Modularizing the Formal Description of a Database System

J.S. Fitzgerald and C.B. Jones*

January 23, 1990

Abstract

Specification languages have been proposed, and are being developed, which offer ways of splitting specifications into separate components or modules. It is important that such languages are able to cope with modularizations which are required by realistic specification tasks. This paper offers a challenge problem in modularization based on a description[1] is of an existing database system. The chosen modularization is motivated by the need to separate coherent units about which useful properties can be deduced.

Introduction

Module facilities in programming languages (such as Modula, Ada[2], Clu) have long been regarded as important tools for coping with the size and complexity of today's programming tasks. As formal methods of software specification and development are applied to larger problems, so the need arises here too for facilities supporting a modular approach. Hence, for example, the work to develop and incorporate such facilities in BSI Standard VDM [And88], COLD-K [Jon88] and the RAISE Specification Language (RSL) [Haf89]. It is important to identify the goals of such techniques in the context of a formal development method.

One goal is the effective *separation of concerns* within a specification. This allows formal reasoning about a part of a system unencumbered by 'baggage' which describes unrelated parts of the same system. Informally, a reader can understand a specification as 'the sum of its parts' rather than as a monolithic whole. In a model-oriented specification, effective separation of concerns motivates the provision of an interface around each module, stating what the module expects of the rest of the specification and what properties the module guarantees to exhibit. The interface delineates the model changes which can be made within the module without affecting the rest of the specification. If changes in one module do affect the rest of the specification, the structuring mechanism can help control those changes.

In the context of a formal *development*, separation of concerns might allow [Mor90] separate development of modules (via data reification, for example) provided an appropriate module calculus at the code level ensures the 'correct' recombination of code.

Another goal is *module generality*. This is as desirable for specification modules as for code. Parameterization allows some modules describing relatively general concepts to be used in a variety of contexts.

These goals lead the authors to suggest criteria which should govern the choice of structure in a modular specification:

*This paper has been accepted for the VDM'90 Symposium and is to be published in the Proceedings (to appear in the Springer-Verlag LNCS Series).

[1]As a concession to common usage, the authors employ the terms 'specification' and 'description' interchangeably in this paper.

[2]Trademark: US Dept. of Defense, Ada Joint Program Office

- the complexity of component modules as measured by the number of hypotheses in the statements of theorems about each module.
- the suitability of generalized modules for re-use in other specifications;
- intuitive clarity of the structuring – the module structure should aid rather than impede comprehension of the whole specification;

Note that none of these criteria concern the suitability of a specification's module structure for formal development; see Section 4.1 for a further consideration.

This paper presents a case study in the modular structuring of a previously 'flat' specification. The study aims to cast light on the problems of choosing a structure for a modular specification, as well as illustrating some desirable features in the yet-to-be-fully-defined module formalism used and pointing out consequent problems for its semantics. These objectives are not specific to any one specification or development method, so the exercise described here may be regarded as a 'challenge problem'.

The description below is of an existing database system. The advantages of basing the study on an existing piece of software extend beyond honesty and masochism since some of the arcane choices made in the actual development of the software pose interesting problems for the modularization, notably in handling the curious distribution of typing information in the database structure described.

The specification is presented in Meta-IV, the specification language of VDM, and was prompted by a study in RSL. 'Classical VDM' [Jon80, BJ82] has no structuring facilities, but introductory material on structuring techniques in VDM can be found in [Jon90] and [Sha90]. Proposals regarding the semantics of VDM modules can be found in [Bea88]. The module syntax used below is related to that of [BSI].

1 A Flat Description

This section introduces the database system whose description is being considered by giving a 'flat' or single level description; the task of modularization is begun in Section 2. 'NDB' stood for 'Norman's Database' and was described in [WS79]; it eventually became an IBM product, more prosaically named the 'Non-programmer Database' [IBM]. NDB is a nicely engineered binary relational (cf. [Dat86] for general database matters) database system which was designed with the non-expert user in mind. Some of the engineering decisions which both make it easy to use and limit it to medium-sized databases are discussed in Section 4.1. A particularly pleasing aspect of the design is the way in which so-called 'meta-data' (i.e. information about the relations etc.) is handled. A fuller description than is justified here of the development of the formal description of NDB is given in [Wal90]. Here, a brief introduction must suffice to identify the main points in the specification. 'Entities' are the basic objects about which information is stored. Each entity is identified by a unique 'Entity Identifier' (*Eid*). Most entities have values (here, it is sufficient to have this set unanalysed as *Value*) but entities without values can exist. The relationship between entity identifiers and their values can be stored in an 'Entity map' which is $Eid \xrightarrow{m} [Value]$.

In order to introduce typing constraints, entities are grouped into 'Entity Sets' whose names are in *Esetnm*. In this version of the formal description,[3] entity sets do not have to be disjoint in the sense that the same entity identifier can belong to more than one set (e.g. an individual could be both a customer and an employee): the grouping is therefore stored as an entity set map which is $Esetnm \xrightarrow{m} Eid\text{-set}$.

Many relations can be stored in a database and, as one might expect, they can be named (*Rnm*). But a name can be used for more than one relation (e.g. the name 'Owns' might relate departments to machinery and people to cars) and a 'Relation Key' (*Rkey*) is therefore made to contain the entity set names of both the 'from set' (*fs*) and 'to set' (*ts*) of a relation, thus:

[3]See [Wal90] for a discussion of alternative architectures and their models.

$$\begin{aligned} Rkey ::\ & nm : [Rnm] \\ & fs : Esetnm \\ & ts : Esetnm \end{aligned}$$

Notice that the name itself (*nm*) is optional: un-named relations arise as inverses of user-given relations.

Having captured most of the typing information in the relation key, the 'Relation Information' (*Rinf*) is little more than a set of tuples. What is extra is information about the 'Mapping type' (*Maptp*) which records whether the relation is restricted to be one-to-one (1:1), one-to-many (1:M), etc. A *Tuple* contains two entity identifiers – one as a 'from value' (*fv*) and one as a 'to value' (*tv*), thus:

$$Maptp = \{1{:}1, 1{:}\mathrm{M}, \mathrm{M}{:}1, \mathrm{M}{:}\mathrm{M}\}$$

$$\begin{aligned} Tuple ::\ & fv : Eid \\ & tv : Eid \end{aligned}$$

$$\begin{aligned} Rinf ::\ & tp : Maptp \\ & r : Tuple\text{-set} \end{aligned}$$

The main 'Relation Map' is then $Rkey \xrightarrow{m} Rinf$

VDM specifications are normally built around a state. Here, *Ndb* collects the three components described above:

$$\begin{aligned} Ndb ::\ & esm : Esetnm \xrightarrow{m} Eid\text{-set} \\ & em : Eid \xrightarrow{m} [Value] \\ & rm : Rkey \xrightarrow{m} Rinf \end{aligned}$$

This set of objects needs to be restricted by a data type invariant whose clauses require that for an *Ndb* to be well-formed it must be such that:

- each entity in the domain of *em* must be present in one (or more) entity set;
- the entity set names in *fs* and *ts* of each relation key must be in the domain of the entity set map (*esm*);
- the sets of tuples in each relation information must respect the *Maptp*;
- the *fv* and *tv* of each tuple must respect the typing information (as given by *fs*/*ts*) in the relation key.

