is reassuring to note that only very minor errors were found during this process. Of course, the effort required to achieve full formality is repaid by the possibilities arising from automatic manipulation of the formal description, for example, proof obligation and test case generation, animation and even code generation.

4.5 A Concrete EXPRESS Database

A refinement of the abstract EXPRESS database was built on top of the relational database.

The concrete EXPRESS database module modules import the abstract syntax of EXPRESS defined in the abstract specification and the relational database.

module *CEDB*

imports

from *ET* all ,

from *RDB* all

The state of the concrete EXPRESS database is exactly a relational database.

We first define some auxiliary functions.

functions

$$make\text{-}entity\text{-}name : RDB`RelationName \to ET`entity\text{-}name$$
$$make\text{-}entity\text{-}name\ (r) \triangleq$$
$$\ldots;$$

$$make\text{-}RelationName : ET`entity\text{-}name \to RDB`RelationName$$
$$make\text{-}RelationName\ (e) \triangleq$$
$$\ldots;$$

$$make\text{-}attribute\text{-}name : RDB`AttributeName \to ET`attribute\text{-}name$$
$$make\text{-}attribute\text{-}name\ (an) \triangleq$$
$$\ldots;$$

$$make\text{-}AttributeName : ET`attribute\text{-}name \to RDB`AttributeName$$
$$make\text{-}AttributeName\ (an) \triangleq$$
$$\ldots;$$

$make\text{-}DataType : ET`attribute\text{-}def \to RDB`DataType$

$make\text{-}DataType\,(an) \triangleq$
  cases an :
    mk-$ET`entity\text{-}name\,(n) \to make\text{-}RelationName\,(n)$,
    mk-$ET`type\text{-}name\,(\text{"}Integer\text{"}) \to$ INTEGER,
    mk-$ET`type\text{-}name\,(\text{"}String\text{"}) \to$ STRING
  end

These are not specified in any more detail, except that they are all bijections, and have the following inverse properties.

$\forall\, en \in ET`entity\text{-}name \cdot make\text{-}entity\text{-}name\,(make\text{-}RelationName\,(en)) = en$
$\forall\, rn \in RDB`RelationName \cdot make\text{-}RelationName\,(make\text{-}entity\text{-}name\,(rn)) = rn$
$\forall\, an \in ET`attribute\text{-}name \cdot make\text{-}attribute\text{-}name\,(make\text{-}AttributeName\,(an)) = an$
$\forall\, an \in RDB`AttributeName \cdot make\text{-}AttributeName\,(make\text{-}attribute\text{-}name\,(an)) = an$

The definitions of operations in this model are generally straightforward, they break down the abstract structures and rebuild the relevant concrete structures. We give just a single example of an explicit version of the abstract operations defined on top of the relational database.

operations

$add\text{-}entity : ET`entity\text{-}name \times ET`entity\text{-}def \xrightarrow{o} ()$

$add\text{-}entity\,(en, ed) \triangleq$
  let $fields = \{$mk-$RDB`Field$
    $(make\text{-}AttributeName\,(an),$
    $make\text{-}DataType\,(ed\,(an))) \mid an \in$ dom $ed\}$ in
  $RDB`Create(make\text{-}RelationName\,(en), fields)$

Thus the design has the character of a programming task.

4.6 A Refinement Proof

In this section, we give a proof that the relational database based specification is a refinement of the abstract model. We restrict our interest to just one operation, *add_entity*. Note that in this section, for clarity, we omit the module prefixes.

4.6.1 The Retrieve Function

The refinement proof is based around the retrieve function, relating the concrete state to the abstract. In this case, the concrete state is the relational database model; the abstract the model of EXPRESS.

