On the Derivation of Executable Database Programs from Formal Specifications*

Thomas Günther, Klaus-Dieter Schewe, Ingrid Wetzel

University of Hamburg, Dept. of Computer Science,
Vogt-Kölln-Str. 30, D-W-2000 Hamburg 54, FRG

Abstract. Achieving wide acceptance of formal methods in software development requires a smooth integration with requirements analysis, design and implementation. Especially for database application systems there exist well-known approaches to conceptual modeling as well as a sophisticated implementation technology on the basis of database programming languagues. The work described in this paper is based on a scenario, where the B method is coupled with a conceptual modeling language TDL and a database programming language DBPL. Both these languages can be represented in B. We concentrate on the problem of characterizing those B specifications that are sufficiently refined in order to be transformed into equivalent DBPL programs. This gives rise to some kind of *implementability proof obligation*. Moreover, we show that the *transformation* itself can be regarded as a term rewriting task based on a representation by term algebras of the languages involved. For this task we exploit order-sorted algebra by using the OBJ system.

1 Introduction

Formal Methods such as B [1], VDM [8], OBJ [4] or Z [14] have often been criticized as being cumbersome, hard to understand, hard to handle and in any case not suitable for real application systems of large size. The core of the problem seems to be that an accompanying methodology is almost always left unclear. How should the user proceed in order to get a first formal specification? Which refinement steps should be applied? Where should one stop the refinement process? Therefore, a smooth integration with requirements analysis, design and implementation is required to enhance the acceptability of formal methods in software development.

Our approach to solve this problem assumes the coupling of a formal method with a "front-end" design language and a "back-end" implementation language. We believe that requirements analysis and first design should be oriented toward the application domain. There will be no loss of formal software safety as long as the semantics of the used design language can be described by the formal method.

On the other hand, formal methods will fail their goals, if they ignore the high-level mechanisms offered by modern programming languages. It should be possible to capture the semantics of such implementation languages by the formal method and to exploit this to combine the safety achieved by the use of a formal method with the efficiency achieved by the use of a sophisticated implementation technology.

* This work has been supported in part by research grants from the E.E.C. Basic Research Action 3070 FIDE: "Formally Integrated Data Environments".

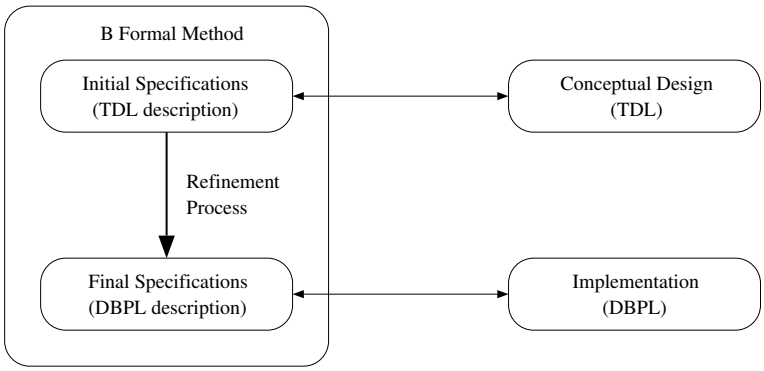

Fig. 1. The Development Process

1.1 A Scenario for the Development of Data Intensive Application Systems

The development of database application systems usually starts with a conceptual design of a database schema. Semantic datamodels [6] are commonly used for this purpose. On the other hand, the implementation is usually done using a relational DBMS or a database programming language such as DBPL [9]. Then the *reification problem* occurs, i.e. to transform a given conceptual design into a running implementation.

The DAIDA project [2, 7] proposed to use formal methods to fill this gap. This is in accordance with the general approach described above, using a slightly revised version (TDL) of TAXIS [10] as a "front-end", DBPL as a "back-end" and a slight revision of the B formal method [12]. Figure 1 illustrates this scenario.

The transformation of TDL designs into a B representation and standard refinement rules directed towards an implementation in DBPL have been described in [13] and will not be repeated here. Then the following problems remain:

- characterize final B specifications that are equivalent to DBPL programs, and
- set up an automatic transformation of final specifications into DBPL syntax.

In order to solve the first problem, we have to show that DBPL programs are indeed equivalent to certain B specifications. We express DBPL language constructs by pieces of B. Then the characterization consists in some general properties such as determinism, termination and consistency that are independent from the specific implementation language and in a specific form required for the data description.

1.2 An Algebraic Approach to Program Generation

In order to solve the second problem we use OBJ, an algebraic specification language and term rewriting system based on order-sorted algebra. It has been pointed out [4] that order-sorted algebra is useful for the specification of programming languages,

where terms represent language constructs and conditional equations represent semantic equivalences.