This invariant is presented formally in Appendix A where the whole specification is given. The initial state contains empty mappings for all three components. Operations are defined in Appendix A which add (*ADDES*) or delete (*DELES*) entity sets, add (*ADDENT*) or delete (*DELENT*) entities, add (*ADDREL*) or delete (*DELREL*) empty relations, add (*ADDTUP*) or delete (*DELTUP*) tuples. Notice that, although described as a flat specification, it is parameterized on the types *Eid, Value, Esetnm, Relnm* about which no information is required other than that an equality operator is available.

2 A First Modularization

It is explained in the introduction that the main aim of the modularization considered in this paper is to provide general concepts. The obvious concept which is a candidate for re-use in the foregoing specification is the relation. It can be generalized from the specifics of NDB in a number of ways. In this section, a simple generalization forms the basis of a modularization.

2.1 *N*-ary *Relation*

The most obvious generalization is from NDB's restriction to binary relations to allow '*n*-ary' relations. The idea is to create a module which specifies relations of arbitrary arity which can be used in a specification of NDB and, perhaps, a range of other database system specifications. This gives rise to the module *RELATION* shown in Appendix B.1.

Since a tuple may now contain an arbitrary number of attribute values, the type *Tuple* should be specified $Attr \xrightarrow{m} E$ where E is an entity type.[4] *Attr* is the type of permissible attribute names. As far as *RELATION* is concerned, *Attr* can be parametric to the module. A relation is still specified as a set of tuples.[5] The invariant inv-*Relation* is necessary to ensure that a relation consists only of tuples with the same attributes.

A number of functions are defined to complete the encapsulation. The formalism used in this paper assumes the hiding of data model representations – access to exported data types (and the module state, if any) is permitted only via exported functions and operations. This approach has the advantage that semantically consistent changes to a representation in one module do not force changes in importing modules. However, it often necessitates the trivial renaming of standard data type operators so that they can be explicitly exported (e.g. *value,empty* in Appendix B.1), thereby inflating the module interface.

2.2 NDB specified using *RELATION*

The specification of NDB which results from the use of *RELATION* is shown in the module *FIRST-NDB* in Appendix B.2. The parameters of *RELATION* are instantiated by particular types and

$$RELATION[Eid, Fsel]$$

is imported. The main body of *FIRST-NDB* accesses relations solely through the interface to *RELATION* using the exported functions and types. Otherwise, the body of *FIRST-NDB* is much the same as that of the flat specification.

2.3 Shortcomings of the First Modularization

What has been gained by this structuring? Certainly the aim of module re-use has been achieved, but at a price: it is difficult for the reader to comprehend the structured specification as the 'sum of its parts' because there remain properties of relations which could be encapsulated in the relevant module, but are left to importing modules to describe.

Consider the function *attr-match* defined in the interface to *RELATION*: it partitions the set of all possible *Relations* into those of equivalent 'shape' (in the sense that they have the same sets of *Attrs*). But this fact is *not* stated in the type *Relation*; the module *RELATION* exports the function *attr-match* which has to be used by *inv-Ndb*. More could be gained (for the price of the modularization) if the *RELATION* module were to include this 'shape' information. Given the current specification language this can not be done. The extensions to the language, which are discussed in the next section, not only facilitate this encapsulation but also make further generalizations possible.

In support of this, consider the statement of theorems about relations exported by the *RELATION* module. Theorems about typing must have a typing mechanism fixed in the hypotheses. Thus (using an obvious with notation) one could assert the invariant-preservation property of *add*:

[4]Note that the type E is parametric to this module. NDB has both a *Value* and *Eid* set. This division comes from one, rather concrete, view of how entities are to be handled. What needs to be fixed at this stage is no more than a name (E) for the set of entities and the fact that equality is a valid operator for pairs of entities.

[5]The separation of *Relation* and *Tuple* into two modules results in much duplication and no useful generalization.

with $RELATION[E, Attr]$;
$t \in Tuple; r \in Relation;$

$$\boxed{\mathrm{Th1}}\;\frac{pre\text{-}add(r,t)}{add(r,t) \in Relation}$$

But in order to make an assertion about the 'shape' of the relations, one must write:

with $RELATION[E, Attr]$;
$t \in Tuple; r \in Relation;$
$pre\text{-}add(r,t);$
$as \in Attr\text{-set};$

$$\boxed{\mathrm{Th2}}\;\frac{attr\text{-}match(r,as)}{attr\text{-}match(add(r,t),as)}$$

3 Modularization with Typing and Normalization

Once the notion of shape (in the sense of Section 2.3) has been introduced into the module describing relations, the way is clear to fixing a number of other properties there, including the typing of values in tuples/relations and the expression of functional dependencies between attributes. It could be argued that this goes against our principle of generalization, but it is clear that choosing a modularization involves balancing the generality of derived modules with their applicability.

This section discusses the process described above and considers the required extensions to the specification language.

3.1 *TYPED-RELATION*

Considerations leading to the fixing of further properties in the relation module are:

- The type checking, which in NDB is governed by the *esm* mapping ($Esetnm \xrightarrow{m} Eid\text{-set}$), can be generalized to a type checking function *tpc*: assuming some elementary set of entity types (*Etp*), *tpc* must be of type $E \times Etp \to \mathbb{B}$. The subsequent specialization of this in *NDBRELATION* results in an interesting parameter (see Section 3.2.1).
- The typing information in the flat specification is carried in the *Rkey* of each relation (in the *fs* and *ts* fields). In a specification based on a generalized module for relations, it is reasonable to move this information into the relation module. Note that, since the relation module deals with n-ary relations, the typing information must be expressed in a mapping (*tpm*) which is $Attr \xrightarrow{m} Etp$. Notice that the entity type (*Etp*) links back to the generalized type checking.
- Clearly, there must be attendant generalization of *Maptp*. A wide class of 'normalizations'(also called 'functional dependencies') of sets of tuples can be described by specifying a set of constraints each of which requires that the value under one *Attr* is determined by the values under a specified set of *Attrs*. Thus, a many-to-one relation might have a (single) constraint $\{\{\mathrm{FS}\}, \mathrm{TS}\}$ (see *conv* in Appendix C.2).