The retrieve function on the relational database state is broken down into the composition of retrieve functions on the constituent types in the state. This is only a

partial definition; we omit for example, all the details of the *retr_model* function, as this plays no part in this part of the refinement proof.

functions

$$retr : rdb \to db$$

$$retr\,(r) \triangleq$$
$$\quad \mathsf{mk}\text{-}db\,(retr\text{-}schema\,(r), retr\text{-}model\,(r));$$

$$retr\text{-}schema : rdb \to schema\text{-}def$$

$$retr\text{-}schema\,(r) \triangleq$$
$$\quad \mathsf{mk}\text{-}schema\,(retr\text{-}exp\text{-}types\,(r), retr\text{-}entities\,(r), retr\text{-}global\text{-}rules\,(r));$$

$$retr\text{-}entities : rdb \to (entity\text{-}name \xrightarrow{m} entity\text{-}def)$$

$$retr\text{-}entities\,(\mathsf{mk}\text{-}rdb\,(rdb)) \triangleq$$
$$\quad \{make\text{-}entity\text{-}name\,(rn) \mapsto retr\text{-}entity\text{-}def\,(rdb\,(rn)) \mid rn \in \mathsf{dom}\ rdb\};$$

$$retr\text{-}entity\text{-}def : Relation \to entity\text{-}def$$

$$retr\text{-}entity\text{-}def\,(\mathsf{mk}\text{-}Relation\,(s, \text{-})) \triangleq$$
$$\quad \{make\text{-}attribute\text{-}name\,(an) \mapsto retr\text{-}attribute\text{-}def\,(s\,(an)) \mid an \in \mathsf{dom}\ s\};$$

$$retr\text{-}attribute\text{-}def : Attribute \to attribute\text{-}def$$

$$retr\text{-}attribute\text{-}def\,(\mathsf{mk}\text{-}Attribute\,(dt)) \triangleq$$
$$\quad \mathsf{cases}\ dt :$$
$$\quad\quad RelationName \to \mathsf{mk}\text{-}entity\text{-}name\,(dt),$$
$$\quad\quad \text{INTEGER} \to \mathsf{mk}\text{-}type\text{-}name\,(\text{"}INTEGER\text{"}),$$
$$\quad\quad \text{STRING} \to \mathsf{mk}\text{-}type\text{-}name\,(\text{"}STRING\text{"})\ldots$$
$$\quad \mathsf{end}$$

Note that *retr_attribute_def* and *mk_DataType* are inverses.

Proofs about the retrieve function. Two obligations need to be proven about the retrieve function: *retr-S-adeq* (the is surjective) and *init-adeq* (the initial concrete state is retrieved to the initial abstract state). In this case these give rise to the following proof obligations.

$$\frac{d : db}{\exists\, r : rdb \cdot retr\,(r) = d}$$

$$\frac{d : db, r : rdb, d = retr\,(r), rinit\text{-}rdb\,(r)}{rinit\text{-}db\,(d)}$$

The second clearly holds: the retrieve function maps the empty map onto the empty map. The first requires the consideration of the invariants on the two states. This is more complicated, and is omitted here.

4.6.2 The Refinement Proof Obligations

Two proof obligations are required for each operation: *OP-dom-obl*, which demonstrates that the abstract precondition implies the concrete precondition, and *OP-res-obl*, which demonstrates that the abstract precondition and concrete postcondition together satisfy the abstract postcondition. Here, we only consider one operation, *add_entity*.

We first consider the obligation *add-entity-dom-obl*. This translates into the following:

$$\frac{\begin{array}{c} en : \mathit{entity\text{-}name}, \\ ed : \mathit{entity\text{-}def}, \\ c : \mathit{cedb}, \\ en \notin \mathsf{dom}\ (\mathit{retr}\,(c).\mathit{schema}.\mathit{entities}) \end{array}}{\mathit{make\text{-}RelationName}\,(en) \notin \mathsf{dom}\ (c.\mathit{db})}$$