In our case the languages in question are DBPL and B, where constructs of the former one are equivalent to specific constructs of the latter one. We may exploit the ordering on sorts to capture this inclusion. Moreover, the conditional equations are given by the representation of DBPL in B as mentioned above.

Regarding each conditional equation as a rewrite rule and taking a complex term that represents a final B specification OBJ will produce a normal form term, i.e. a term representing a DBPL program.

1.3 The Organization of the Paper

The remainder of the paper is organized as follows. After a short review of the B formal method in Section 2 without discussing refinement we address the derivation of an implementability proof obligation in Section 3. We briefly describe the implementation language DBPL and then show the representation of selected language constructs using B. On this basis we are able to characterize sufficiently refined formal specifications.

Section 4 is devoted to the transformation task. After a brief introduction into order-sorted algebra on the basis of OBJ, we proceed to describe in part the algebra associated with B and DBPL. For selected language constructs this sets up the term rewriting rules used for the transformation. We conclude with a short valuation of what has been gained by our approach focussing on the suitability of the chosen formal method for our problem.

2 Specifications Using the B Formal Method

B is a model-based formal method developed by J. R. Abrial [1]. Basically a B specification is composed of a specification of structure and behaviour. The main difference to its forerunners VDM [8] and Z [14] are the style of operation specification and the coupling of structure and behaviour specification in basic units called *abstract machines*.

2.1 Specification of Structure

The structural part of a B specification consists of a collection of abstract machines and contexts that are used to specify state spaces. A state space is given by a list of variable names, called the *state variables* and by a list of well-founded formulas of a many-sorted first-order language $\mathcal{L}$ called the *invariant* and denoted by $\mathcal{I}$. Free variables occurring in $\mathcal{I}$ must be state variables.

Each state variable belongs to a unique *basic set*, which has to be declared in some context. Hence, in order to complete the state space specification we must give a list of *contexts* that can be seen by the machine.

Such a context is defined by a list of *basic sets*, a list of *constant names* and a list of closed formulae over the language $\mathcal{L}$ called *properties*. A basic set may be either the set of natural numbers, an abstract set given only by its name, a set given by the

enumeration of its elements or a constructed set, where cartesian product, powerset and partial function space are the only allowed constructors. We may then assume to have a fixed preinterpretation of these sorts s by sets $\mathcal{D}_s$.

Then the state space of an abstract machine with state variables $x_1, \ldots, x_n$ is semantically denoted by the set

$$\Sigma \;=\; \{\sigma : \{x_1, \ldots, x_n\} \rightarrow \mathcal{D} \mid \sigma(x_i) \in D_{s_i} \; for \;\; all \;\; i\} \;,$$

where each s_i is the sort of the variable x_i and $\mathcal{D}$ denotes the union of the sets $\mathcal{D}_s$.

The language $\mathcal{L}$ associated with an abstract machine can then easily be formalized. The basic sorts of $\mathcal{L}$ are NAT and the other non-constructed basic sorts. The set of sorts is recursively defined using the basic sorts and the sort constructors pow, $\times$, $\mapsto$ denoting powerset construction, cartesian products and partial functions.

Since we use a fixed preinterpretation of the sorts as sets, one may regard the elements of these sets as constant symbols in $\mathcal{L}$ of the corresponding sort. Other function symbols are given by the usual functions $+$, $*$ on NAT, $\cup, \cap, \setminus$ on powersets or by the constant declarations in some context. The terms and formulas in $\mathcal{L}$ are defined in the usual way. The semantics of $\mathcal{L}$ is given by an interpretation $(\mathcal{A}, \sigma)$, where $\mathcal{A}$ is a structure extending the preinterpretation on sorts and σ is a variable binding.

We assume $\mathcal{A}$ to be fixed and write $\models_\sigma \mathcal{R}$, iff $\mathcal{R}$ is true under the interpretation $(\mathcal{A}, \sigma)$.

A well-formed formula $\mathcal{R}$ of $\mathcal{L}$ such that the free variables of $\mathcal{R}$ are state variables denotes a subset of Σ, namely

$$\Sigma_{\mathcal{R}} \;=\; \{\sigma \mid\mid\models_\sigma \mathcal{R}\} \;.$$

Hence the invariant serves as a means to distinguish legal states in Σ_I from others.