These considerations – in conjunction with the earlier work on *RELATION* – give rise to the module *TYPED-RELATION* (see Appendix C.1) which has four sorts of parameter. The 'trivial' type parameters are *E*, *Etp*, *Attr*[6]. The shape of a particular relation is governed by *tpm*; type information is given by *tpc*; and the normalization is governed by the parameter *norm*.

The various functions defined in terms of these models (see Appendix C.1) should be obvious. In terms of these functions, a number of lemmas can be formulated (and proved via their models)[7]. For

[6] *Etp* has been added so that it is available for the signatures of *tpm* and *tpc*.
[7] The equivalent of Th1 in the context of *TYPED-RELATION* is clear. Th3 is like Th2, but 'tighter' in the sense that the normalization constraints are fixed in the latter.

example, the uniform ability to add new *Tuples* to *Relations* of the same type if there are no normalization constraints can be written:

with $TYPED\text{-}RELATION[E, Etp, Attr, tpm, tpc, \{\ \}]$;

$$\boxed{\text{Th3}}\quad \frac{\begin{array}{c} t \in Tuple;\ r \in Relation; \\ pre\text{-}add(r,t) \end{array}}{add(r,t) \in Relation}$$

Whereas the ability, with any normalization constraint, to insert the first *Tuple* into a *Relation* can be written:

with $TYPED\text{-}RELATION[E, Etp, Attr, tpm, tpc, norm]$;

$$\boxed{\text{Th4}}\quad \frac{t \in Tuple}{add(empty(), t) \in Relation}$$

This theory is general in the sense that its results are true for any instantiation of *TYPED-RELATION*; specific results about *NDBRELATION* are given in Section 4.2.

3.2 Specifying NDB using *TYPED-RELATION*

The final structured version of the NDB specification is in Appendices C.2 and C.3. As in the flat specification, the main module *NDB* is parameterized over primitive types only. It uses *TYPED-RELATION* as in Appendix C.1 but, rather than import *TYPED-RELATION* directly, access to its exported types and operators comes via a module *NDBRELATION* which partially particularizes *TYPED-RELATION*. *NDBRELATION* is peculiar to NDB and is therefore nested in the *NDB* module. Nesting provides a fixed context in which *NDBRELATION* can be interpreted and thus reduces the size of its parameter list (see Section 3.2.2).

Section 3.2.1 considers *NDBRELATION* alone. Section 3.2.2 places it in context in *NDB*.

3.2.1 *NDBRELATION*

NDBRELATION is shown in Appendix C.2 isolated and out of context. It is to be interpreted as nested in the *NDB* module as shown in Appendix C.3. Relevant parts of that context are shown in the let clauses, but for semantic purposes may be regarded as parameters.

NDBRELATION can be viewed as providing a translation mechanism, expressing *NDB*-specific concepts in the the more general framework of *TYPED-RELATION*. It 'looks up' the relevant instance of *TYPED-RELATION* for a given set of typing and normalization criteria and re-exports its public types and operations. The parameters identifying the relevant *TYPED-RELATION* are determined in a number of ways:

- Some names (*Eid, Esetnm, Maptp*) set the basic types and come from the module's nested context. These are common to all the versions of *TYPED-RELATION* used by *NDBRELATION*.
- Parameters to *NDBRELATION* give the typing and normalization information which differentiates specific instances of *TYPED-RELATION* as they arise in *NDB* (for example in modelling the dependent type situation described below).
- Some NDB-specific concepts are *fixed* within *NDBRELATION*, e.g. the binary nature of *NDB* relations is established by fixing a two-value *Fsel* as the attribute set.

The function *tpc* defines a very simple type-checking mechanism based on the supplied type map *esm*. Note, however, that *esm* is an unusual parameter. It is a component of the *NDB* state and is therefore subject to change by operations such as *ADDES*.

A theory can be developed about this specialization of *TYPED-RELATION*:

with *NDBRELATION[fs, ts, esm,* M:M];

$$\boxed{\text{Th5}}\ \frac{t \in \mathit{Tuple};\ r \in \mathit{Relation}}{\mathit{add}(r,t) \in \mathit{Relation}}$$

with *NDBRELATION[fs, ts, esm, maptp]*;

$$\boxed{\text{Th6}}\ \frac{t \in \mathit{Tuple}}{\mathit{add}(\mathit{empty}(), t) \in \mathit{Relation}}$$

3.2.2 *NDBRELATION* in *NDB*

When nested in context in *NDB* (Appendix C.3), *NDBRELATION* is used as a source of types and operators which are re-exported from *TYPED-RELATION*, but with some of *TYPED-RELATION*'s parameters fixed. The full instantiation of *TYPED-RELATION* takes place in the invariant's final conjunct where, in the authors' notation,

$$\mathit{Relation}[fs, ts, esm, tp]$$

refers to

$$\mathit{NDBRELATION}[fs, ts, esm, tp]\,.\mathit{Relation}$$

which in turn refers to

$$\mathit{TYPED\text{-}RELATION}[\mathit{Eid}, \mathit{Esetnm}, \mathit{Fsel}, tm, tpc, \mathit{conv}(mtp), fs, ts, esm, tp]\,.\mathit{Relation}$$

In Section 1 the Relation Map ($\mathit{Rkey} \xrightarrow{m} \mathit{Rinf}$) was introduced. Typing and normalization information about each relation is distributed across the *Rkey* and the *Rinf*. The *Rinf* contains a relation *r* which conforms to that information. In the flat specification, the notion of conformance is made precise in the invariant over the whole database. In the structured specification (Appendix C.3), the typing and normalization mechanisms built into the *TYPED-RELATION* module permit the removal of the definition of conformance from *inv-Ndb*. The invariant now need only state $r \in \mathit{Relation}[fs, ts, esm, tp]$. The type of *r* in *Rinf* should cover all the versions of *Relation* exported by *NDBRELATION*. This is indicated by the [*] notation.