Once definedness is established, the proof can proceed via forward reasoning, generating new hypotheses from the existing ones. The last hypothesis can be simplified by equal substitution using the *f-defn* rules for the retrieve functions:

$$\begin{array}{l} en \notin \mathsf{dom}\ (\mathit{retr}\,(c).\mathit{schema}.\mathit{entities}) \\ \quad = en \notin \mathsf{dom}\ (\mathit{retr\text{-}entities}\,(c)) \\ \quad = en \notin \mathsf{dom}\ \{\mathit{make\text{-}entity\text{-}name}\,(rn) \mapsto \mathit{retr\text{-}entity\text{-}def}\,(c.\mathit{db}\,(rn)) \\ \qquad\qquad\qquad \mid rn \in \mathsf{dom}\ (c.\mathit{db})\} \\ \quad = en \notin \{\mathit{make\text{-}entity\text{-}name}\,(rn) \mid rn \in \mathsf{dom}\ (c.\mathit{db})\} \end{array}$$

Since *make_RelationName* is an injection, we can apply this function to this last hypothesis, pushing it through the set comprehension giving a new hypotheses, which can be simplified as follows:

$$\begin{array}{l} \mathit{make\text{-}RelationName}\,(en) \notin \{\mathit{make\text{-}RelationName}\,(\mathit{make\text{-}entity\text{-}name}\,(rn)) \\ \qquad\qquad\qquad\qquad\qquad \mid rn \in \mathsf{dom}\ (c.\mathit{db})\} \\ \quad = \mathit{make\text{-}RelationName}\,(en) \notin \{rn \mid rn \in \mathsf{dom}\ (c.\mathit{db})\} \\ \quad = \mathit{make\text{-}RelationName}\,(en) \notin \mathsf{dom}\ (c.\mathit{db}) \end{array}$$

and the obligation holds. The second obligation *add-entity-res-obl* is more complicated, and we only give a proof sketch. The form of the obligation is as follows.

$$
\frac{\begin{array}{c} en\colon entity\text{-}name, \\ ed\colon entity\text{-}def, \\ c\colon cedb, \\ en \notin \mathsf{dom}(retr(c).schema.entities), \\ \mathsf{let}\ fields = \{ \\ \mathsf{mk}\text{-}Field(make\text{-}AttributeName(an), make\text{-}DataType(ed(an))) \\ \mid an \in \mathsf{dom}\ ed \\ \}\ \mathsf{in} \\ \mathsf{post}\text{-}Create(make\text{-}RelationName(en), fields) \end{array}}{\begin{array}{c} retr(c).schema.entities = retr(\overleftarrow{c}).schema.entities \dagger \{en \mapsto ed\} \wedge \\ retr(c).schema.exp\text{-}types = retr(\overleftarrow{c}).schema.exp\text{-}types \wedge \\ retr(c).schema.global\text{-}rules = retr(\overleftarrow{c}).schema.global\text{-}rules \end{array}}
$$

The fifth hypothesis of this obligation is complex. It is derived from the post-condition of the concrete *add-entity* operation. As this operation calls the *Create* operation, the post-condition of the latter operation is substituted.

We can split the goal into three, one for each conjunction. It is the first one which we are primarily interested in. Again, we proceed by using a forward proof. Most of the information required to prove this goal is encapsulated within the last hypothesis. This can be simplified as follows. We first expand out the post-condition of Create, leading to the following nested expression.

$$
\begin{array}{rl} \mathsf{let}\ fields = & \{ \\ & \mathsf{mk}\text{-}Field(make\text{-}AttributeName(an), make\text{-}DataType(ed(an)) \\ & \qquad \mid an \in \mathsf{dom}\ ed \\ & \}\ \mathsf{in} \\ \mathsf{let}\ rel = & \mathsf{mk}\text{-}Relation(Make\text{-}Scheme(fields), \{\})\ \mathsf{in} \\ & c.db = \overleftarrow{c}.db \dagger \{make\text{-}RelationName(en) \mapsto rel\} \end{array}
$$

By the equational congruence rule ($=$-*extend*), we can apply the retrieve function to both sides of this expression (the let clauses are omitted for clarity in the following proof steps):

$$
retr(c.db) = retr(c.db \dagger \{make\text{-}RelationName(en) \mapsto rel\})
$$

