Formal Aspects of Computing (1992) 4: 100–142

Formal Aspects of Computing

Modularising the Specification of a Small Database System in Extended ML[1]

Edmund Kazmierczak

Department of Computer Science, University of Edinburgh, Edinburgh

Keywords: Extended ML; Specification; Modular program development; Parameterised specifications; Parameterised programs

Abstract. A case study in the modular specification and refinement of a small database system is presented in Extended ML. Two similar requirements specifications are given and a program development step from each these is presented. The structure resulting from the first program development step is similar to that given in [FiJ90] and is presented as an answer to the challenge problem given in that paper, while the second development step is presented as a possible alternative which is more suited to the Extended ML style of program development. In the context of these two development steps the module facilities of Extended ML, their role in specification and program development and their ability to meet the challenge of [FiJ90] are examined.

1. Introduction

Formal program specifications serve many purposes in software engineering, for example, in defining precisely what a program must do (but not how it must do it) in order to solve a particular problem, or in the detailed design of a program module. The goal of program development is then to provide a program which will *meet* the specification (and by doing so solve the original problem).

Specifications, just as programs, can be large and unwieldy making them difficult to understand or reason about. Specification languages with facilities for structuring specifications have been developed to cope with the large specifications which may arise in practice (see for example [GoB80, EhM85, EhM90, FiJ90,

[1] This research was supported by SERC grant GR/E 78463

Correspondence and offprint requests to: Edmund Kazmierczak, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, UK. email : eka@lfcs.ed.ac.uk

San89]). From a clean structuring of a specification we may, for example, see the consequences of our definitions and axioms more readily. This is not only important from the perspective of the specification writer but also of the many parties who may come to rely on the specification during the course of a system's lifetime. In addition, module facilities enhance the flexibility of a specification language by limiting the scope of the consequence of changes, or by facilitating the re-use of previously developed modules in different contexts.
Two proposed goals of a module system for structuring specifications are (see [Par72, FiJ90, EhM90]):

1. *Separation of Concerns*, by which we mean the ability to concentrate on the specification of one aspect of a system without being hindered by details of irrelevant parts of the same specification and, in the context of program development, the ability to develop modules without reference to other parts of the same system;
2. *Module Generality*, by which we mean that modules may be defined with sufficient generality to enable them to be used in a variety of contexts.

A means of achieving (1) is to surround each module with well defined interfaces which isolate it from any context in which it may be used. Interfaces around each module are also important in defining the precise nature of the interaction between the module and its environment [SST90]. An important corollary of this is that the structure of a specification can be used to limit the search space for proofs of theorems about that specification (see, for example [SaB83, SaT88] where theorem proving in structured specifications is addressed).

Module generality should provide for the re-use of modules in a variety of contexts. This is important not only for the sake of convenience but also since time spent on getting one module "right" should not be wasted in redeveloping what is essentially the same specification again in every new context (with all the attendant possibility of error).

In [FiJ90] a case study in the modular specification of a simple database system was presented. The specification language used was VDM [Jon86] augmented with a number of facilities for writing modular VDM specifications. In particular parameterised modules, nested modules and dependent types were used to create *parameterised specifications*. The paper concluded with a challenge for other specification languages which provide facilities for structuring specifications to specify the database using the same structure as in [FiJ90][2].

The aim of this paper is then twofold: first, to answer the challenge problem by performing the case study using Extended ML [SaT89, San89] and second, to examine the ability of Extended ML's module system to meet this challenge in the light of the two general goals outlined above. Extended ML has many of the module facilities mentioned above but in Extended ML they play a dual role in both structuring specifications and the resulting programs (or combinations of the two). In particular in Extended ML we wish to specify *parameterised programs* and formally develop Standard ML programs from these. We present two similar requirements specifications and perform a single program development step, using the Extended ML methodology, from each of these. The first of these results in an analogue of the structure of the specification in [FiJ90] while the second is given in a style more suited to program development in Extended ML.

[2] See [FiJ90], appendix C

The remainder of the paper is organised as follows. In section 2 we briefly review Extended ML by giving some simple examples while in section 3 we give an overview of the database system that forms the subject of our case study. In section 4 we present the first requirements specification in some more detail as the specification of a parameterised program which will eventually implement the data base. The two different modular specification structures are then investigated by performing a single program development step from this requirements specification and from the second similar requirements specification in section 6. The first development step given in section 5, uses *coding* to give a similar structure to that in [FiJ90][3] while the second is presented in section 6 and uses *functor decomposition* to achieve an alternative structuring of the specification. Our conclusions are presented in section 7.

2. Extended ML

Extended ML is a wide-spectrum language for specifying and developing parameterised programs in the functional programming language Standard ML [Har86, Tof89, HMT90, MiT90]. Standard ML is a functional programming language with the ability to define data types by just giving the constructors for that type, polymorphic data types [Mil77, DaM82, CaW85] and higher order functions as well as providing a powerful module system for "programming in the large". It also has a completely formal mathematical definition [HMT90, MiT90] which makes it an extremely good target language for program development of the kind which requires formal proof.

Extended ML is an extension of Standard ML obtained by allowing axioms in modules and module interfaces and using the modules facilities already present in Standard ML to structure specifications [San89, SaT89]. Specifications in Extended ML are of *parameterised program* modules, rather than the more usual parameterised specifications (of programs) found in other algebraic specification languages, for example, [GoB80, EhM85, EhM90]. Program development is by stepwise refinement (described below) from algebraic specifications of a (parameterised) module's interfaces. We briefly overview the salient features of both the Extended ML specification language and the program development methodology below.

2.1. Structures, Functors and Signatures

Specifications in Extended ML are written in a higher order, polymorphic, equational logic and are structured using the modules system of Standard ML [HMT90, MiT90, Har86, Mac86, Tof89] which is composed of *Signatures, Structures* and *Functors. Structures*, in Standard ML, are program modules which contain definitions of types, functions and substructures. *Signatures*, in Standard ML, are interfaces to structures which specify what components of a structure are externally visible. In Extended ML, signatures and structures may include axioms which makes them specifications, for example, a signature `PO` specifying a partial order is given in Fig. 1.

[3] See [FiJ90] appendix C

```
signature PO =
  sig
    eqtype elem
    val le : elem * elem -> bool
    axiom forall x => le(x,x)
    axiom forall x => forall y =>
                  le(x,y) andalso le(y,x) implies x=y
    axiom forall x => forall y => forall z =>
             le(x,y) andalso le(y,z) implies le(x,z)
  end;
```

Fig. 1. A signature specifying a partial order

A structure which *matches* `PO` must include at least a type called `elem`, which must have equality defined on it[4] (as required by the specification `eqtype`) and a function `le` with the type `elem * elem -> bool`. Two examples of structures which will match the signature `PO` of Fig. 1 are given Fig. 2. The first uses the predefined type of integers and the predefined operation `<=` (less than or equal to) defined on integers to give a structure matching `PO`. The second uses natural numbers which are generated by a data type definition: each member of the type `elem` is either a term `ZERO` or `SUC(x)` where `x` is a term of type `elem`. The functions `le` and `plus` in the structure `Element'` are both defined by cases on the data type `elem`. Note that we can have more components in a structure which matches `PO` than is required by `PO`, for example, `plus` in the structure `Element'`, but the extra components are *hidden* by the signature.

As well as just *flat* signatures like the partial order in Fig. 1 signatures may exhibit internal structure which includes local (and therefore hidden) functions. Signatures may refer to other signatures which are to be *included* or to substructures whose visible components are specified by yet another signature. Substructures and locally specified functions are present in the signature `SORT` of Fig. 3. The local functions are `permutation`, `member` and `ordered` and are only used in the signature but are not required of any structure matching this signature: they are *local* to the signature `SORT`.

Note also that in Fig. 3 quantifiers may range over *polymorphic* types. Polymorphic types are distinguished by a leading quote, for example, the type `'a list` in Fig. 3 is a polymorphic type. A function with polymorphic types in its domain, such as `member` in Fig. 3, may be applied to arguments of many different types. For example, `member` may be applied to pairs of type `int * (int list)` or `bool * (bool list)` but no matter what type of argument is supplied to `member` the specification is the same. Polymorphic types which which admit equality (equality types) are distinguished by a double leading quote, for example, the type `''a Set` in appendix A.1.

Functors are parameterised modules. They are specified by two signatures, one for the parameter and one for the result. The parameter signature specifies the class of Standard ML structures which can be actual parameters to the functor while the result signature specifies the class of Extended ML structures which

---

[4] Not all types in Standard ML have equality defined on them, for example, function types do not have an equality [HMT90]. Those types which do have equality defined on them are called *Equality Types* [HMT90, MiT90]

```
structure Element:PO =
  struct
     type elem = int
     val  le = op <=
  end;

structure Element':PO =
  struct
    datatype elem = ZERO | SUC of elem

    fun le(ZERO,ZERO) = true
      | le(ZERO,SUC(x)) = true
      | le(SUC(x),ZERO) = false
      | le(SUC(x),SUC(y)) = le(x,y)

    fun plus(ZERO,x) = x
      | plus(SUC(x),y) = SUC(plus(x,y))
  end
```

Fig. 2. Two structures which match the signature `PO`

can result and this may depend upon the actual parameter. Intuitively Extended ML functors can be thought of as functions from Standard ML structures to Extended ML structures. For example, a sorting functor may be presented as in Fig. 4. The phrase `include SORT` again means that the resulting interface in Fig. 4 includes all the declarations and axioms of `SORT`. The sharing constraint `sharing X = Elements` states that the substructure `Elements` of `Sort` must be identical to the parameter `X` and is similar to the sharing constraints of Standard ML.