Observe the complexity of the type dependencies involved in the definition of *Ndb*. The type of *r* in *Rinf* is dependent on the value of the *tp* component in the *Rinf*, the *fs* and *ts* values in the *Rkey* and the value of the *esm* component of *Ndb*. If a direct type dependency notation were available in Meta-IV, one might write something like:

$$\begin{array}{llll} \mathit{Rkey} :: & nm & : & [\mathit{Rnm}] \\ & fs & : & \mathit{Esetnm} \\ & ts & : & \mathit{Esetnm} \end{array}$$

$$\begin{array}{llll} \mathit{Rinf}(fs_p, ts_p, esm_p) :: & tp & : & \mathit{Maptp} \\ & r & : & \mathit{Relation}[fs_p, ts_p, esm_p, tp] \end{array}$$

$$\begin{array}{llll} \mathit{Ndb} :: & esm & : & \mathit{Esetnm} \xrightarrow{m} \mathit{Eid}\text{-set} \\ & em & : & \mathit{Eid} \xrightarrow{m} [\mathit{Value}] \\ & rm & : & \mathit{mk\text{-}Rkey}(\mathit{name}, \mathit{from\text{-}set}, \mathit{to\text{-}set}) \colon \mathit{Rkey} \xrightarrow{m} \mathit{Rinf}[\mathit{from\text{-}set}, \mathit{to\text{-}set}, esm] \end{array}$$

where *Rinf* is a type with formal parameters fs_p, ts_p, esm_p, instantiated in the type of the *rm* component of *Ndb*.

NDB may be viewed as importing an indexed class of instances of *TYPED-RELATION* (this is done via *NDBRELATION*). Operators as well as data types exported by these instances must be differentiated from one another by an index. The specification of *ADDREL* (Appendix C.3) illustrates this in the post-condition:

```
ADDREL (nm: [Rnm], fs, ts: Esetnm, tp: Maptp)
ext rd esm  :  $Esetnm \xrightarrow{m} Eid$-set
    wr rm   :  $Rkey \xrightarrow{m} Rinf$
pre ...
post $rm = \overleftarrow{rm} \cup \{mk\text{-}Rkey(nm, fs, ts) \mapsto mk\text{-}Rinf(tp, empty[fs, ts, esm, tp]())\}$
```

The *empty* operator is qualified to indicate which *empty* is required. One must be able to reason about the full range of *empty*s in order to discharge proof obligations. For example, the implementability obligation on *ADDREL* involves quantification over the parameters of *ADDREL* and hence over the parameters indexing *empty*.

The ability to reason about operators from two different sources in the same context is required. The well-formedness of expressions such as the following (which might be part of a specification for a relational *JOIN* operator, for example) is therefore called into question:

```
let $rk_i = mk\text{-}Rkey(nm_i, fs_i, ts_i)$ in
$\{value[fs_1, ts_1, esm, mtp](t_1, f) = value[fs_2, ts_2, esm, mtp](t_2, f) \mid$
    $t_1 \in tuples[fs_1, ts_1, esm, mtp](r(rm(rk_1))) \wedge$
    $t_2 \in tuples[fs_2, ts_2, esm, mtp](r(rm(rk_2)))\}$
```

where *value* and *tuples* are exported operators of the *TYPED-RELATION* module which can be defined in the obvious way.

This discussion raises an interesting question which the authors cannot claim to have resolved satisfactorily. Where would the specifications of the operations of relational algebra be placed in a structure such as this? Some alternatives can be briefly considered:

- The operations could be specified along with the whole database system (in the present case, *NDB*; in Section 4.2, *RDB* or *IS/1*). Expressions involving different indexed versions of the same operators would arise. Anyway, this is hardly an appropriate place for the specification of operators as general as those of relational algebra.
- The operators could not be specified in *TYPED-RELATION* because that module's definitions are specific to relations of one given 'shape'. Thus operators with more than one instance of *Relation* in their signature could not be specified in their full generality.
- Perhaps a separate module for relational algebra is required. This would import the full range of relations of all shapes and would export the fully general operators. The next problem is to determine how that module would be used in, say, *NDB*. It will be necessary to show the applicability of relational algebra operators on relations imported from *NDBRELATION*.

4 Related Topics

4.1 Modular Implementation

It is tempting to assume that a modularization chosen for one purpose will also fulfill other objectives. It would indeed be nice if the decomposition which makes it easiest to understand a system would also serve to control an efficient implementation via separate modules. This objective is strongly underlies [Mor90]. There is certainly no link between the modularization developed here (for understanding the specification) and the actual structure of the product implementation of NDB. In order to dampen enthusiasm for a search for such a modularization, a few of the mismatches are listed here:

- The actual implementation also stores the inverse of any relation so that the user does not experience surprising performance differences between responses to queries.[8]

[8]This, and similar decisions, would make it inappropriate to use NDB for very large databases.

- Entities are generated for each relation so that meta-data (e.g. the *fs*/*ts* information about a relation) can be stored *as relations*.[9]
- In contrast to the preceding point, the link between Entities and Entity sets was handled by special pointers in order to enhance performance.
- Corresponding to each Entity, a record is created which *inter alia* contains a pointer to a vector of pointers: each element in this vector corresponds to one of the relations in which the entity is used as a *fv*. Each of these pointers addresses what is in the general case (but there is an optimization for M:1 and 1:1 relations) a vector of 'C-element' pointers where each C-element points to the Entity required as *tv*. This design was clearly tailored to the special case of binary relations, but – it must be realized – confounds any separation of notions like Entity and Relation.[10]

Work on formally specifying, and implementing by refinement, another database system ('MDB', which was the database underlying the 'Mule' system) bears out the message that it is rarely possible, (for non-trivial problems) to obtain an efficient implementation by slavishly following the module structure of the specification. This should actually come as no surprise to anyone with experience of specifications. Who, for example, would begin implementing GCD (Greatest Common Divisor), from a specification given in terms of the intersection of the sets of divisors of the two numbers, by building set operators? Specifications describe what a system is to do; not how to do it. This old battle cry applies just as much to modularization as it does to post-conditions (see related arguments in [HJ89]).

4.2 Re-use of *TYPED-RELATION*

In an attempt to illustrate its potential for re-use, the generalized *TYPED-RELATION* module has been used to specify two other database systems. The specifications (Appendix D) are briefly introduced below.

4.2.1 RDB

RDB (Appendix D.1) illustrates generalization to an *n*-ary relational database with a different normalization scheme to that of NDB. Only simple entity typing is used.

The whole database specification module *RDB* is parameterized over relevant primitive types and an entity type-checking function. The nested module *RDBRELATION* forms the interface to *TYPED-RELATION* in a way analogous to *NDBRELATION*. It differs in that it fixes a typing mechanism (*tpc* comes from context) and parameterizes over a relationship type rather than a map type.

Within *RDBRELATION*, *rtp* is interpreted as a pair consisting of a set of attributes participant in a relation key, the values of which determine the values of attributes in the second of the pair. The appropriate form of relation is imported.

RDB works in much the same way as *NDB*, but note that *Fsel* is a parameter. Each relation's attribute set is the domain of its type map *tp*. A subset of the attributes are distinguished as participants in the *key*.