Expanding the definition of *retr* gives:

$$
\begin{array}{l} \mathsf{mk}\text{-}db(retr\text{-}schema(c.db), retr\text{-}model(c.db)) = \\ \quad \mathsf{mk}\text{-}db(retr\text{-}schema(\overleftarrow{c}.db \dagger \{make\text{-}RelationName(en) \mapsto rel\}), \\ \qquad retr\text{-}model(\overleftarrow{c}.db \dagger \{make\text{-}RelationName(en) \mapsto rel\})) \end{array}
$$

As it is the schema component we are primarily interested, we can use the selector definition rule *schema-defn* to give the equation:

$$
retr(c.db).schema = retr\text{-}schema(\overleftarrow{c}.db \dagger \{make\text{-}RelationName(en) \mapsto rel\})
$$

Again using the selectors, this can be decomposed into the following three hypotheses:

$$
\begin{array}{l}
retr\,(c.db).schema.entities = \\
\quad retr\text{-}schema\,(\overleftarrow{c}.db \dagger \{make\text{-}RelationName\,(en) \mapsto rel\}).entities \\
retr\,(c.db).schema.exp\text{-}types = \\
\quad retr\text{-}schema\,(\overleftarrow{c}.db \dagger \{make\text{-}RelationName\,(en) \mapsto rel\}).exp\text{-}types \\
retr\,(c.db).schema.global\text{-}rules = \\
\quad retr\text{-}schema\,(\overleftarrow{c}.db \dagger \{make\text{-}RelationName\,(en) \mapsto rel\}).global\text{-}rules
\end{array}
$$

We are only interested in the first of these which can be rewritten to the following hypothesis.

$$
\begin{array}{l}
retr\,(c.db).schema.entities = \\
\quad retr\text{-}entities\,(\overleftarrow{c}.db \dagger \{make\text{-}RelationName\,(en) \mapsto rel\})
\end{array}
$$

Now we can push *retr-entities* through the overriding since we know from the previous proof obligation that $make\text{-}RelationName(en) \notin \mathsf{dom}(old(c).db)$. This gives:

$$
\begin{array}{l}
retr\,(c.db).schema.entities = \\
\quad retr\text{-}entities\,(\overleftarrow{c}.db) \dagger retr\text{-}entities\,(\{make\text{-}RelationName\,(en) \mapsto rel\})
\end{array}
$$

Unfolding the definition of *retr-entities* gives:

$$
\begin{array}{l}
retr\,(c.db).schema.entities = \\
\quad retr\,(\overleftarrow{c}.db).schema.entities \dagger \\
\quad\quad \{make\text{-}entity\text{-}name\,(make\text{-}RelationName\,(en)) \mapsto retr\text{-}entity\text{-}def\,(rel)\}
\end{array}
$$

and unfolding again on both sides:

$$
\begin{array}{l}
retr\,(c).schema.entities = \\
\quad retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto retr\text{-}entity\text{-}def\,(rel)\}
\end{array}
$$

Unfolding the *retr-entity-def(rel)*, expanding the definition of *rel* as in the let clause, we can give the following steps:

$$
\begin{array}{l}
retr\,(c).schema.entities \\
\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto \\
\quad\quad retr\text{-}entity\text{-}def\,(make\text{-}Relation\,(Make\text{-}Scheme\,(fields), \{\}))\ \} \\
\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{\ en \mapsto \\
\quad\quad retr\text{-}entity\text{-}def\,(\mathsf{mk}\text{-}Relation(\\
\quad\quad\quad \{f.name \mapsto \mathsf{mk}\text{-}Attribute\,(f.type) \mid f \in fields\}),\ \{\})\} \\
\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{\ en \mapsto \\
\quad\quad \{\ make\text{-}attribute\text{-}name\,(an) \mapsto retr\text{-}attribute\text{-}def\,(att) \mid \\
\quad\quad\quad \mathsf{mk}\text{-}(an, att)^5 \in \{f.name \mapsto \mathsf{mk}\text{-}Attribute\,(f.type) \mid f \in fields\}\ \}\}
\end{array}
$$