Sharing is important in Standard ML because it is required in deducing the correct types in modules. It is important in Extended ML because it is often used to express the dependence of axioms in the result signature on types and values in the actual parameter. In Fig. 4 for example, the sharing constraint specifies that the type `Elements.elem` in `SORT` is the same as the type `X.elem` in the (actual) parameter thus making the result *dependent* upon the actual parameter. Without this sharing constraint the type `Elements.elem` and the value `Elements.le` need not be the same as those of the parameter `X`, and so the axioms would not explicitly require us to sort lists of type `X.elem` nor compare their values using the partial order `X.le` in the parameter.

2.2. Constructing Standard ML Programs

One proceeds from a requirements specification to a program by a series of development steps. Each development step results in a program which is *correct* (in the sense described below) with respect to the results of the previous development step if all the proof obligations associated with that step are *formally* discharged. We may think of each development step as filling in some detail left open in the previous step, for example, making an abstract type within a structure concrete, or providing an algorithm for some function which hitherto has only been specified using axioms. Once the results of a development step includes no axioms, all

```
signature SORT =
  sig
    structure Elements : PO

    val sort : Elements.elem list -> Elements.elem list

    local
      val count       : 'a * 'a list -> int
      and permutation : 'a list * 'a list -> bool
      and ordered     : Elements.elem list -> bool

      axiom forall x : 'a => count(x,nil) = 0
        and forall x : 'a =>
            forall l : 'a list =>
              x = y implies
              count(x,y::l) = 1 + count(x,l)
        and forall x : 'a =>
            forall l : 'a list =>
              not(x = y) implies
              count(x,y::l) = count(x,l)

      axiom forall x : 'a =>
            forall l : 'a list =>
            forall l' : 'a list =>
              count(x,l) = count(x,l')
              implies permutation(l,l')

      axiom forall a : Elements.elem =>
              ordered(a :: nil) = true
        and forall a : Elements.elem =>
            forall b : Elements.elem =>
            forall l : Elements.elem list =>
              ordered(a::b::l) = (Elements.le(a,b))
              andalso ordered(b::l)
    in
      axiom forall l : Elements.elem list =>
              permutation(l,sort(l))
              andalso ordered(sort(l))
    end
  end;
```

Fig. 3. A signature with substructures and hidden functions

```
functor Sort(X:ELEMENT):sig
                          include SORT
                          sharing X = Elements
                        end = ?
```

Fig. 4. Specification of a Sorting Functor

the types are concrete and all the functions are defined by Standard ML code then the development process is complete. If all the proof obligations have been discharged then this final program satisfies the original requirements specification by *construction.*

There are three possible kinds of development step in the Extended ML program development methodology [San89].

Functor Decomposition

Intuitively functor decomposition is used to break a task into subtasks. Suppose we are given the following specification:

$$\texttt{functor F}(\texttt{X} : \Sigma) : \Sigma' = ?$$

The first of the development steps allows us to define the functor F in terms of the composition of a number of other functors, for example, in the simple case of two new functors G and H we have:

$$\texttt{functor F}(\texttt{X} : \Sigma) : \Sigma' = \texttt{G}(\texttt{H}(\texttt{X}))$$

where

$$\texttt{functor G}(\texttt{Y} : \Sigma_{\texttt{G}}) : \Sigma'_{\texttt{G}} = ?$$
$$\texttt{functor H}(\texttt{Z} : \Sigma_{\texttt{H}}) : \Sigma'_{\texttt{H}} = ?$$

and $\Sigma_{\texttt{H}}$, $\Sigma'_{\texttt{H}}$, $\Sigma_{\texttt{G}}$ and $\Sigma'_{\texttt{G}}$ are all appropriately defined Extended ML signatures. The task of finding a solution to F has been broken up into the subtasks of finding solutions to G and H. This decomposition is *correct* if:

1. All structures matching the parameter signature of F also match the parameter signature of H, that is, $\Sigma \models \Sigma_{\texttt{H}}$;
2. All structures matching the result signature of H can be used as an argument for G, that is, $\Sigma'_{\texttt{H}} \models \Sigma_{\texttt{G}}$;
3. All structures matching the result signature of G also match the result signature of F, that is, $\Sigma'_{\texttt{G}} \models \Sigma'$.

The development of the functors H and G may now proceed separately.

Coding

Given a specification of the form:

$$\texttt{structure A} : \Sigma = ?$$

or

$$\texttt{functor F}(\texttt{X} : \Sigma) : \Sigma' = ?$$

coding is used to replace the `qmark` by an actual structure body to give

$\texttt{structure A} : \Sigma = \mathit{strexp}$

or in the case of functors

$\texttt{functor F(X} : \Sigma) : \Sigma' = \mathit{strexp}$

A *coding* development step is *correct* if

$\mathit{strexp} \models \Sigma$

in the case of structures and

$\Sigma \cup \mathit{strexp} \models \Sigma'$

in the case of functors. A structure body need not be all Standard ML code and indeed the possibility of fixing only some design details exists since axioms are allowed within Extended ML structure bodies. For example, a value which is left specified in this way can be written as `val v =?`, (or in the case of functions `fun f(x) =?`) while types may be made abstract within structure bodies by writing `type t =?`. Axioms may also be added to specify more detailed properties of such unrefined values.

Refinement

Refinement is the third kind of development step used to fill in design choices left open by a coding step or by another refinement step. Refinement is most often used in choosing concrete types for abstract types or in filling in the details of a function with an actual algorithm. Given a functor of the form:

$\texttt{functor F(X} : \Sigma) : \Sigma' = \mathit{strexp}$

we can replace *strexp* by *strexp'* in a refinement step to give:

$\texttt{functor F(X} : \Sigma) : \Sigma' = \mathit{strexp}'$

A refinement step is *correct* if

$\Sigma \cup \mathit{strexp}' \models \mathit{strexp}$

The rules for coding structures are similar.

3. The "Non-Programmer Database"

The "Non-Programmer Database" (NDB) which forms the subject of the challenge problem is a simple existing database system described in [FiJ90, Wal90, WiS79]. The salient features of the NDB system are given below.

The data base stores information about *entities* and (binary) *relations* between them. Each entity is identified by a unique entity identifier (Eid) and is usually associated with a *value* (although this need not be the case). Entities (and their values) are grouped into entity sets (Esetnm) for the purpose of imposing constraints. Relations in the database are binary relations between two sets of entities and may be named or unnamed [FiJ90, Wal90]. In addition each relation has an associated pair of *entity set* names specifying the domain and codomain of that relation as well as information stating the kind of relationship which exists between the two sets of entities, whether one to one, many to one, one to many or many to many. This latter information is referred to as the *functional dependency* information of a relation (see [Dat86] for more about relational databases).

To maintain the integrity of the data the following two constraints are imposed on the database.

1. Sets of tuples within a relation must respect the functional dependency of that relation. This is a constraint on relations and is referred to in the sequel as the "functional dependency constraint".
2. The first and second components of a tuple in a relation must be drawn from the entity sets named by the domain and codomain of the relation. This is a form of *Typing* constraint placed on relations and is referred to in the sequel as the "type checking constraint"[5].

An example taken from [Wal90] is the following relation:

Country	Currency
Scotland	pound
China	yuan
Australia	dollar

which is a relation between the two entity sets *Country* and *Currency* and entities with values Scotland, China and Australia (each drawn from the *Country* entity set) and pound, yuan and dollar (each drawn from the *Currency* entity set).

Finally there are the operations which update the database, `ADDES`, `DELES`, `ADDENT`, `DELENT`, `ADDTUP`, `DELTUP`, `ADDREL` and `DELREL`. `ADDES` is used to add a new entity set name to the database and `ADDENT` adds a new entity identifier to each one of a number of entity sets. `ADDREL` and `ADDTUP` are used to add relations and tuples respectively to the database. The remaining operations `DELES`, `DELENT`, `DELREL` and `DELTUP` delete various elements from the database, for example, `DELES` deletes an entity set name and `DELENT` deletes an entity identifier.

4. A Specification of the Programming Task

The requirements specification for a parameterised version of NDB is outlined below while the full specification is given in appendix A. Specifications are given in an algebraic style. For the sake of brevity we omit quantifiers in the specifications and assume that all axioms are universally quantified outermost over their free variables unless otherwise stated. The specification which is the result signature of the functor implementing NDB can be naturally broken up into several substructures, one for the basic objects, a second for relations and a third for the update operations.

The four basic sets of objects in the data base, entity identifiers, entity set names, relation names and values, are specified by the four (abstract) types `Eid`, `Esetnm`, `Rnm` and `Value` respectively in the signature `BASICS` of Fig. 5. Each of these must admit equality as designated by the `eqtype` keyword.