4.2.2 IS/1

The description of IS/1 (see Appendix D.2) introduces a primitive form of indirection into the type checking mechanism. *IS1RELATION* is very simple: it only restricts the parameter list of *TYPED-RELATION* by fixing context. The two-level hierarchical typing is done entirely in *IS*/1 where a type name (from *Tpnm*) maps to an entity type (from *Etp*). The indirection is resolved in the reference to *IS1RELATION* in the invariant. Extension to other typing systems may be approached similarly.

[9]In fact, an implementation can be designed which requires little more than a 'triple store' with an associative search operation.

[10]The informal 'specification' of NDB actually consists a collection of pictures explaining these pointers.

5 Summary

The NDB example is a useful reference point because it shows that the challenges which are identified are not contrived. The modularization proposed in Section 3 requires a number of features which are not supported by (at least) the current module proposal for BSI-VDM. It is therefore of interest to see which other specification languages can cope with our modularization.[11] The main features which are required above are:

- separate modules;
- parameterized modules;
- the ability to nest modules so as to instantiate (some of the) parameters of a nested module from its context;
- some form of dependent types (regardless of whether the constraints are expressed by general predicates or by some special-purpose notation);
- the ability to create multiple instances of one module within another;
- ways of relating operators to the appropriate instances of multiply instantiated modules.

These points can be illustrated, albeit with less conviction, around an example based on the ubiquitous stack. A general stack can be defined which is parameterized both on a type and on a maximum depth.

Module *STACK*
Parameters
  types $X : Triv$
  values $m : \mathbb{N}$
Exports
  types *Stack*
  functions *empty, is-full, push*
Definitions
  types

$Stack = X^*$

$\text{inv}\,(s) \triangleq \text{len}\, s \leq m$

$empty : \to Stack$

$empty() \;\triangleq\; [\,]$

$is\text{-}full : Stack \to \mathbb{B}$

$is\text{-}full(s) \;\triangleq\; \text{len}\, s = m$

[11] Our chosen structuring is, of course, also open to counter-proposals. It does however seem reasonable to ask whether our structure can be handled by a specification language before being shown an 'improved modularization' which does happen to suit that language well. It must also be said that seeing earlier splits which illustrated little more than that various finite maps could be axiomatized was one of our initial stimuli to find a meaningful structuring of the specification.

```
    $push : X \times Stack \to Stack$
    $push(e, s) \;\triangleq\; [e] \frown s$
    pre $\neg\, is\text{-}full(s)$

End  $STACK$
```

A *USER* module, which is parameterized on a type, has an embedded *YSTACK* which partially instantiates *STACK* by resolving its type parameter: *YSTACK* is therefore only parameterized on its maximum depth. *USER* has a type which uses multiple instances of *YSTACK* and which has an invariant linking the domain value with the instance.

```
Module  $USER$
Parameters
   types  $Y : Triv$
Exports
   ...
Local Modules
      Module  $YSTACK$
      Parameters
         values  $n : \mathbb{N}$
      Exports
         types  $Stack$
         functions  $empty, isfull, push$
      Definitions
         types
            $Stack = STACK[Y, n].Stack$
      ⋮
      End  $YSTACK$
Definitions
   types
   $StackMap = \mathbb{N} \xrightarrow{m} Stack[*]$
   inv $(mk\text{-}StackMap(m)) \triangleq \forall i \in \text{dom}\, m \cdot m(i) \in Stack[i]$
⋮
End  $USER$
```

Further Work

The problem of preparing a modularization of the flat *NDB* specification in the style of Section 3.2 and Appendix C is presented as a 'challenge' for a existing specification languages. Responses have already been received in COLD-K [Fei89] and RSL[Geo89], as have comments on approaches in BSI-VDM[Bea89]. The authors consider responses to date have tended to use the approaches free of multiple instances of imported modules rather than that of Section 3.2. Responses to the problems of this latter style are particularly invited.

This paper has suggested specification language features which are considered desirable in solving this structuring problem. One intention behind offering this 'challenge' is to discover whether other languages can supply these features, and what the consequences of the provision of such features are for the language semantics. This continues to be the main area of ongoing research for one of the authors (JSF).

Acknowledgements

The authors are grateful to the METEOR project for the invitation to CBJ to present this work at their final workshop and for Loe Feijs's continued interest in the paper. CBJ also acknowledges stimulating discussions with Søren Prehn, Chris George, Peter Haff and Robert Milne on an early version of this problem whose 'flat' formulation was due to Ann Walshe. Andrzej Tarlecki, Martin Wirsing, and Don Sannella have offered useful comments on our evolving ideas, while Juan Bicarregui, Bob Fields, Ian Hayes, Andrew Malton and anonymous referees have provided helpful review comments on drafts of the present paper. The authors gratefully acknowledge financial support for their research: CBJ from the SERC and Wolfson Foundation; JSF from the Department of Education for Northern Ireland.

References

[And88] D.J. Andrews. Report from the BSI panel for the standardisation of VDM (IST/5/50). In *[BJM88]*, pages 74–78, 1988.

[Bea88] S. Bear. Structuring for the VDM specification language. In *[BJM88]*, 1988.

[Bea89] S. Bear. Private communication. e-mail, December 1989.

[BJ82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Prentice Hall International, 1982.

[BJM88] R. Bloomfield, R. B. Jones, and L. S. Marshall, editors. *VDM '88: VDM – The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

[BSI] BSI. *VDM Specification Language – Proto-Standard*. BSI Working Document: BSI IST/5/50 Document No. 40.

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

[Fei89] L. Feijs. Private communication. e-mail, September 1989.

[Geo89] C. George. Private communication. e-mail, December 1989.

[Haf89] P. Haff. RSL Syntax Summary. Technical Report RAISE/DDC/PH/98/V3, Dansk Datamatik Center, October 1989.

[HJ89] I. J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):320–338, November 1989.

[IBM] IBM. *Data Mapping Program: User's Guide*. SB11 - 5340.

[Jon80] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon88] H.B.M. Jonkers. An introduction to COLD-K. Technical Report METEOR/t8/PRLE/8, Philips Research Labs, Eindhoven, July 1988.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM (2nd Edition).* Prentice Hall International, 1990.

[JS90] C.B. Jones and R.C.F. Shaw, editors. *Case Studies in Systematic Software Development.* Prentice Hall International, 1990.

[Mor90] J. M. Morris. A methodology for designing and refining specifications. In J. Woodcock, editor, *Proceedings of the 3rd Workshop on Refinement* . Springer-Verlag (BCS Workshop Series), 1990.

[Sha90] R. C. F. Shaw. The ISTAR database. In *[JS90]*, 1990.

[Wal90] A. Walshe. NDB: The formal specification and rigorous design of a single-user database system. In *[JS90]*, 1990.