Now, the definition of *fields* states that the first component (i.e. *f.name*) of each pair is *mk-AttributeName(an)* for $an \in \mathsf{dom}\ ed$, and thus performing this expansion gives:

$$
\begin{aligned}
&retr\,(c).schema.entities \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{\ en \mapsto \\
&\qquad \{\ make\text{-}attribute\text{-}name\,(an) \mapsto retr\text{-}attribute\text{-}def\,(att)\ \mid \\
&\qquad\quad \mathsf{mk}\text{-}(an, att) \in \{f.name \mapsto \mathsf{mk}\text{-}Attribute\,(f.type) \mid f \in \\
&\qquad\quad \{\ \mathsf{mk}\text{-}Field\ (\ make\text{-}AttributeName\,(an), \\
&\qquad\qquad\qquad make\text{-}DataType\,(ed\,(an))\ )\ \mid an \in \mathsf{dom}\ ed\}\}\}\} \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto \\
&\qquad \{make\text{-}attribute\text{-}name\,(an) \mapsto retr\text{-}attribute\text{-}def\,(att)\ \mid \\
&\qquad \mathsf{mk}\text{-}(an, att) \in \{\mathsf{mk}\text{-}(make\text{-}AttributeName\,(an), \\
&\qquad\quad make\text{-}DataType\,(ed\,(an)))\mid an \in \mathsf{dom}\ ed\}\}\} \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto \\
&\qquad \{\ an \mapsto retr\text{-}attribute\text{-}def\,(att) \mid \mathsf{mk}\text{-}(an, att) \in \\
&\qquad \{\mathsf{mk}\text{-}(an, make\text{-}DataType\,(ed\,(an))) \mid an \in \mathsf{dom}\ ed\}\ \}\ \} \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{\ en \mapsto \\
&\qquad \{\ an \mapsto retr\text{-}attribute\text{-}def\,(make\text{-}DataType\,(ed\,(an)))\ \mid \\
&\qquad\quad an \in \mathsf{dom}\ ed\}\}
\end{aligned}
$$

retr-attribute-def is the inverse of *make-DataType*, and so:

$$
\begin{aligned}
&retr\,(c).schema.entities \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto \{an \mapsto ed\,(an) \mid an \in \mathsf{dom}\ ed\}\} \\
&\quad = retr\,(\overleftarrow{c}).schema.entities \dagger \{en \mapsto ed\}
\end{aligned}
$$

and the proof is complete.

4.6.3 Thoughts on the Refinement Proof

This example is of a "medium-complexity" proof, but typical of the style required to discharge such obligations. The proof of the obligations for the *add-instance* operation are more complex, due to the necessity to ensure the integrity of the data.

Much of the complexity of the proofs is contained within the retrieve function. The proof of the surjectivity of the retrieve is more problematic, due to the recursive definitions used.

It is striking that this task could be described as "low-level theorem proving". Most of the proof steps are small and relatively trivial: unfolding of function definitions and let clauses; extensive simple equational reasoning; and the application of retrieve

[5] In this proof sketch, $\mathsf{mk}\text{-}(x, y) \in m$ is being used as a shorthand for $x \in \mathsf{dom}\ m \wedge y = m\,(x)$.

function to generate new equational congruences. However, there are a large number of these steps, even in a modest proof such as this one. The steps are tedious, with substitutions and expansions into large, unwieldy terms, and are thus are highly prone to human error.

This emphasises the need for appropriate machine support. Proofs of this type are needed for each of the operations, with many such low-level reasoning steps, which are prime candidates for automation, and many repeated chains of reasoning, calling for the construction of repositories of reusable lemmas.

4.7 General Experiences and Conclusions

Concurrent engineering of the abstract specification, refinement and implementation worked well. A major task was the understanding of the SDAI standard; by working on specification and implementation together, the abstract model in the standard and the language bindings were both considered which was a help when trying to interpret the standard.

No routine technique was found to develop a VDM model from an EXPRESS one. No uniform approach was appropriate for object identifiers or subtypes, rather it was necessary build the VDM model as dictated by an understanding of the overall specification. Unsurprisingly perhaps, it is not appropriate to try to use EXPRESS style in VDM!