The signature `BINARY_RELATION` in appendix A.2 introduces two abstract

[5] In [FiJ90] two more constraints on the database are given but these are concerned with the properties of maps which do not feature in our axiomatic specification

```
signature BASICS =
  sig
    eqtype Eid and Esetnm and Value and Rnm
  end
```

Fig. 5. Basic Types

```
signature BINARY_RELATION =
  sig

    include BASICS
    include SET

    . . .

  end;
```

Fig. 6.

types, `Tuple` and `BinaryRel`, as well as the operations for these. A third concrete data type `Maptp` is also introduced for the purposes of handling the functional dependency information for binary relations. This signature also includes two substructures: `B`, a substructure for the basic objects, and `S` a substructure for sets. To refer to components of these we prefix the identifiers in the signature with the name of the structure to which they belong, for example, `B.Eid` is used to refer to the type `Eid` in the substructure `B`. Also since relations can be named or unnamed two operations are used to construct new relations, one for anonymous relations and the other for named relations:

```
val mk_rel  : Maptp * B.Esetnm * B.Esetnm -> BinaryRel
val mk_rel' : B.Rnm * Maptp * B.Esetnm * B.Esetnm
               -> BinaryRel
```

Using substructures to structure a signature, such as `BINARY_RELATION` means that the final program will need to contain substructures matching the signatures `BASICS` and `SET` respectively. An alternative would have been to `include` the signatures `BASICS` and `SET` as in Fig. 6. We have used substructures, however, because they allow us to specify some necessary sharing later (see section 5).

Note also that the types and functions in the signature `SET` (of appendix A.1) are *polymorphic* which means that the operations specified there can be applied to arguments of many different types. Polymorphic types provide one way of creating signatures whose components can be re-used in a variety of contexts, for example, `SET` is one such signature which is used in (at least) two different ways: to specify sets of tuples in `BINARY_RELATION` (see appendix A.2) and to specify sets of entity set names in `NDB` (see appendix A.3).

The signature `NDB` now introduces the remaining update operations. To impose similar constraints on the final program to those in [FiJ90] several hidden auxiliary operations are introduced, for example, `esm`, `em` and `rm` (see appendix A.3). These auxiliary functions are used to capture the "state" of the database in our algebraic specifications much as the maps *esm*, *em* and *rm* define the state in the VDM specification of [FiJ90]. In the VDM specification *ADDES* is specified as in Fig. 7 while using the auxiliary functions the Extended ML analogue would be:

$ADDES(es{:}Esetnm)$
ext wr $esm : Esetnm \xrightarrow{m} Eid\text{-set}$
pre $es \notin \mathrm{dom}\ esm$
post $esm = \overleftarrow{esm} \cup \{es \mapsto \{\}\}$

Fig. 7.

```
axiom not(Es_in(es,ndb))
      implies
      (Es_in(es,ADDES(es,ndb))
       andalso esm(es,ADDES(es,ndb))
               = Binary.S.empty_set
       andalso
         (forall eid : Binary.B.Eid =>
             Eid_in(eid,ndb) implies
             em(eid,ADDES(es,ndb)) = em(eid,ndb))
       andalso
         (forall es'  : Binary.B.Esetnm =>
          forall es'' : Binary.B.Esetnm =>
             Es_in(es',ndb)
             andalso Es_in(es'',ndb)
             andalso Rel_in(es',es'',ndb)
             implies rm(es',es'',ADDES(es,ndb))
                     = rm(es',es'',ndb)))
```

The only question which now remains is where to formulate the two database constraints. The "functional dependency constraint" is a property of relations in the database and since it depends only upon the types and operations pertaining to relations it is given in the signature dealing with relations. The placement of the type checking constraint influences the structure of the program design specifications which we give. In section 5 it is a constraint on N-ary relations while in section 6 it is to be a constraint on relations in the database (but not necessarily on N-ary relations).

The first requirements specification is now given in Fig. 8. The sharing constraint again states that the substructure `Binary.B` of the final functor must be the same as the formal parameter of the functor: in other words, the types `Eid`, `Esetnm`, `Value` and `Rnm` appearing in the output signature must be the same as in the actual parameter of the functor.

5. Modularisation with Typing and Functional Dependencies

Recall from section 4 that the requirements specification in Fig. 8 is of a parameterised program (and not a parameterised specification). Below we give the first of our *program design* specifications which is obtained by *coding* from this requirements specification. The structure of our program design specification is

```
functor Ndb(B : BASICS):
            sig
              include NDB
              sharing B = Binary.B
            end = ?
```

Fig. 8. The specification of the database module to be developed

intended to be an algebraic analogue of that given in [FiJ90] where the type checking constraint is imposed on N-ary relations.

5.1. Generalising Binary Relations

The observation in [FiJ90] that binary relations are just a special case of N-ary relations is used to motivate a specification module for N-ary relations. In terms of our parameterised program specifications these modifications can be stated as follows.

1. Relations are now to be considered as N-ary relations for some fixed but arbitrary N. The elements of the type `Attr` are the acceptable field names of tuples. A functional correspondence between elements of the type `Attr` and values (Eids) defines a tuple.
2. The functional dependency information must also be generalised appropriately. In appendix B.2 the type `Norm` is used for this information. The means of constructing values of this type is through the function `mk_norm` where the domain of `mk_norm` is a type `(Attr S.Set * Attr) S.Set`. If `(s,f)` is an element of type `(Attr S.Set * Attr) S.Set` then `s` is to be thought of as a set of attributes which *functionally determine* the attribute `f`.
3. Tuples in relations are to satisfy the type checking constraint and this is to be a property of the module for N-ary relations.

In [FiJ90] three parameterised specification modules are used:

1. *TYPED-RELATION* which encapsulates the specification for N-ary relations;
2. *NDBRELATION* which specialises the specification of N-ary relations in *TYPED-RELATION* to a specification of binary relations;
3. it NDB which is a specification of the database based upon the specification module *NDBRELATION* which introduces binary relations and operations on binary relations.

To achieve a similar program design specification structure we use three functors which correspond in broad terms to the specification modules above:

1. `Typed_Relation` which is the module for N-ary relations is specified in Fig. 9;
2. `NDB_Relation` which is a functor that specialises N-ary relations to binary relations is specified in Fig. 10 (see appendix C.2)
3. `Ndb` which is the database module (see appendix C.3).

One feature of the module system in [FiJ90] is that theories can be dynamically created by passing parameters to specification modules, for example, in Fig. 11 a

```
functor Typed_Relation(E : ESM) : sig
                                   include TYPED_RELATION
                                   sharing E = ESM
                                  end = ?
```

Fig. 9. Requirements Specification for TYPED_RELATION

```
functor NDB_Relation(Tpm : TPM) :
  sig

    include BINARY_RELATION

    (*  Type Checking Constraint  *)

    axiom Tpm.S.member(t, tuples(r))
          implies
          Tpm.S.member(fv(t),Tpm.esm(fs(r),ndb))
          andalso Tpm.S.member(tv(t),Tpm.esm(ts(r),ndb))
  end =
```

Fig. 10. Requirements Specification for NDB_Relation

new instantiation of the module *NDBRELATION* is created for each value of *rk* in the domain of *rm*.

In Extended ML structures and not theories are created when functors are applied to actual parameters. The properties which are observable in the resulting structure, or class of structures, is given in the result interface of the functor. For example, all that is known about the class of structures resulting from the application of the functor Typed_Relation in Fig. 12 is that which can be deduced from the signature TYPED_RELATION (provided that every structure which can result on right hand side also matches this signature).

Signatures are not explicitly parameterised and sharing is only a form of dependent typing and not parameterisation. In the case of the functor Typed_Relation in Fig. 9 the result signature depends on the function esm in the parameter but since no axioms are given in the parameter signature constraining esm then there are also no constraints on esm visible in the result signature. Apart from the dependence of the result interface of Typed_Relation on the parameter this means that the interfaces to Typed_Relation are fixed

inv *mk-Ndb(esm,em,rm)*
  dom $em = \bigcup \mathrm{rng}\ esm\ \wedge$
  $\forall rk \in \mathrm{dom}\ rm \cdot$
  let *mk-Rkey(nm,fs,ts)* = *rk* in
  let *mk-Rinf(tp,r)* = *rm(rk)* in
  $\{fs, ts\} \subseteq \mathrm{dom}\ esm \wedge$
  $r \in NDBRELATION\,[fs, ts, esm, tp].Relation$

Fig. 11.

```
structure T : TYPED_RELATION =
        Typed_Relation(struct
                          structure S = Set
                          structure B = Basics
                             eqtype Attr = Attr
                               type NDB = NDB

                                val esm = Tpm.esm
                        end)
```

Fig. 12. Applying a functor

and consequently the development of `Typed_Relation` can be carried out in isolation.

5.2. N-ary Relations

To achieve the generalisations above we need to formulate the type checking and functional dependency constraints in the result signature of the functor `Typed_Relation`. If this is done then the body of the functor `Typed_Relation` must be formally developed to satisfy these two properties.

For the purposes of enforcing the type checking constraint attributes (elements of the type `Attr`) must be related to actual entity set names. Since each relation may associate attributes to entity set names in a different way this association depends upon relations themselves. This is done in appendix B.2 by including a function mapping attributes to entity set names in the two constructor functions for relations which consequently have the following types,

```
val empty  : Norm * (Esm.Attr -> Esm.B.Esetnm)
             -> Relation
and empty' : Esm.B.Rnm
             * Norm
             * (Esm.Attr -> Esm.B.Esetnm) -> Relation
```

Notice as well that the corresponding axioms are consequently higher order. The dependence of the association between attributes and entity set names upon relations is also the reason it is given in the signature `TYPED_RELATION` rather than being passed as a parameter as in [FiJ90].

The functional dependency constraint is now easy to formulate in `TYPED_RELATION`but the type checking constraint still requires an external component. We do not use a type checking function

$$tpc : Eid \times Esetnm \rightarrow bool$$

as in [FiJ90] since the formulation of the type checking constraint depends more precisely on the association between entity set names and the set of entity identifiers which they denote. This must come from outside the functor `Typed_Relation` and it is done by the function `esm` in our specifications. The sharing constraint ensures that the component `esm` of the result signature which is used in the type checking constraint is the same as that in the parameter.