[WS79] N. Winterbottom and G. C. H. Sharman. NDB: Non-programmer database facility. Technical Report IBM TR.12.179, IBM Hursley Laboratory, England, September 1979.

A The Flat NDB Specification

```
Module  $FLAT\text{-}NDB$
Parameters
  types  $Eid, Value, Esetnm, Rnm : Triv$
Exports
  operations  $ADDES, ADDENT, ADDREL, ADDTUP, \ldots$
Definitions
  types
   $Maptp = \{1{:}1, 1{:}M, M{:}1, M{:}M\}$

   $Tuple :: fv : Eid$
             $tv : Eid$

   $Rinf :: tp : Maptp$
             $r : Tuple\text{-set}$

   $Rkey :: nm : [Rnm]$
             $fs : Esetnm$
             $ts : Esetnm$
  state
   $Ndb :: esm : Esetnm \xrightarrow{m} Eid\text{-set}$
            $em : Eid \xrightarrow{m} [Value]$
            $rm : Rkey \xrightarrow{m} Rinf$
   inv $(mk\text{-}Ndb(esm, em, rm)) \triangleq$
      dom $em = \bigcup$ rng $esm \wedge$
      $\forall rk \in$ dom $rm \cdot$
          let $mk\text{-}Rkey(nm, fs, ts) = rk$ in
          let $mk\text{-}Rinf(tp, r) = rm(rk)$ in
          $\{fs, ts\} \subseteq$ dom $esm \wedge$
          $(tp = 1{:}M \Rightarrow \forall t_1, t_2 \in r \cdot tv(t_1) = tv(t_2) \Rightarrow fv(t_1) = fv(t_2)) \wedge$
          $(tp = M{:}1 \Rightarrow \forall t_1, t_2 \in r \cdot fv(t_1) = fv(t_2) \Rightarrow tv(t_1) = tv(t_2)) \wedge$
          $(tp = 1{:}1 \Rightarrow \forall t_1, t_2 \in r \cdot fv(t_1) = fv(t_2) \Leftrightarrow tv(t_1) = tv(t_2)) \wedge$
          $\forall mk\text{-}Tuple(fv, tv) \in r \cdot fv \in esm(fs) \wedge tv \in esm(ts)$
```

init(ndb) $\triangleq$ $ndb = mk\text{-}Ndb(\{\,\}, \{\,\}, \{\,\})$

operations

ADDES $(es\colon Esetnm)$

ext wr $esm \;:\; Esetnm \xrightarrow{m} Eid$-set

pre $es \notin$ dom esm

post $esm = \overleftarrow{esm} \cup \{es \mapsto \{\,\}\}$

DELES $(es\colon Esetnm)$

ext wr $esm \;:\; Esetnm \xrightarrow{m} Eid$-set
    rd $rm \;:\; Rkey \xrightarrow{m} Rinf$

pre $es \in$ dom $esm \wedge esm(es) = \{\,\} \wedge$
    $es \notin \{fs(rk) \mid rk \in \text{dom}\, rm\} \cup \{ts(rk) \mid rk \in \text{dom}\, rm\}$

post $esm = \{es\} \mathbin{\lhd\!\!\!-} \overleftarrow{esm}$

ADDENT $(memb\colon Esetnm$-set, $val\colon [Value])\; eid\colon Eid$

ext wr $esm \;:\; Esetnm \xrightarrow{m} Eid$-set
    wr $em \;:\; Eid \xrightarrow{m} [Value]$

pre $memb \subseteq$ dom esm

post $eid \notin$ dom $\overleftarrow{em} \wedge$
    $em = \overleftarrow{em} \cup \{eid \mapsto val\} \wedge$
    $esm = \overleftarrow{esm} \dagger \{es \mapsto \overleftarrow{esm}(es) \cup \{eid\} \mid es \in memb\}$

DELENT $(eid\colon Eid)$

ext wr $esm \;:\; Esetnm \xrightarrow{m} Eid$-set
    wr $em \;:\; Eid \xrightarrow{m} [Value]$
    rd $rm \;:\; Rkey \xrightarrow{m} Rinf$

pre $eid \in$ dom $em \wedge$
    $\forall t \in \bigcup\{r(ri) \mid ri \in \text{rng}\, rm\} \cdot fv(t) \neq eid \wedge tv(t) \neq eid$

post $esm = \{es \mapsto \overleftarrow{esm}(es) - \{eid\} \mid es \in \text{dom}\, \overleftarrow{esm}\} \wedge$
    $em = \{eid\} \mathbin{\lhd\!\!\!-} \overleftarrow{em}$

ADDREL $(rk\colon Rkey, tp\colon Maptp)$

ext rd $esm \;:\; Esetnm \xrightarrow{m} Eid$-set
    wr $rm \;:\; Rkey \xrightarrow{m} Rinf$

pre $\{fs(rk), ts(rk)\} \subseteq$ dom $esm \wedge$
    $rk \notin$ dom rm

post $rm = \overleftarrow{rm} \cup \{rk \mapsto mk\text{-}Rinf(tp, \{\,\})\}$

DELREL $(rk\colon Rkey)$

ext wr $rm \;:\; Rkey \xrightarrow{m} Rinf$

pre $rk \in$ dom $rm \wedge r(rm(rk)) = \{\,\}$

post $rm = \{rk\} \mathbin{\lhd\!\!\!-} \overleftarrow{rm}$

$ADDTUP\ (fval, tval{:}\ Eid, rk{:}\ Rkey)$

ext wr $rm \ : \ Rkey \xrightarrow{m} Rinf$

rd $esm \ : \ Esetnm \xrightarrow{m} Eid$-set

pre $rk \in$ dom $rm \wedge$

let $ri = \mu(rm(rk), r \mapsto r(rm(rk)) \cup \{mk\text{-}Tuple(fval, tval)\})$ in

inv-$Ndb(mk\text{-}Ndb(esm, em, rm \dagger \{rk \mapsto ri\}))$

post let $ri = \mu(\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) \cup \{mk\text{-}Tuple(fval, tval)\})$ in

$rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$

$DELTUP\ (fval, tval{:}\ Eid, rk{:}\ Rkey)$

ext wr $rm \ : \ Rkey \xrightarrow{m} Rinf$

pre $rk \in$ dom rm

post let $ri = \mu(\overleftarrow{rm}(rk), r \mapsto r(\overleftarrow{rm}(rk)) - \{mk\text{-}Tuple(fval, tval)\})$ in

$rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$

End *FLAT-NDB*

B Simple Modularization

B.1 *RELATION*

Module *RELATION*
Parameters
  types *E, Attr : Triv*
Exports
  types *Tuple, Relation*
  functions *create, value, empty, add, rem, attr-match, tuples, ...*
Definitions
  types