2.2 Specification of Behaviour

The dynamic part of a B specification is given through an *initialization* assigning initial values to each of the state variables of an abstract machine and *transitions* that update this state space. Both parts are associated with abstract machines. Both kinds of state transitions are specified using guarded commands introduced by Dijkstra [3] and generalized by G. Nelson [11].

The semantics of a transition S is given by means of two specific predicate transformers $wlp(S)$ and $wp(S)$, which satisfy the *pairing condition*, i.e. for all predicates $\mathcal{R}$

$$wp(S)(\mathcal{R}) \;\equiv\; wlp(S)(\mathcal{R}) \;\wedge\; wp(S)(true)$$

and the *universal conjunctivity condition*, which states for any family $(R_i)_{i \in I}$ of predicates

$$wlp(S)(\forall i \in I \cdot R_i) \;\equiv\; \forall i \in I \cdot wlp(S)(R_i) \;.$$

These conditions imply the conjunctivity of $wp(S)$ over non-empty families of predicates. As usual $wlp(S)$ will be called the *weakest liberal precondition* of S, and $wp(S)$

will be called the *weakest precondition* of S. The notation f^*, which we shall use later, denotes the *conjugate predicate transformer* of f. It is defined by

$$f^*(\mathcal{R}) \equiv \neg f(\neg\mathcal{R}) .$$

The definitions of $wlp(S)$ and $wp(S)$ for all the guarded commands are given in [11].

3 The Formal Description of Database Programs

Let us now proceed to characterize sufficiently refined B specifications with respect to a follow-on implementation in the relational database programming language DBPL. First we give a brief overview on DBPL and then outline its representation in B. We focus on selected DBPL language constructs. On this basis we give a proof obligation for finalizing the refinement process.

3.1 An Overview on the Language DBPL

The language DBPL [9] integrates an extended relational view of database modeling into the programming language Modula-2. Basically the extension comprises new *data types*, new expressions called *access expressions* used for queries and orthogonal *persistence*.

- DBPL provides a new data type *relation* which allows relational database modeling to be coupled with the expressiveness of the programming language. This new datatype is orthogonal to the existing types of Modula-2, hence sets of arrays, arrays of relations, records of relations, etc. can be modeled.
- Complex *access expressions* as usual in relational databases allow to express complex queries on a database without using iteration.
- Persistence is added allowing modules to be qualified as database modules, keyword *DATABASE*, which turns the variables in them to be persistent and shared. Specific procedures are characterized to be *transactions* denoting atomic state changes on persistent data. For these transactions DBPL provides mechanisms for controlled concurrent access to such data and for recovery.

Example 1. Let us illustrate DBPL taking an example from [13].

```
DATABASE MODULE ResearchCompaniesModule;
  IMPORT Identifier, String;
  TYPE
    Agencies = (ESPRIT, DFG, NSF, ...);
    CompNames, EmpNames, ProjNames = String.Type;
    EmpIds = Identifier.Type;
    ProjIdRecType =
        RECORD projName : ProjNames; getsGrantFrom : Agencies END;
    ProjIdRelType = RELATION OF ProjIdRecType;
    CompRelType = RELATION compName OF
        RECORD compName : CompNames; engagedIn : ProjIdRelType END;
```

```
    EmpRelType = RELATION employee OF
        RECORD employee : EmpIds; empName : EmpNames;
            belongsTo : CompNames; worksOn : ProjIdRelType END;
    ProjRelType = RELATION projId OF
        RECORD projId : ProjIdRecType;
            consortium : RELATION OF CompNames END;
  VAR compRel : CompRelType;
      empRel : EmpRelType;
      projRel : ProjRelType;
  TRANSACTION hireEmployee (name:EmpNames;
        belongs:CompNames; works:ProjIdRelType) : EmpIds;
    VAR tEmpId : EmpIds;
    BEGIN
      IF SOME c IN compRel (c.compName = belongs) AND
        ALL w IN works (SOME p IN compRel[belongs].engagedIn (w = p))
      THEN tEmpId := Identifier.New;
        empRel :+ EmpRelType{{tEmpId,name,belongs,works}};
        RETURN tEmpId
      ELSE RETURN Identifier.Nil
      END
    END hireEmployee;
END ResearchCompaniesModule   □
```

3.2 The Formal Representation of Selected DBPL Constructs

Let us now represent language constructs of DBPL as above in B. We concentrate on those types, expressions and procedures that are essential in the database context.

Type Representation. In general a type is an algebra, hence consists of a fixed set of values and fixed operations on that set. Thus, a DBPL type corresponds to a basic set in a context plus additional functions that can also be represented in a context using constants and properties.