Since we can make very few assumptions as to the form or use of `esm` it is left unconstrained.

5.3. Specialising N-ary Relations to Binary Relations

At this point we do not need to develop `Typed_Relation` further. All that we need to know about `Typed_Relation` for the purpose of specialising N-ary relations to binary relations is given in the requirements specification of Fig. 9.

We use a functor `NDB_Relation` to specialise N-ary relations to binary relations. Recall now from section 2 that it is possible to mix programs and specifications in Extended ML. The translation from N-ary to binary relations can be described by a Standard ML program just as well as by an axiomatic description and in appendix C.2 we give such a *program*. To ensure the type correctness of the functor body with respect to the signature `BINARY_RELATION` we need to include a function `unconv` which maps the representation of functional dependencies in terms of the type `Norm` back into functional dependencies represented in terms of the data type `Maptp`.

5.4. `NDB_Relation` in `Ndb`

In the body of the functor `Ndb` a structure `Binary` is created by applying `NDB_Relation` to an actual parameter structure. The result is an Extended ML structure [6]. To give an actual parameter for `NDB_Relation` we need a function `esm` to associate entity set names with sets of entity identifiers and this is done with the specification of a local function in the body of the functor (see appendix C.3)

```
fun esm(es : B.Esetnm, ndb : NDB) : Eid Set.Set = ?
```

which is not required in any further development of `Ndb`.

The function `esm` is local and therefore not required in any further developments of the functor `Ndb`. For the next development step to result in a *correct* refinement of the functor body in appendix C.3 the substructure `Binary` in the refinement needs only to be *observationally equivalent* [SaT87, SaT89] to the substructure `Binary` in appendix C.3.

We still need to show that the proof obligations for this step are met, that is,

$$\mathtt{BASICS} \cup \mathtt{Body}_{\mathtt{Ndb}} \models \mathtt{NDB}$$

which is straight forward but notice that the type checking constraint is hidden by the signature `NDB`.

If the type checking constraint were included in the signature `NDB`[7] then the proof obligation for this coding step would not be met. Since signatures are not parameterised there is no means for extending the result interface of `Typed_Relation` or `NDB_Relation` with a theory of `esm` local to the body of `Ndb`. Consequently the version of the type checking constraint visible in the substructure `Binary` in appendix C.3 will be weaker than that in the signature `NDB` and so the proof obligation for this step could not be discharged.

---

[6] Which specifies a class of Standard ML structures.

[7] See appendix D.2 where this is done.

```
functor Ndb(B : BASICS):
            sig
              include NDB'
              sharing B = Binary.B
            end = ?
```

Fig. 13.

```
functor Ndb(B : BASICS): sig
                           include NDB'
                           sharing B = Binary.B
                         end =
  Database(NDB_Relation'(Typed_Relation'(B)))
```

Fig. 14.

6. Modularisation by Functor Decomposition

Some criticisms about the program design specifications in section 5 are as follows:

1. `NDB_Relation` is intended to specialise N-ary relations to binary relations but there is nothing in the requirements specification which expresses this;
2. The type checking constraint is not a visible consequence in the signature `NDB` which may be useful knowledge for later users of this module;
3. The function `esm` which is used to formulate the type checking constraint is "under-specified" in the context of binary relations (this was necessary in order to meet all the proof obligations).

An alternative is to start with the requirements specification in Fig. 13 and use functor decomposition as in Fig. 14 to avoid some of these criticisms. The dependence of the type checking constraint on the association between entity identifiers and entity set names (given by `esm`) is best expressed in the `NDB` signature and this leads to a new signature `NDB'` given in appendix D.2. `NDB'` is identical to `NDB` except that it contains the type checking constraint previously given in the signature `TYPED_RELATION`. The requirements specifications for the three new functors in Fig. 14 are given in Fig. 15 while the signature `TYPED_RELATION'` is given in appendix D.1.

The resulting program structure is one in which the original task has been decomposed into three independent subtasks which interact only through the module interfaces. This means making the `Typed_Relation'` functor independent of its environment which we do by giving an abstract the type `Attr` in the signature `TYPED_RELATION'`. The idea is that now, unlike `TYPED_RELATION`, `TYPED_RELATION'` does not depend on any external functions or types another than those given in the signature `BASICS`. To avoid the second criticism we no longer wish to impose the type checking constraint on `Typed_Relation'` and consequently the parameter `esm` to `Typed_Relation` is also no longer needed.

The functor `NDB_Relation'` now takes any structure matching `TYPED_RELATION'` and results in a structure matching `BINARY_RELATION`. The requirements specification clearly states that `NDB_Relation` is to accept a module

```
functor Typed_Relation'( Basics : BASICS ) :
                         sig
                           include TYPED_RELATION'
                           sharing B = Basics
                         end = ?

functor NDB_Relation'( R : TYPED_RELATION' ) :
                       sig
                         include BINARY_RELATION
                         sharing R.B = B
                             and R.S = S
                       end = ?

functor DataBase( B : BINARY_RELATION) :
                  sig
                    include NDB'
                    sharing B = Binary
                  end = ?
```

Fig. 15.

for N-ary relations and construct a module matching `BINARY_RELATION` from it. The final functor `Database`, given in appendix D.3, then constructs the operations for updating the database from those of `BINARY_RELATION`.

The drawback to this structuring is that type checking constraint is no longer imposed on the interfaces of the module `Typed_Relation'` which was one of the goals of the generalisation from binary relations. What has been gained however, is a cleaner structuring of the program design specification in which the interfaces specify more clearly what each functor is to do in order to implement the original requirements specification.

7. Conclusion

In this paper we have considered the specification in Extended ML of the database described in [Wal90, FiJ90, WiS79] and a single program development step from each of two similar requirements specification. In section 5 a program design structure based on the structure of the specification in [FiJ90] was given by a coding development step while in section 6 an alternative program design given by a functor decomposition step was given.

The problem with our solution to the challenge problem in section 5 is that the type checking constraint, as given in the interface to `Typed_Relation`, is *independent* of context while to solve the challenge problem properly we would need to pass in axioms describing `esm` from whatever context `Typed_Relation` is used. This is not possible directly in Extended ML because there is no mechanism for explicitly parameterising a signature and sharing is only a mechanism for *dependent* typing in Extended ML.

Signatures in Extended ML are not parameterised for the reason that modules are to be developed, using the methodology outlined in section 2, without reference to other parts of the system. All that is required for the development of a module is specified in the interfaces to that module. If the signatures were parameterised we may still develop a program in isolation to meet that signature but each time an actual parameter was substituted for the formal parameter of the signature the complicated process of verifying the body of the module against the new signature would need to be done. This also effects the re-use of modules.

Extended ML meets the two criteria given in the introduction (as far as separation of program development concerns and module re-use are concerned) precisely because all the information one knows about a module is specified in the interfaces and this does not change in any contexts.

Finally, the program design specification in section 6 is given as the composition of three parameterised programs. Each functor builds on the operations of its argument to realise the original requirements specification of Fig. 14. A question that may be asked here is if the VDM structuring mechanisms can be used to give a compositional specification structure analogous to our functor decomposition described in section 6.

It is known that the class programs satisfying a parameterised specification is not generally the same as the class programs satisfying a parameterised program specification [SST90, SaT91]. The problems encountered when trying to formulate the type checking constraint are a consequence of this distinction and are chiefly due to the fact that signatures are not parameterised. The gain from this restriction is that once a parameterised program has been developed it can be simply treated as a black box where all that we need to know about it is captured in its interfaces. For example, we did not need to develop `Typed_Relation` further in order to construct the parts of the `Ndb` functor in which we were interested. The Extended ML approach is still very much in its infancy but with the completion of more examples using the Extended ML methodology we anticipate a better practical understanding of the particular strengths and weaknesses of this approach.

Acknowledgements

We would like to thank Arantxa Zatarain for reading over some of the earlier drafts of of the specifications and also to Don Sannella for his keen interest, many helpful suggestions and valuable comments on earlier versions of this paper. Also to Andrjez Tarlecki for his help with Extended ML and to Stephen Gilmore for his help with the VDM notation. Finally our thanks to the Science and Engineering Research Council for supporting this research and to the anonymous referees for their suggestions.

References

[CaW85] Cardelli, L. and Wegner, P.: On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4) 471–522, 1985.

[Dat86] Date, C.J.: *An Introduction to Database Systems.* Addison-Wesley, 1986.

[DaM82] Damas, L. and Milner, R.: Principle Type Schemes and Functional Programs. In *Principles of Programming Languages*, 207–212, 1982.

[EhM85] Ehrig, H. and Mahr. B.: *Fundamentals of Algebraic Specification 1.* EATCS: Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[EhM90] Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification 2 : Module Specifications and Constraints.* EATCS: Monographs on Theoretical Computer Science. Springer-Verlag, 1990.

[FiJ90] Fitzgerald, J. S. and Jones, C. B.: Modularising the Formal Description of a Database System. Technical Report UMCS-90-1-1, University of Manchester, January 1990.

[GoB80] Goguen, J. A. and Burstall, R. M.: The Semantics of Clear, A Specification Language. In *Abstract Software Specifications, LNCS 86*. Springer-Verlag, 1980.

[Har86] Harper, R.: Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, November 1986.

[HMT90] Harper, R., Milner, R. and Tofte, M.: *The Definition of Standard ML*. The MIT Press, 1990.

[Jon86] Jones, C.B.: *Systematic Software Development Using VDM*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1986.

[Mac86] MacQueen, D.: Modules for Standard ML. Technical Report ECS-LFCS-86-2, University of Edinburgh, March 1986.