$Tuple = Attr \xrightarrow{m} E$

$Relation = Tuple$-set

inv $(r) \triangleq \forall t_1, t_2 \in r \cdot \mathrm{dom}\, t_1 = \mathrm{dom}\, t_2$

functions

$create : (Attr \xrightarrow{m} E) \to Tuple$

$create(m) \quad \triangleq \quad m$

$value : Tuple \times Attr \to E$

$value(t, a) \quad \triangleq \quad t(a)$

pre $a \in$ dom t

$empty : \to Relation$

$empty() \quad \triangleq \quad \{\,\}$

$add : Relation \times Tuple \to Relation$

$add(r, t) \quad \triangleq \quad r \cup \{t\}$

pre $inv\text{-}Relation(r \cup \{t\})$

$rem : Relation \times Tuple \to Relation$

$rem(r, t) \quad \triangleq \quad r - \{t\}$

$attr\text{-}match : Relation \times Attr\text{-set} \to \mathbb{B}$

$attr\text{-}match(r, as) \quad \triangleq \quad \forall t \in r \cdot \mathrm{dom}\, t = as$

$tuples : Relation \to Tuple\text{-set}$

$tuples(r) \quad \triangleq \quad r$

⋮

End *RELATION*

B.2 *NDB* using *RELATION*

Module *FIRST-NDB*
Parameters
  types *Eid, Value, Esetnm, Rnm : Triv*
Exports
  operations *ADDES, ADDENT, ADDREL, ADDTUP, …*

Definitions
  types

$Fsel = \{\mathrm{Fs}, \mathrm{Ts}\}$

Imports
  all from *RELATION*[*Eid, Fsel*]
Definitions
  types

$Maptp = \{1{:}1, 1{:}\mathrm{M}, \mathrm{M}{:}1, \mathrm{M}{:}\mathrm{M}\}$

```
Rkey :: nm : [Rnm]
        fs : Esetnm
        ts : Esetnm

Rinf :: tp : Maptp
        r  : Relation[Eid, Fsel]
```

```
state
$Ndb$ :: $esm$ : $Esetnm \xrightarrow{m} Eid$-set
         $em$ : $Eid \xrightarrow{m} [Value]$
         $rm$ : $Rkey \xrightarrow{m} Rinf$
inv $(mk\text{-}Ndb(esm, em, rm)) \triangleq$
    dom $em = \bigcup$ rng $esm \wedge$
    $\forall rk \in$ dom $rm \cdot$
        let $mk\text{-}Rkey(nm, fs, ts) = rk$ in
        let $mk\text{-}Rinf(tp, r) = rm(rk)$ in
        $attrs\text{-}match(r, Fsel) \wedge$
        $\{fs, ts\} \subseteq$ dom $esm \wedge$
        $(tp = 1{:}\mathrm{M} \Rightarrow$
          $\forall t_1, t_2 \in tuples(r) \cdot$
            $value(t_1, \mathrm{TS}) = value(t_2, \mathrm{TS}) \Rightarrow value(t_1, \mathrm{FS}) = value(t_2, \mathrm{FS})) \wedge$
        $(tp = \mathrm{M}{:}1 \Rightarrow$
          $\forall t_1, t_2 \in tuples(r) \cdot$
            $value(t_1, \mathrm{FS}) = value(t_2, \mathrm{FS}) \Rightarrow value(t_1, \mathrm{TS}) = value(t_2, \mathrm{TS})) \wedge$
        $(tp = 1{:}1 \Rightarrow$
          $\forall t_1, t_2 \in tuples(r) \cdot$
            $value(t_1, \mathrm{FS}) = value(t_2, \mathrm{FS}) \Leftrightarrow value(t_1, \mathrm{TS}) = value(t_2, \mathrm{TS})) \wedge$
        $\forall t \in tuples(r) \cdot value(t, \mathrm{FS}) \in esm(fs) \wedge value(t, \mathrm{TS}) \in esm(ts)$

init$(ndb)$  $\triangleq$  $ndb = mk\text{-}Ndb(\{\,\}, \{\,\}, \{\,\})$

operations
  ⋮

ADDTUP $(fval, tval{:}\ Eid, rk{:}\ Rkey)$
ext wr $rm$  : $Rkey \xrightarrow{m} Rinf$
    rd $esm$ : $Esetnm \xrightarrow{m} Eid$-set
pre $rk \in$ dom $rm \wedge$
    let $ri = \mu(rm(rk), r \mapsto add(r(rm(rk)), create(\{\mathrm{FS} \mapsto fval, \mathrm{TS} \mapsto tval\})))$ in
    inv-$Ndb(mk\text{-}Ndb(esm, em, rm \dagger \{rk \mapsto ri\}))$
post let $ri = \mu(\overleftarrow{rm}(rk), r \mapsto add(r(\overleftarrow{rm}(rk)), create(\{\mathrm{FS} \mapsto fval, \mathrm{TS} \mapsto tval\})))$ in
      $rm = \overleftarrow{rm} \dagger \{rk \mapsto ri\}$

  ⋮
End  FIRST-NDB
```

C Modularization with Typing and Normalization

C.1 *TYPED-RELATION*

Module *TYPED-RELATION*
Parameters
  types $E, Etp, Attr : Triv$
  values $tpm : Attr \xrightarrow{m} Etp$
  functions $tpc : E \times Etp \rightarrow \mathrm{B}$
  values $norm : (Attr\text{-set} \times Attr)\text{-set}$
Exports
  types $Tuple, Relation$
  functions $create, value, empty, add, rem, \ldots$
Definitions
  types

$Tuple = Attr \xrightarrow{m} E$

inv $(m) \triangleq \mathrm{dom}\, m = \mathrm{dom}\, tpm \wedge \forall a \in \mathrm{dom}\, tpm \cdot tpc(m(a), tpm(a))$