Let us first examine *record types* in DBPL that are heterogenous aggregates of the form

$$T \;=\; RECORD\, tag_1 : D_1; \ldots; tag_n : D_n\; END;$$

For concrete record types see Example 1.

Since $D_1 \ldots D_n$ are also types, we may assume that there are basic sets also denoted $D_1 \ldots D_n$ represented them. Then the underlying set of the *record type* T can simply be represented by the cartesian product

$$T \;=\; D_1 \times \cdots \times D_n\ . \tag{1}$$

The tags $tag_1 \ldots tag_n$ are used as designators for the components of the type T, hence give rise to functions $(i = 1, \ldots, n)$

$$tag_i \;\in\; T \rightarrow D_i \qquad \text{defined as} \tag{2}$$

$$tag_i \;=\; \lambda x \cdot \; (\; x \in T \wedge x = (d_1, \ldots, d_n) \;\mid\; d_i \;)\;. \tag{3}$$

Then (1), (2) and (3) in a context – more precicely in the `basic sets`, `constants` and `properties` sections respectively – define a B representation of the record type T.

Example 2. Let us take the type declaration

ProjIdRecType =
RECORD projName : ProjNames; getsGrantFrom : Agencies END;

from Example 1. Then a representation in a B context would be

`Basic Sets` $ProjIdRecType = ProjNames \times Agencies$
`Constants` $projName \in ProjIdRecType \rightarrow ProjNames$
$getsGrantFrom \in ProjIdRecType \rightarrow Agencies$
`Properties` $projName = \lambda x \cdot (\; x \in ProjIdRecType \wedge x = (y, z) \;\mid\; y \;)$
$getsGrantFrom = \lambda x \cdot (\; x \in ProjIdRecType \wedge x = (y, z) \;\mid\; z \;)$ □

There exists an alternative representation of record types using functions. We omit the details here [5]. Moreover, the tags are used in expressions and hence also in transactions, e.g. on the left or right hand side of an assignment such as $v.projName := \ldots$. In B this has to represented by applying the function *projName* to v, i.e. $projName(v)$. We also omit the details.

Let us now turn to the representation of *relation types* – for concrete relation types see Example 1 – such as

$$T = RELATION\ key_1\ ,\ \ldots,\ key_n\ OF\ D\ ;$$

Basically each element of this type is a finite set of elements of type D with unique values of the attributes $key_1, \ldots, key_n$. Assume that the type D has been introduced and named explicitly – the general case can be easily reduced to this. Then also functions $key_i : D \rightarrow D_i$ are defined $(i = 1, \ldots, n)$, and the set underlying T is representable as

$$T \;=\; \{x \mid x \in 2^D \wedge \forall d, e \in x \cdot \\ key_1(d) = key_1(e) \wedge \ldots \wedge key_n(d) = key_n(e) \;\Rightarrow\; d = e\} \tag{4}$$

If r is a variable of type T, then $r[k_1, \ldots, k_n]$ denotes in DBPL the selection of an element from r with key values $k_1, \ldots, k_n$. This selection function can be represented in a context as

$$sel_T \;\in\; T \times D_1 \times \cdots \times D_n \;\rightarrow\; D \qquad \text{defined as} \tag{5}$$

$$sel_T \;=\; \lambda x, y_1, \ldots, y_n \;\cdot\; (\; z \in x \wedge key_1(z) = y_1 \wedge \ldots \wedge key_n(z) = y_n \;\mid\; z \;)\;. \tag{6}$$

The DBPL expression $r[k_1, \ldots, k_n]$ then corresponds to $sel_T(r, k_1, \ldots, k_n)$. Then the representation of the relation type T comprises (4), (5), (6) and the representation of

operations $:+$, $:-$ and $:\&$ for the *insertion*, *deletion* and *update*. As an alternative, relations could also be represented by partial functions as done in [13]. For further details see [5].

Example 3. Take the type declaration

$$CompRelType = RELATION\ compName\ OF\ RECORD\ compName : CompNames;\ engagedIn : ProjIdRelType\ END;$$

from Example 1, which can be represented (in part) as follows in a context:

`Basic Sets`

$$CompRelType = \{x \mid x \in 2^{CompNames \times ProjIdRelType} \wedge \forall d, e \in x \cdot compName(d) = compName(e) \wedge engagedIn(d) = engagedIn(e) \Rightarrow d = e\}$$

`Constants`

$$compName \in CompNames \times ProjIdRelType \rightarrow CompNames$$
$$engagedIn \in CompNames \times ProjIdRelType \rightarrow ProjIdRelType$$
$$sel_{CompRelType} \in CompRelType \times CompNames \times ProjIdRelType \rightarrow CompNames \times ProjIdRelType$$

`Properties`

$$compName = \ldots$$
$$engagedIn = \ldots$$
$$sel_{CompRelType} = \lambda x, y, z \cdot (v \in x \wedge compName(v) = y \wedge engagedIn(v) = z \mid v)$$ □

The Representation of Access Expressions. In DBPL queries can be formulated through *access expressions* that are either *constructive* or *selective*. Constructive access expressions describe derivation rules that take the values of some given relations and produce another relation using the operations of relational algebra. Selective access expressions can only produce subsets of one given relation. The main difference is the possibility to update relations derived by selective expressions.