[Mil77] Milner, R.: A Theory of Type Polymorphism in Programming. Internal Report CSR-9-77, University of Edinburgh, Department of Computer Science, September 1977.

[MiT90] Milner, R. and Tofte, M.: *Commentary on Standard ML*. MIT Press, 1990.

[Par72] Parnas, D.: On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 5(12) 1053–1058, 1972.

[San89] Sannella, D.: Formal Program Development in Extended ML for the Working Programmer. Technical Monograph ECS-LFCS-89-102, Laboratory for the Foundations of Computer Science, December 1989.

[SaB83] Sannella, D. and Burstall, R.: Structured Theories in LCF. In *Proceedings of the 8th Colloq. on Trees in Algebra and Programming, LNCS 159*, 377–391. Springer Verlag, 1983.

[SST90] Sannella, D., Sokolowski, S. and Tarlecki, A.: Toward Formal Development of Programs from Algebraic Specifications : Parameterisation Revisited. Technical Report Report 6/90, University of Bremen, 1990.

[SaT87] Sannella, D. T. and Tarlecki, A.: On Observational Equivalence and Algebraic Specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[SaT88] Sannella, D. and Tarlecki, A.: Building Specifications in an Arbitrary Institution. *Information and Control*, 76:165–210, February 1988.

[SaT89] Sannella, D. and Tarlecki, A.: Toward Formal Development of ML Programs: Foundations and Methodology:0 – Extended Abstract. In *Proceedings of the Colloquium on Current Issues in Programming Languages, LNCS 352*, 375–389. Springer Verlag, 1989.

[SaT91] Sannella, D. and Tarlecki, A.: A Kernel Specification Formalism with Higher Order Parameterisation . In *7th Workshop on Specification of Abstract Data Types*. Lecture Notes in Computer Science, to appear. Springer Verlag, 1991.

[Tof89] Tofte, M.: Four Lectures on ML. Technical Report ECS-LFCS-89-73, University of Edinburgh, March 1989.

[Wal90] Walshe, A.: NDB: The formal specification and rigorous design of a single user database. In: C. B. Jones and R. F. C. Shaw, (eds), *Case Studies in Systematic Software Development*, 12–45. Prentice Hall International, 1990.

[WiS79] Winterbottom, N. and Sharman, G. C. H.: NDB:Non-programmer database facility. Technical Report IBM TR.12.179, IBM Hursley Laboratory, England, September 1979.

Appendixes

A. The Database Signature

A.1. Preliminaries

```
signature SET =
  sig
     (*  Types  *)

     type ''a Set

     (*  Operators  *)

     val empty_set : ''a Set
     val insert    : ''a * ''a Set -> ''a Set
     val delete    : ''a * ''a Set -> ''a Set
     val member    : ''a * ''a Set -> bool
     val is_empty  : ''a Set -> bool
     val equals    : ''a Set * ''a Set -> bool

     (*  Axioms  *)

     axiom is_empty(empty_set) = true
       and is_empty(insert(e,S)) = false

     axiom member(e,empty_set) = false
       and e = e' implies member(e,insert(e',S)) = true
       and e <> e'
           implies member(e,insert(e',S)) = member(e,S)

     axiom e = e' implies member(e,delete(e',S)) = false
       and e <> e'
           implies member(e',delete(e,S)) = member(e',S)

     axiom equals(empty_set,empty_set)
       and (forall e  : ''a =>
            forall e' : ''a =>
               member(e,S) implies member(e,S')
               andalso
               member(e',S') implies member(e',S))
           implies equals(S,S')
  end;
```

A.2. Binary Relations

```
signature BINARY_RELATION =
  sig

    structure B : BASICS
```

```
structure S : SET


(*  Types  *)

eqtype Tuple and BinaryRel

datatype Maptp = ONE_ONE | MANY_ONE |
                 ONE_MANY | MANY_MANY

(*  Operations for Tuples  *)

val mk_tuple : B.Eid * B.Eid -> Tuple
val   fv     : Tuple -> B.Eid
val   tv     : Tuple -> B.Eid

(*  Axioms for Tuples  *)

axiom tv(mk_tuple(eid,eid')) = eid'
axiom fv(mk_tuple(eid,eid')) = eid
axiom mk_tuple(fv(t),tv(t)) = t

(*  Operations for Binary Relations  *)

val mk_rel  : Maptp * B.Esetnm * B.Esetnm
              -> BinaryRel
val mk_rel' : B.Rnm * Maptp * B.Esetnm * B.Esetnm
              -> BinaryRel
val add     : Tuple * BinaryRel -> BinaryRel
val map     : BinaryRel -> Maptp
val fs      : BinaryRel -> B.Esetnm
val ts      : BinaryRel -> B.Esetnm
val tuples  : BinaryRel -> Tuple S.Set

(*  Axioms for Binary Relations  *)

axiom map(mk_rel(mp,es,es')) = mp
  and map(mk_rel'(rnm,mp,es,es')) = mp
  and map(add(t,r)) = map(r)

axiom fs(mk_rel(mp,es,es')) = es
  and fs(mk_rel'(rnm,mp,es,es')) = es
  and fs(add(t,r)) = fs(r)

axiom ts(mk_rel(mp,es,es')) = es'
  and ts(mk_rel'(rnm,mp,es,es')) = es'
  and ts(add(t,r)) = ts(r)

local
   val tuples_of : Relation -> Tuple Esm.S.Set
   axiom tuples_of(empty) = Esm.S.empty_set
```

```
         and tuples_of(add(t,r)) = Esm.S.insert(t,r)
    in

      axiom tuples(mk_rel(mp,es,es')) = S.empty_set
        and tuples(mk_rel'(rnm,mp,es,es')) = S.empty_set

        (*  Constraints on adding Tuples to a Relation  *)

        and (S.member(t',tuples_of(add(t,r))) andalso
             S.member(t'',tuples_of(add(t,r)))
             implies
             case map(r) of
               ONE_ONE   => fv(t') = fv(t'')
                            iff tv(t') = tv(t'')
             | MANY_ONE  => fv(t') = fv(t'')
                            implies tv(t') = tv(t'')
             | ONE_MANY  => tv(t') = tv(t'')
                            implies fv(t') = fv(t'')
             | MANY_MANY => true)
             iff
             (tuples(add(t,r)) = S.insert(t,tuples(r))
              andalso map(add(t,r)) = map(r)
              andalso fs(add(t,r)) = fs(r)
              andalso ts(add(t,r)) = ts(r))
    end
  end;
```