The implicit dereferencing available in EXPRESS has both benefits and drawbacks. In its favour is that many concise forms can be defined for particular situations, on the other hand, it can overly complicate the data to require an indirection every time a tuple is created. It is unfortunate that there is no other way of constructing tuples in EXPRESS.

The IFAD VDM-SL Toolbox was invaluable for writing, typesetting and typechecking the specifications. There is no doubt that the task would have been more time consuming and error prone without it. Many errors were found in the specification whilst it was developed and some minor ones in the previously published module. The animation and testing facilities available in the toolbox were not used.

The Toolbox also provides a facility for generation of C++ code which could be very useful. Code generated from the specification of the "EXPRESS shell" could form the basis of an actual implementation of the software. However, if this were to be used, it would have to be integrated (and maintained) with other software which would interface to the relational database and the EXPRESS application. Although this was not attempted, the documentation for the code generator describes the structure of the resulting code. The present authors were surprised to find that the code generation process builds classes according to the type constructors of VDM rather than the data structures of the application data model. It could make the integration task unnecessarily difficult. Similar problems will of course be present with any automatic code generation tool [12].

Although, the formalisation of the refinement and proofs were not completed in full detail, it was certainly worthwhile developing abstract and concrete views of the system. Having two specifications helped one resist the temptation to gradually decrease the level of abstraction as the specification developed. Refinement and proof would both require considerable effort, particularly as neither aspect is supported by the Toolbox.

Certain pragmatic decisions also had to be taken as to what aspects to focus on and what not to formalise. Thus, although the formal specification has helped enormously in the understanding of the system and its design, in this instance it has done little to increase confidence in correctness of the implementation.

4.8 Bibliography

[1] STEP Overview, ISO 10303, Part 1.

[2] SDAI Specification, ISO 10303, Part 22. TC184/SC4/WG7.

[3] *ibid.* Section 6 to 8.

[4] Semantic Unification Meta Model. ISO 10303.

[5] A Formal Semantics For SQL. Meira, S., Motz, M. and Tepedino, F. Intern. J. Computer Math. Vol. 34. pp. 43-63 (1990)

[6] Schenck and Wilson. Information Modelling the EXPRESS way, Oxford University Press, 1995.

[7] *ibid.* Appendix E.

[8] Analysis of the STEP Standard Data Access Interface Using Formal Methods. Botting, R.M. and Godwin, A.N. Computer Standards and Interfaces. 17(5-6), pp. 437-456, North Holland, 1995.

[9] Bicarregui J.C., An evaluation of methods for generating SQL from EXPRESS. ESPRIT 6212, Processbase, Document RAL/6212/TNR/016/4. January 1995.

[10] Bicarregui, Ritchie and Haughton, Experiences in Using Abstract Machine Notation in a GKS Case Study. FME'94: Industrial Benefit of Formal Methods, LNCS 873, Springer-Verlag, 1994.

[11] Vienna Development Method - Specification Language, Draft International Standard, ISO/IEC DIS 13817-1, 1995(E).

[12] Bicarregui, Dick and Woods, Supporting the length of formal development: form diagrams to VDM to B to C, 7th International Conference on Putting into Practice Methods and Tools for Information System Design. IBSN: 2-906082-19-9 October 1995.

[13] The CAESAR STEP Toolkit User manual (Version 1.3). Caesar Systems Limited, June 1992.

[14] The EXPRESS Language Reference Manual, ISO IS 10303-11 : 1994(E).

[15] Translating Express to SQL: A User Guide. Morris, K.C., NISTIR 90-4341, National Institute of Standards and Technology (NIST), Gaithersburg MD., 20899. USA.

[16] STEP Relational Interface, Raghaven, V., Hardwick, M., Rensselaer Polytechnic Institute, Computer Science Masters Project. (1993)

[17] Thomas D, Implementing the emerging ISO Standard STEP into a relational database, BNCOD-8, Proceedings of the 8th British national Conference on Databases.