$Relation = Tuple\text{-set}$

inv $(r) \triangleq \forall (s,f) \in norm \cdot \forall t_1, t_2 \in r \cdot s \lhd t_1 = s \lhd t_2 \;\Rightarrow\; t_1(f) = t_2(f)$

  functions

$create : (Attr \xrightarrow{m} E) \rightarrow Tuple$

$create(m) \quad \triangleq \quad m$

pre inv-$Tuple(m)$

$value : Tuple \times Attr \rightarrow E$

$value(t, a) \quad \triangleq \quad t(a)$

pre $a \in \mathrm{dom}\, t$

$empty : \; \rightarrow Relation$

$empty() \quad \triangleq \quad \{\,\}$

$add : Relation \times Tuple \rightarrow Relation$

$add(r, t) \quad \triangleq \quad r \cup \{t\}$

pre $inv\text{-}Relation(r \cup \{t\})$

$rem : Relation \times Tuple \rightarrow Relation$

$rem(r, t) \quad \triangleq \quad r - \{t\}$

  ⋮

End *TYPED-RELATION*

C.2 *TYPED-RELATION* specialized to *NDBRELATION*

```
    let $Eid, Esetnm: Triv$ in
    let $Maptp = \{1{:}1, 1{:}M, M{:}1, M{:}M\}$ in


Module  NDBRELATION
Parameters
    values
    $fs, ts: Esetnm,$
    $esm: Esetnm \xrightarrow{m} Eid$-set,
    $mtp: Maptp$
Definitions
    types
     $Fsel = \{\text{FS}, \text{TS}\}$

     $Norm = (Fsel\text{-set} \times Fsel)\text{-set}$

    functions

     $conv : Maptp \to Norm$

     $conv(ty) \;\triangleq$  cases $ty$ of
                      $M{:}M \to \{\}$,
                       $M{:}1 \to \{(\{\text{FS}\}, \text{TS})\}$,
                       $1{:}M \to \{(\{\text{TS}\}, \text{FS})\}$,
                        $1{:}1 \to \{(\{\text{TS}\}, \text{FS}), (\{\text{FS}\}, \text{TS})\}$
                      end


     $tpc : Eid \times Esetnm \to \mathbf{B}$

     $tpc(eid, esn) \;\triangleq\; eid \in esm(esn)$

    values

     $tm = \{\text{FS} \mapsto fs, \text{TS} \mapsto ts\}$
Imports
    all from   $TYPED\text{-}RELATION[Eid, Esetnm, Fsel, tm, tpc, conv(mtp)]$
Exports
    types   $Tuple, Relation$
    functions   $create, value, empty, add, rem, \ldots$

End  NDBRELATION
```

C.3 The Final Structuring of *NDB*

```
Module  NDB
Parameters
    types  $Eid, Value, Esetnm, Rnm : Triv$
```

```
Imports
   ...
Exports
   ...
Definitions
   types
    $Maptp = \{1{:}1, 1{:}M, M{:}1, M{:}M\}$
Local Modules
      Module  $NDBRELATION$
      ...
      End  $NDBRELATION$
   types
    $Rkey$ :: $nm$ : $[Rnm]$
              $fs$ : $Esetnm$
              $ts$ : $Esetnm$

    $Rinf$ :: $tp$ : $Maptp$
              $r$ : $Relation[*]$
   state
    $Ndb$ :: $esm$ : $Esetnm \xrightarrow{m} Eid\text{-set}$
             $em$ : $Eid \xrightarrow{m} [Value]$
             $rm$ : $Rkey \xrightarrow{m} Rinf$
   inv $(mk\text{-}Ndb(esm, em, rm)) \triangleq$
       dom $em = \bigcup$ rng $esm \wedge$
       $\forall rk \in$ dom $rm \cdot$
           let $mk\text{-}Rkey(nm, fs, ts) = rk$ in
           let $mk\text{-}Rinf(tp, r) = rm(rk)$ in
           $\{fs, ts\} \subseteq$ dom $esm \wedge$
           $r \in Relation[fs, ts, esm, tp]$

   init$(ndb)$  $\triangleq$  $ndb = mk\text{-}Ndb(\{\,\}, \{\,\}, \{\,\})$

   operations
   ⋮

   $ADDREL\ (nm{:}\ [Rnm], fs, ts{:}\ Esetnm, tp{:}\ Maptp)$
   ext rd $esm$ : $Esetnm \xrightarrow{m} Eid\text{-set}$
       wr $rm$ : $Rkey \xrightarrow{m} Rinf$
   pre ...
   post $rm = \overleftarrow{rm} \cup \{mk\text{-}Rkey(nm, fs, ts) \mapsto mk\text{-}Rinf(tp, empty[fs, ts, esm, tp]())\}$

   ⋮
End  $NDB$
```

D Re-use of *TYPED-RELATION*

D.1 RDB

```
Module  $RDB$
Parameters
   types  $Value, Etp, Fsel, Rnm : Triv$
   functions  $tpc\colon Value \times Etp \to \mathbb{B}$
Exports
   ...
Definitions
   types

   $Norm = (Fsel\text{-set} \times Fsel)\text{-set}$

Local Modules
      Module  $RDBRELATION$
      Parameters
         values  $tpm\colon Fsel \xrightarrow{m} Etp,$
                 $rtp\colon Fsel\text{-set} \times Fsel\text{-set}$
      Definitions
         functions

         $nf : Fsel\text{-set} \times Fsel\text{-set} \to Norm$

         $nf(ks, rs) \;\;\triangleq\;\; \{(ks, r) \mid r \in rs\}$

      Imports
         all from  $TYPED\text{-}RELATION[Value, Etp, Fsel, tpm, tpc, nf(rtp)]$
      End  $RDBRELATION$
Definitions
   types

   $Rinf$ ::  $tp$  : $Fsel \xrightarrow{m} Etp$
             $key$  : $Fsel\text{-set}$
               $r$  : $Relation[*]$

   inv $(mk\text{-}Rinf(tp, key, r)) \triangleq key \subseteq \text{dom}\, tp$

   state

   $Rdb = Rnm \xrightarrow{m} Rinf$

   inv $(rdb) \triangleq \forall mk\text{-}Rinf(tp, key, r) \in \text{rng}\, rdb \cdot r \in Relation[tp, (key, \text{dom}\, tp\text{-}key)]$

   ⋮
End  $RDB$
```

D.2 IS/1

```
Module  $IS/1$
Parameters
   types  $Value, Etp, Fsel, Tpnm, Rnm : Triv$
   functions  $tpc\colon Value \times Etp \to \mathbb{B}$
Exports
   ...
Definitions
   values
    $norm_0 = \{\,\}$
    Local Modules
       Module  $IS1RELATION$
       Parameters
          values  $tpm\colon Fsel \xrightarrow{m} Etp$
       Imports
       all from  $TYPED\text{-}RELATION[Value, Etp, Fsel, tpm, tpc, norm_0]$
       End  $IS1RELATION$
Definitions
   types
   $Rinf :: tp : Fsel \xrightarrow{m} Tpnm$
              $r : Relation[*]$
   state
   $Is1 :: tm : Tpnm \xrightarrow{m} Etp$
           $rm : Rnm \xrightarrow{m} Rinf$
   inv $(mk\text{-}Is1(tm, rm)) \triangleq$
      $\forall mk\text{-}Rinf(tp, r) \in \text{rng}\, rm \cdot$
            $\text{rng}\, tp \subseteq \text{dom}\, tm \wedge$
            $r \in Relation[tm \circ tp]$

   ⋮
End  $IS/1$
```