A.3. The Data Base

```
signature NDB =
  sig

    structure Binary : BINARY_RELATION


    (*  The Data Base *)

    type NDB

    (*  Operations  *)

    val NewDB  : NDB
    val ADDES  : Binary.B.Esetnm * NDB -> NDB
    val ADDENT : Binary.B.Esetnm Binary.S.Set
                 * Binary.B.Value
                 * Binary.B.Eid
                 * NDB -> NDB
    val ADDTUP : Binary.Tuple
                 * Binary.B.Esetnm
                 * Binary.B.Esetnm
```

```
                * NDB -> NDB
  val ADDREL : Binary.BinaryRel * NDB -> NDB
  val DELES  : Binary.B.Esetnm * NDB -> NDB
  val DELENT : Binary.B.Eid * NDB -> NDB
  val DELTUP : Binary.Tuple
                * Binary.B.Esetnm
                * Binary.B.Esetnm
                * NDB -> NDB
  val DELREL : Binary.B.Esetnm
                * Binary.B.Esetnm
                * NDB -> NDB

  (*  Axioms  *)

 local

   (*  Auxiliary Functions  *)

   val isNewDB : NDB -> bool

   axiom isNewDB(NewDB) = true
     and isNewDB(ADDES(es,ndb)) = false
     and isNewDB(ADDENT(memb,value,eid,ndb)) = false
     and isNewDB(ADDREL(r,ndb)) = false
     and isNewDB(ADDTUP(t,es,es',ndb)) = false

   val Es_in : Binary.B.Esetnm * NDB -> bool

   axiom Es_in(es, NewDB) = false
     and Es_in(es, ADDES(es',ndb)) =
         (es = es' orelse Es_in(es,ndb))
     and Es_in(es, ADDENT(memb,value,eid,ndb)) =
         Es_in(es,ndb)
     and Es_in(es, ADDREL(r,ndb)) = Es_in(es,ndb)
     and Es_in(es, ADDTUP(t,es,es',ndb)) =
         Es_in(es,ndb)

   val Eid_in : Binary.B.Eid * NDB -> bool

   axiom Eid_in(eid, NewDB) = false
     and Eid_in(eid, ADDES(es,ndb)) = Eid_in(eid,ndb)
     and Eid_in(eid, ADDENT(memb,value,eid',ndb)) =
             (eid = eid') orelse Eid_in(eid,ndb)
     and Binary.S.member(t,Binary.Tuples(r))
         implies
         Eid_in(eid, ADDREL(r,ndb)) =
             eid = Binary.fv(t)
             orelse eid = Binary.tv(t)
             orelse Eid_in(eid,ndb)
     and Eid_in(eid, ADDTUP(t,es,es',ndb)) =
           eid = Binary.fv(t)
```

```
         orelse eid = Binary.tv(t)
         orelse Eid_in(eid, ndb)

val Rel_in : Binary.B.Esetnm
             * Binary.B.Esetnm
             * NDB -> bool

axiom Rel_in(es, es', NewDB) = false
  and Rel_in(es, es', ADDES(es'',ndb))
      = Rel_in(es, es',ndb)
  and Rel_in(es, es', ADDENT(memb,value,eid,ndb))
      = Rel_in(es,es',ndb)
  and Rel_in(es, es', ADDREL(r,ndb)) =
          (es = Binary.fs(r) andalso es' =
          Binary.ts(r))
          orelse Rel_in(es, es', ndb)
  and Rel_in(es, es', ADDTUP(t,es1,es2,ndb))
      = Rel_in(es, es',ndb)

(*  A local function associating entities
    with entity sets  *)

val esm : Binary.B.Esetnm * NDB
          -> Binary.B.Eid Binary.S.Set

axiom esm(es,NewDB) = Binary.S.empty_set
  and esm(es,ADDES(es',ndb)) = esm(es,ndb)
  and Binary.S.member(es,memb) implies
        esm(es,ADDENT(memb,value,eid,ndb)) =
        Binary.S.insert(eid,esm(es,ndb))
  and esm(es,ADDREL(r,ndb)) = esm(es,ndb)
  and esm(es,ADDTUP(t,es',es'',ndb)) = esm(es,ndb)

(*  A local function for associating
    Entity Identifiers with Values  *)

val em : Binary.B.Eid * NDB -> Binary.B.Value

axiom em(eid,ADDES(es,ndb)) = em(eid,ndb)
  and eid = eid' implies
      em(eid,ADDENT(memb,value,eid',ndb)) = value
  and eid <> eid' implies
      em(eid,ADDENT(memb,value,eid',ndb))
      = em(eid,ndb)
  and em(eid,ADDREL(r,ndb)) = em(eid,ndb)
  and em(eid,ADDTUP(t,es,es',ndb)) = em(eid,ndb)

(*  A local function for associating Entity
    set names with a relation  *)

val rm : Binary.B.Esetnm
```

```
            * Binary.B.Esetnm
            * NDB -> Binary.BinaryRel
   axiom rm(es, es', ADDES(es,ndb)) = rm(es, es',ndb)
     and rm(es, es', ADDENT(memb,value,eid,ndb))
         = rm(es, es',ndb)
     and (es = Binary.fs(r) andalso es'
         = Binary.ts(r))
         implies
         rm(es, es', ADDREL(r,ndb)) = r
     and (es = es1 andalso es' = es2)
         implies
         rm(es, es', ADDTUP(t,es1,es2,ndb)) =
             Binary.add(t,rm(es, es',ndb))
     and (es<>es1) andalso (es<>es2)
         implies
         rm(es, es', ADDTUP(t,es1,es2,ndb)) =
            rm(es, es',ndb)

in

   (* Operations for Constructing the Data Base *)

   (* ADDES *)

   axiom not(Es_in(es,ndb))
         implies
            Es_in(es,ADDES(es,ndb))
         andalso
            esm(es,ADDES(es,ndb)) = Binary.S.empty_set
          andalso
            forall eid : Binary.B.Eid =>
               (Eid_in(eid,ndb)
                implies
                em(eid,ADDES(es,ndb)) = em(eid,ndb))
          andalso
             forall es'  : Binary.B.Esetnm =>
             forall es'' : Binary.B.Esetnm =>
                    Es_in(es',ndb)
                    andalso Es_in(es'',ndb)
                    andalso Rel_in(es',es'',ndb)
                    implies
                    rm(es',es'',ADDES(es,ndb))
                    = rm(es',es'',ndb))

   (* ADDENT *)

   axiom Binary.S.member(es,memb)
         andalso Es_in(es,ndb)
         andalso not(Eid_in(eid,ndb))
         implies
            esm(es,ADDENT(memb,val,eid,ndb)) =
```

```
      Binary.S.insert(eid,esm(es,ndb))
    andalso em(eid,ADDENT(memb,val,eid,ndb))
            = val
    andalso
      forall es' : Binary.B.Esetnm =>
      forall es'': Binary.B.Esetnm =>
         Es_in(es',ndb) andalso Es_in(es'',ndb)
         implies
         rm(es',es'',ADDENT(memb,val,eid,ndb)
            = rm(es',es'',ndb)))

(* ADDREL *)

axiom Es_in(Binary.fs(r),ndb)
      andalso Es_in(Binary.ts(r),ndb)
      andalso Binary.S.is_empty(Binary.Tuples(r))
      andalso not(Rel_in(Binary.fs(r),
                         Binary.ts(r),ndb))
      implies
      Rel_in(Binary.fs(r),Binary.ts(r),
             ADDREL(r,ndb))
      andalso
      Binary.S.is_empty(
               Binary.Tuples(
               rm(Binary.fs(r),
               Binary.ts(r),ADDREL(r,ndb))))
       andalso
          forall eid : Binary.B.Eid =>
                 Eid_in(eid,ADDREL(r,ndb))
                 implies
                 em(eid,ADDREL(r,ndb))
                    = em(eid,ndb)
       andalso
          forall es : Binary.B.Esetnm =>
                 Es_in(es,ADDREL(r,ndb))
                 implies
                 esm(es,ADDREL(r,ndb))
                     = esm(es,ndb))
       andalso
          forall es'  : Binary.B.Esetnm =>
          forall es'' : Binary.B.Esetnm =>
                 es' <> Binary.fs(r)
                 andalso es'' <> Binary.ts(r)
                 implies
                 rm(es',es'',ADDREL(r,ndb))
                    = rm(es',es'',ndb)
       andalso
          forall es'  : Binary.B.Esetnm =>
          forall es'' : Binary.B.Esetnm =>
                 es' <> Binary.fs(r)
                 andalso es'' <> Binary.ts(r)
```

```
                     implies
                     rm(es',es'',ADDREL(r,ndb))
                        = rm(es',es'',ndb)

(* ADDTUP *)

axiom Rel_in(es,es',ndb)
      andalso Binary.Tuples(
              rm(es,es',ADDTUP(t,es,es',ndb))) =
              Binary.Tuples(
              Binary.add(t,rm(es,es',ndb)))
      implies
         rm(es,es',ADDTUP(t,es,es',ndb)) =
              Binary.add(t,rm(es,es',ndb))
      andalso
         forall es : Binary.B.Esetnm =>
            Es_in(es,ADDTUP(t,es',es'',ndb))
            implies
            esm(es,ADDTUP(t,es',es'',ndb))
                = esm(es,ndb)
      andalso
         forall eid : Binary.B.Eid =>
            Eid_in(eid,ndb) implies
            em(eid,ADDTUP(t,es',es'',ndb))
               = em(eid,ndb)

(* Operations for Deleting from the Data Base *)

(* DELES *)

axiom Binary.S.is_empty(esm(es,ndb))
      andalso
      (forall r : Binary.BinaryRel =>
          Rel_in(Binary.fs(r),Binary.ts(r),ndb)
          implies
          Binary.fs(r)<>es
          andalso Binary.ts(r)<>es)
      implies
         Es_in(es,DELES(es,ndb)) = false
         andalso
            forall eid : Binary.B.Eid =>
               em(eid,DELES(es,ndb)) = em(eid,ndb)
         andalso
         forall r : Binary.BinaryRel =>
            Rel_in(Binary.fs(r),Binary.ts(r),ndb)
            implies
            rm(Binary.fs(r),
               Binary.ts(r),DELES(es,ndb)) =
            rm(Binary.fs(r),Binary.ts(r),ndb))
```

```
(* DELENT *)

axiom (forall es  : Binary.B.Esetnm =>
       forall es' : Binary.B.Esetnm =>
          Rel_in(es,es',ndb) implies
          forall t : Binary.Tuple =>
             Binary.S.member(t,
                      Binary.tuples(rm(es,es',ndb)))
             implies
             Binary.fv(t)<>eid
             andalso Binary.tv(t)<>eid)
      implies
        (Eid_in(eid,DELENT(eid,ndb)) = false
         andalso
            forall eid' : Binary.B.Eid =>
               eid<>eid' implies
               em(eid',DELENT(eid,ndb))
                  = em(eid',ndb)
         andalso
            forall es'' : Binary.B.Esetnm =>
               esm(es'',DELENT(eid,ndb)) =
               Binary.B.delete(eid,esm(es'',ndb))
         andalso
            forall es  : Binary.B.Esetnm =>
            forall es' : Binary.B.Esetnm =>
               Rel_in(es,es',ndb)
               implies
               rm(es,es',DELENT(eid,ndb))
                  = rm(es,es',ndb)

(* DELREL *)

axiom (Rel_in(es,es',ndb)
       andalso Binary.S.is_empty(Binary.tuples(r)))
      implies
      (Rel_in(es,es',DELREL(es,es',ndb)) = false
       andalso
          forall es'' : Binary.B.Esetnm =>
             Es_in(es'',ndb)
             implies
             esm(es'',DELREL(es,es',ndb))
                 = esm(es,ndb)
       andalso
          forall Eid : Binary.B.Eid =>
             Eid_in(eid,ndb)
             implies
             em(eid,DELREL(es,es',ndb))
                = em(eid,ndb))
```

```
      (* DELTUP *)

      axiom Rel_in(es,es',ndb)
            implies
            (Binary.Tuples(
             rm(es,es',DELTUP(t,es,es',ndb))) =
             Binary.S.delete(
             t,Binary.tuples(rm(es,es',ndb)))
             andalso
                forall es'' : Binary.B.Esetnm =>
                   Es_in(es'',ndb)
                   implies
                   esm(es'',DELTUP(t,es,es',ndb))
                       = esm(es,ndb)
             andalso
                forall Eid : Binary.B.Eid =>
                   Eid_in(eid,ndb)
                   implies
                   em(eid,DELREL(es,es',ndb))
                      = em(eid,ndb))
    end
  end;
```