[18] P. Clement, Internal Report on the EXPRESS to SQL Compiler. Loughborough University of Technology Technical Report. 1991.

[19] The IFAD VDM-SL Toolbox, in Woodcock and Larsen (Eds.). *FME'93: Industrial Strength Formal Methods*. Springer-Verlag, 1993.

[20] C.B. Jones C.D. Allen, D.N.Chapman. A formal definition of algol 60. Technical Report 12.105, IBM Laboratory, Hursley, Aug. 1972.

[21] D. Andrews and W. Henhapl. Pascal. In *Formal Specification and Software Development*, chapter 7, pages 175–252. Prentice-Hall, 1982.

[22] S.P.A. Lau, Derek Andrews, Anjula Garg and J.R. Pitchers. The Formal Definition of Modula-2 and Its Associated Interpreter. In L.M̃arshall R.B̃loomfield and R. Jones, editors, *VDM '88 VDM – The Way Ahead*, pages 167–177. VDM-Europe, Springer-Verlag, September 1988.

[23] W.Henhapl C.B.Jones P.Lucas H.Bekić, D.Bjørner. A formal definition of a pl/i subset. Technical Report 25.139, IBM Laboratory, Vienna, December 1974.

[24] O.Oest (eds.) D. Bjørner. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[25] J.F. Nilsson. Formal Vienna-Definition-Method models of Prolog. In J.A. Campbell, editor, *Implementations of PROLOG*, pages 281–308. Ellis Horwood Series: Artificial Intelligence, 1984.

Index

accountability, 67
Ammunition Control System, 31

British Nuclear Fuels, 2

circular reasoning, 61
class-centered model, 109
confidentiality, 67
consistency proof, 84
counter-example, 13, 19

emergent property, 24
environmental precondition, 78
exception condition, 79
EXPRESS, 95

formation property, 42
fully formal proof, v, vi, 11, 27, 92

genericity, 26

higher order logic, 159, 195

IFAD VDM-SL Toolbox, 2, 25, 31, 95, 119, 191, 192
indirection, 99, 108, 119
information modelling, 96
instance-centered model, 109
integrity, 67, 118
Isabelle, 191, 193

levels of rigour, 11
Logic of Partial Functions, 192
looseness, 185
LPF, 192

memory model, 133–135, 137–139, 143–146, 148, 149, 151, 152
memory order, 124, 125, 127–132, 143, 146
Ministry of Defence, 32
modules, 26, 31, 50–53, 97, 98, 102, 105, 112
MSMIE, 169
Multiprocessor Shared-Memory Information Exchange, 169
Mural, 92, 124, 133, 145, 146, 148, 149, 151, 152, 192, 209, 217

non-determinism, 185

OBJ3, 31
object identifier, 99, 108

partiality, 184, 192
partitions, 68
per processor program order, 126, 127, 139
precondition for success, 79
program order, 124–128, 132, 138–140, 148
PVS system, 157

reachable states, 24, 171
refinement, 95, 97, 112, 113, 152, 178, 181, 182, 184–186
refinement proof, 113–115, 118
retrieve function, 113–116, 118, 152, 181, 182
rigorous proof, 11, 19

safety analysis, 9
safety requirement, 6
satisfiability proof obligation, 11–13, 15, 86
schema-centred model, 108
seals, 69
security enforcing functions, 70
security policy model, 65
security properties, 78, 85

shared memory system, 123
SpecBox, 2, 25
SQL, 95
STEP standard, 96
system safety, 5

tactics, 198
TCC, 158, 175
testing, 27, 50
textbook proof, 11, 15
theory of VDM primitves, 133
Transport of Dangerous Goods, 32
trusted gateway, 201
trusted path, 70
typechecking constraints, 175

UN regulations, 32, 34, 50, 51
undefined, 184, 192
underspecification, 185
uniprocessor correctness, 124, 128, 144

value condition, 129, 144, 146
VDM-LPF, 195

witness, 16, 39–43, 48, 49