B. N-ary Relations with Type Checking

B.1. Preliminaries

```
signature ESM =
  sig
    structure S : SET
    structure B : BASICS

    eqtype Attr
      type NDB

       val esm : B.Esetnm * NDB -> B.Eid S.Set
  end;
```

B.2. N-ary Relations

```
signature TYPED_RELATION
  sig

    structure Esm : ESM

    (*  Tuples  *)

    eqtype Tuple
```

```
(*  Operations and Axioms for Tuples  *)

val create : (Esm.Attr * Esm.B.Eid) list -> Tuple
and value  : Tuple * Esm.Attr -> Esm.B.Eid

local
  val member : 'a * 'a list -> bool
  axiom member(a,nil) = false
    and member(a,a'::l) = (a = a') orelse member(a,l)

  val function : ('a * 'b) list -> bool
  axiom member((a,v),l) andalso member((a,v'),l)
        implies v = v'

in

  axiom member((a,eid),ae) andalso function(ae)
        implies value(create(ae),a) = eid
end

(*  Functional Dependencies  *)

type Norm

val mk_norm : (Esm.Attr Esm.S.Set * Esm.Attr)
              Esm.S.Set -> Norm
and attrs   : Norm -> (Esm.Attr Esm.S.Set * Esm.Attr)
              Esm.S.Set

axiom attrs(mk_norm(s)) = s
  and mk_norm(attrs(n)) = n

(*  Relations  *)

eqtype Relation

(*  Operations and Axioms for Relations  *)

val empty   : Norm * (Esm.Attr -> Esm.B.Esetnm)
              -> Relation
and empty'  : Esm.B.Rnm
              * Norm
              * (Esm.Attr -> Esm.B.Esetnm)
              -> Relation
and add     : Tuple * Relation -> Relation

(*  Projection functions  *)

and norm    : Relation -> Norm
and tpm     : Relation -> (Esm.Attr -> Esm.B.Esetnm)
and name    : Relation -> Esm.B.Rnm
```

```
(* Other operators on relations *)

and rem      : Tuple * Relation -> Relation
and tuples   : Relation -> Tuple Esm.S.Set

local
   val dom : Tuple -> Esm.Attr S.Set

   axiom dom(create(nil)) = Esm.S.empty_set
     and dom(create((a,eid)::t))
         = Esm.S.insert(a,dom(create(t)))

   val restrict : Tuple * Esm.Attr Esm.S.Set -> Tuple

   axiom forall a : Esm.Attr =>
         forall s : Esm.Attr Esm.S.Set =>
           Esm.S.member(a,s)
           andalso Esm.S.member(a,dom(t))
           implies
           value(a,restrict(t,s)) = value(a,t)

   val tuples_of : Relation -> Tuple Esm.S.Set

   axiom tuples_of(empty(nm,tm) = Esm.S.empty_set
     and tuples_of(empty'(rnm,nm,tm))
         = Esm.S.empty_set
     and tuples_of(add(t,r)) = Esm.S.insert(t,r)

in

  (* axioms for the projections *)

  axiom norm(empty(nm,tm)) = nm
    and norm(empty'(rnm,nm,tm)) = nm

  axiom tpm(empty'(rnm,nm,tm)) = tm
    and tpm(empty(nm,tm)) = tm

  axiom name(empty'(rnm,nm,tm)) = rnm

  (* rem *)

  axiom t = t' implies rem(t,add(t',r)) = r
    and t <> t'
        implies rem(t,add(t',r)) = add(t',rem(t,r))

  (* tuples - incorporating the functional
              dependency constraint  *)

  axiom tuples(empty(nm,tm)) = Esm.S.empty_set
```

```
        and tuples(empty'(rnm,nm,tm)) = Esm.S.empty_set

      axiom forall r     : Relation =>
            forall t     : Tuple =>
            forall t'    : Tuple =>
            forall (s,f) :
                   (Esm.Attr Esm.S.Set * Esm.Attr) =>
            (Esm.S.member((s,f),norm(r))
             andalso Esm.S.member(t,tuples_of(add(t,r)))
             andalso Esm.S.member(t',tuples_of(add(t,r)))
             andalso restrict(t,s) = restrict(t's)
             implies value(t,f) = value(t',f)
            )
               implies
                 tuples(add(t,r))
                 = Esm.S.insert(t,tuples(r))

      (*  ... and the type checking constraint  *)

      axiom forall r : Relation =>
            forall t : Tuple    =>
            forall a : Esm.Attr =>
            Esm.S.member(t,tuples(r))
            andalso Esm.S.member(a,dom(t))
            implies
            Esm.S.member(value(a,t),esm.esm(tpm(a),ndb))
    end
  end;
```

C. Binary Relations in the Body of in `Ndb`

C.1. The Signature `TPM`

```
signature TPM =
  sig
    structure Set    : SET
    structure Basics : BASICS
    type NDB
     val esm : Basics.Esetnm * NDB -> Basics.Eid Set.Set
  end;
```

C.2. The Functor `NDB_Relation`

```
functor NDB_Relation(Tpm : TPM) :
  sig

    include BINARY_RELATION
```

```
    (*  Type Checking Constraint  *)

    axiom Tpm.S.member(t, tuples(r))
          implies
          Tpm.S.member(fv(t),Tpm.esm(fs(r),ndb))
          andalso Tpm.S.member(tv(t),Tpm.esm(ts(r),ndb))
  end =
  struct

      (* Relation Kinds *)

      datatype Maptp = ONE_ONE | MANY_ONE |
                       ONE_MANY | MANY_MANY

      (*  Attribute Names  *)

      datatype Attr = Fs | Ts

      (* Conversion to Functional Dependencies *)

      fun conv(ty) =
        let
          val insert = Tpm.Set.insert
          val empty_set = Tpm.Set.empty_set
        in
          case ty of
            MANY_MANY =>
            insert((insert(Ts,empty_set),Fs),
                insert((insert(Fs,empty_set),Ts),
                               empty_set))
          | MANY_ONE =>
            insert((insert(Fs,empty_set),Ts), empty_set)
          | ONE_MANY =>
            insert((insert(Ts,empty_set),Fs), empty_set)
          | ONE_ONE  => empty_set
        end

      structure T : TYPED_RELATION =
         Typed_Relation(struct
                           structure S = Set
                           structure B = Basics
                              eqtype Attr = Attr
                                type NDB = NDB

                                 val esm = Tpm.esm
                         end)

      (*  Converting norms back into map types  *)

      fun unconv( n : T.Norm) : Maptp = ?
```

```
local
   val insert = Tpm.Set.insert
   val empty_set = Tpm.Set.empty_set
   val Fs_set     = insert(Fs,empty_set)
   val Ts_set     = insert(Ts,empty_set)
in
  axiom
    unconv(T.mk_norm(insert((Ts_set,Fs),
                    insert((Fs_set,Ts), empty_set)))
         = MANY_MANY
    and unconv(T.mk_norm(insert((Fs_set,Ts),
                         empty_set))) = MANY_ONE
    and unconv(T.mk_norm(insert((Ts_set,Fs),
                         empty_set))) = ONE_MANY
    and unconv(T.mk_norm(empty_set)) = ONE_ONE
end

(*  Concrete Programs for Tuples  *)

eqtype Tuple = T.Tuple

fun mk_tuple(eid,eid')
    = T.create([(Fs,eid),(Ts,eid')])
fun fv(t) = T.value(t,Fs)
fun tv(t) = T.value(t,Ts)

(*  Concrete Programs for Binary Relations  *)

eqtype BinaryRel = T.Relation

fun mk_rel(mtp,es,es') =
    let
      val tm = fn Fs => es | Ts => es'
    in
      T.empty(conv(mtp),tm)
    end

fun mk_rel'(rnm,mtp,es,es') =
    let
      val tm = fn Fs => es | Ts => es'
    in
      T.empty'(rnm,conv(mtp),tm)
    end

fun add(t,r) = T.add(t,r)

fun tuples(r) = T.tuples(r)

fun fs(r) = T.tpm(r)(Fs)

fun ts(r) = T.tpm(r)(Ts)
```

```
    fun map(r) = unconv(T.norm(r))
  end;
```

C.3. The NDB Functor

```
functor Ndb( B : BASICS) : sig
                             include NDB
                             sharing B = Binary.B
                           end =
  struct

    structure Set : SET = ?

    (*  The Data Base *)

    type NDB = ?


    local

      (*  A local function associating
          entities with entity sets  *)

      fun esm(es : Esetnm, ndb : NDB) : Eid Set = ?

    in

      (*  A structure for binary relations  *)

      structure Binary :
        sig
          include BINARY_RELATION

          axiom Set.member(t, tuples(r))
                implies
                (Set.member(fv(t),esm(fs(r),ndb))
                 andalso Set.member(tv(t),
                         esm(ts(r),ndb)))
        end =
        Ndb_Relation( struct
                        structure Set = Set
                        structure Basics = B
                        type NDB = NDB
                        val esm = esm
                      end )
    end

    local
```

```
    (*  Auxiliary Functions  *)

    fun isNewDB(ndb : NDB) : bool = ?

    axiom . . .

    fun Es_in(es : B.Esetnm, ndb : NDB) : bool = ?

    axiom . . .

    fun Eid_in(eid : B.Eid, ndb : NDB) : bool = ?

    axiom . . .

    fun Rel_in(rel : Binary.BinaryRel,
               ndb : NDB) : bool = ?

    axiom . . .

    (*  A local function for associating
        Entity Identifiers with Values  *)

    val em : Binary.B.Eid * NDB -> Binary.B.Value

    (*  A local function for extracting
        the set of tuples in a relation  *)

    fun rm(rel : Binary.BinaryRel, ndb : NDB) : NDB = ?

  in

    (* Operations for Constructing the Data Base *)

    val NewDB : NDB = ?

    fun ADDES( es : B.Esetnm, ndb : NDB) : NDB = ?

    fun ADDENT( memb : B.Esetnm Set.Set,
               value : B.Value,
                 eid : B.Eid,
                 ndb : NDB) : NDB = ?

    fun ADDTUP(tuple : Binary.Tuple,
                 rel : Binary.BinaryRel,
                 ndb : NDB) : NDB = ?

    fun ADDREL( mp : Binary.Maptp,
               ndb : NDB) : NDB = ?

    fun DELES( es : B.Esetnm,
              ndb : NDB) : NDB = ?
```

```
      fun DELENT(eid : B.Eid,
                 ndb : NDB) : NDB = ?

      fun DELTUP(eid  : B.Eid,
                 eid' : B.Eid,
                 rel  : Binary.BinaryRel,
                 ndb  : NDB) : NDB = ?

      fun DELREL(rel : Binary.BinaryRel,
                 ndb : NDB) : NDB = ?

      (* Axioms for the update operations *)

      . . .

    end
  end;
```

D. The DataBase Functor

D.1. The Signature TYPED_RELATION'

```
signature TYPED_RELATION' =
  sig

    structure B : BASICS
    structure S : SET

    (*  A type for attributes  *)

    type Attr

     val first : Attr
     and next  : Attr -> Attr

     axiom not(first = next(first))

    (*  Tuples  *)

    eqtype Tuple

    (*  Operations and Axioms for Tuples  *)

    val create : (Attr * B.Eid) list -> Tuple
    and value  : Tuple * Attr -> B.Eid

    local
      val member : 'a * 'a list -> bool
      axiom member(a,nil) = false
```

```
      and member(a,a'::l) = a = a' orelse member(a,l)

    val function : ('a * 'b) list -> bool
    axiom (member((a,v),l) and also member((a,v'),l))
          implies v = v'
  in
    axiom member((a,eid),ae) andalso function(ae)
          implies value(create(ae),a) = eid
  end

  (*  Functional Dependencies  *)

  type Norm

  val mk_norm : (Attr S.Set * Attr) S.Set  -> Norm
  and attrs   : Norm -> (Attr S.Set * Attr) S.Set

  axiom attrs(mk_norm(s)) = s
    and mk_norm(attrs(n)) = n

  (*  Relations  *)

  eqtype Relation

  (*  Operations and Axioms for Relations  *)

  val empty   : Norm * (Attr -> B.Esetnm) -> Relation
  and empty'  : B.Rnm * Norm * (Attr -> B.Esetnm)
                -> Relation
  and add     : Tuple * Relation -> Relation
  and norm    : Relation -> Norm
  and tpm     : Relation -> (Attr -> B.Esetnm)
  and name    : Relation -> B.Rnm
  and rem     : Tuple * Relation -> Relation
  and tuples  : Relation -> Tuple S.Set

  local
     val dom : Tuple -> Attr S.Set

     axiom dom(create(nil)) = S.empty_set
       and dom(create((a,eid)::t))
           = S.insert(a,dom(create(t)))

     val restrict : Tuple * Attr S.Set -> Tuple

     axiom forall a : Attr =>
           forall s : Attr S.Set =>
             S.member(a,s) andalso S.member(a,dom(t))
             implies
             value(a,restrict(t,s)) = value(a,t)
```

```
      val tuples_of : Relation -> Tuple S.Set

      axiom tuples_of(empty(nm,tm) = S.empty_set
        and tuples_of(empty'(rnm,nm,tm)) = S.empty_set
        and tuples_of(add(t,r)) = S.insert(t,r)

   in

     (* axioms for the projections *)

     axiom norm(empty(nm,tm)) = nm
       and norm(empty'(rnm,nm,tm)) = nm

     axiom tpm(empty'(rnm,nm,tm)) = tm
       and tpm(empty(nm,tm)) = tm

     axiom name(empty'(rnm,nm,tm)) = rnm

     (* rem *)

     axiom t = t' implies rem(t,add(t',r)) = r
       and t <> t'
           implies rem(t,add(t',r)) = add(t',rem(t,r))

     (* tuples *)

     axiom tuples(empty(nm,tm)) = S.empty_set
       and tuples(empty'(rnm,nm,tm)) = S.empty_set

     axiom forall r     : Relation =>
           forall t     : Tuple =>
           forall t'    : Tuple =>
           forall (s,f) : (Attr S.Set * Attr) =>
              (S.member((s,f),norm(r))
               andalso S.member(t,tuples_of(add(t,r)))
               andalso S.member(t',tuples_of(add(t,r)))
               andalso restrict(t,s) = restrict(t's)
               implies value(t,f) = value(t',f)
              )
              implies
                tuples(add(t,r)) = S.insert(t,tuples(r))

   end
 end;
```

D.2. The Signature NDB'

```
signature NDB' =
  sig
```

```
structure Binary : BINARY_RELATION

(*  The Data Base *)

type NDB

(*  Operations  *)

val NewDB   : NDB
val ADDES   : Binary.B.Esetnm * NDB -> NDB
val ADDENT  : Binary.B.Esetnm Binary.S.Set
              * Binary.B.Value
              * Binary.B.Eid
              * NDB -> NDB
val ADDTUP  : Binary.Tuple
              * Binary.B.Esetnm
              * Binary.B.Esetnm
              * NDB -> NDB
val ADDREL  : Binary.BinaryRel * NDB -> NDB
val DELES   : Binary.B.Esetnm * NDB -> NDB
val DELENT  : Binary.B.Eid * NDB -> NDB
val DELTUP  : Binary.Tuple
              * Binary.B.Esetnm
              * Binary.B.Esetnm
              * NDB -> NDB
val DELREL  : Binary.B.Esetnm
              * Binary.B.Esetnm
              * NDB -> NDB

(*  Axioms  *)

local

  (*  Auxiliary Functions  *)

  val isNewDB : NDB -> bool

      . . .

  val Es_in : Binary.B.Esetnm * NDB -> bool

      . . .

  val Eid_in : Binary.B.Eid * NDB -> bool

      . . .

  val Rel_in : Binary.B.Esetnm
               * Binary.B.Esetnm
               * NDB -> bool
```

```
        . . .

    (*  A local function associating
        entities with entity sets  *)

    val esm : Binary.B.Esetnm * NDB
                -> Binary.B.Eid Binary.S.Set

        . . .

    (*  A local function for associating
        Entity Identifiers with Values  *)

    val em : Binary.B.Eid * NDB -> Binary.B.Value


    (*  A local function for associating
        Entity set names with a relation  *)

    val rm : Binary.B.Esetnm
               * Binary.B.Esetnm
               * NDB -> Binary.BinaryRel

  in

    (* Operations for Constructing the Data Base *)

        . . .

    (*  Type Checking Constraint  *)

    axiom Rel_in(r,ndb)
          andalso Binary.S.member(t, Binary.tuples(r))
          implies
            Binary.S.member(Binary.fv(t),
                            esm(Binary.fs(r),ndb))
            andalso
            Binary.S.member(Binary.tv(t),
                            esm(Binary.ts(r),ndb))
  end
end;
```

D.3. The Functor DataBase

```
functor DataBase( B : BINARY_RELATION) :
                  sig
                    include NDB
                    sharing B = Binary
                  end =
```

```
struct

  structure Set : SET = ?
  structure Binary = B

  open B Set

  (*  The Data Base *)

  type NDB = ?

  (*  Operations  *)

  val NewDB : NDB = ?

  fun ADDES(es : Esetnm, ndb : NDB) : NDB = ?

  fun ADDENT( memb : Esetnm set,
             value : Value,
               eid : Eid,
               ndb : NDB) : NDB = ?

  fun ADDTUP(tuple : B.Tuple,
               rel : B.BinaryRel,
               ndb : NDB) : NDB = ?

  fun ADDREL(mp : Maptp, ndb : NDB) : NDB = ?

  fun DELES(es : Esetnm, ndb : NDB) : NDB = ?

  fun DELENT(eid : Eid, ndb : NDB) : NDB = ?

  fun DELTUP(eid  : Eid,
             eid' : Eid,
             rel  : B.BinaryRel,
             ndb  : NDB) : NDB = ?

  fun DELREL(rel: B.BinaryRel, ndb : NDB) : NDB = ?

  (*  Axioms  *)

  local

    (*  Auxiliary Functions  *)

    val isNewDB : NDB = ?

    axiom . . .

    fun Es_in(es : Binary.B.Esetnm,
              ndb : NDB) : bool = ?
```

```
    axiom . . .

    val Eid_in(eid : Binary.B.Eid,
               ndb : NDB) : bool = ?

    axiom . . .

    val Rel_in(r : 'b, ndb : NDB) : bool = ?

    axiom . . .

    (*  A local function for extracting
        the set of tuples in a relation  *)

    val rm(r : 'b, nbd : NDB) : NDB = ?

    axiom . . .

    (*  A local function associating
        entities with entity sets  *)

    val esm(es : Esetnm, ndb : NDB) : Eid Set = ?

    axiom . . .

  in

    (* Operations for Constructing the Data Base *)

    . . .

  end
end;
```

Received December 1990
Accepted in revised form September 1991 by B. T. Denvir