Let us concentrate on a selective access expression of the form

$$EACH\ e\ IN\ R : P\ ,$$

where e is a variable, R a relational expression and P a first-order formula with free variable e. In B this corresponds to a set expression of the form

$$\{x \mid x \in R \wedge P(x)\}\ . \tag{7}$$

Access expressions can be used in relational expressions of the form $T\{exp\}$, where T is a relation type and exp is an access expression. This can be represented by the identity function

$$id_T \in T \rightarrow T \qquad \text{with} \qquad id_T = \lambda x \cdot (x \in T \mid x) \tag{8}$$

applied to exp. The generalization of (7) and (8) to constructive access expressions is straightforward. For details see [5].

Example 4. Take the type *CompRelType* of Examples 1 and 3 and the relational expression

$$CompRelType\ \{\ EACH\ e\ IN\ compRel : e.compName = \text{“}MyCompany\text{”}\ \}\ .$$

This can be represented by

$$id_{CompRelType}(\{x \mid x \in compRel \wedge compName(x) = \text{“}MyCompany\text{”}\}\Box$$

Transaction Representation. Since transition specifications in abstract machines use parameterized guarded commands and transactions in DBPL are mostly written procedurally, there is in general no problem to formally represent transactions. Therefore, let us focus on insertions, deletions and updates on relations. We already mentioned these operations when we discussed the representation of relation types.

If r is a variable and R is a relational expression, both of relation type T, then $r :+ R$ denotes the insertion into r of all those elements of R that have key values not already in r. Analogously $r :- R$ and $r :\& R$ are used for deletions and updates respectively.

Insertion can be represented in B specifications by

$$r := id_T(r \cup \{x \mid x \in R \wedge \forall e \in r \cdot (key_1(e) \neq key_1(x) \vee \cdots \vee key_n(e) \neq key_n(x))\})\ .(9)$$

The representation of deletions and updates is analogous.

Example 5. Take the insertion operation

$$empRel :+ EmpRelType\{\{tEmpId,name,belongs,works\}\}$$

from Example 1 which is representable as

$$empRel := id_{EmpRelType}\ (\ empRel \cup \{x \mid x = (tEmpId,name,belongs,works)$$
$$\wedge\ \forall e \in empRel \cdot employee(e) \neq tEmpId\}\ )\ .\Box$$

3.3 A Characterization of Implementable Specifications

The representation of DBPL constructs in B specifications sets up a semantic equivalence between DBPL programs and specific B specifications. However, our original problem was just to find a transformation the other way round. Since this is not possible for every B specification, we have to characterize those specifications that are equivalent to DBPL programs. This gives us a proof obligation that indicates whether the refinement process has to be continued or not.

Before we give this characterization, we make the assumption that there is only one abstract machine in our specification and that all contexts have been collapsed

into one seen by the one and only machine. This assumption is due to the fact that we did not yet consider modularization.

Under these assumptions final specifications can be characterized by seven properties. The first three concern the data structures and are DBPL specific, whereas the last four are general conditions on operations [12].

Completeness. In the final context all basic sets and all constants are unambigously defined. In particular there are no more abstract sets.

Covering. In the final context all basic sets have the structure of a DBPL type and all operations corresponding to such a type are defined as constants.

Typing. For each state variable x of the final machine there exists a typing condition $x \in T$ in the invariant section such that T is a type defined in the final context.

Consistency. Each operation S in the final machine is consistent with respect to the invariant $\mathcal{I}$, i.e. $\mathcal{I} \Rightarrow wlp(S)(\mathcal{I})$ holds. The initialization S_0 satisfies $wp(S_0)(\mathcal{I}) \Leftrightarrow true$.

Totality. Each operation S in the final machine is total, i.e. $wp(S)(false) \Leftrightarrow false$ holds.

Termination. Provided the invariant holds each operation S in the final machine always terminates, i.e. $\mathcal{I} \Rightarrow wp(S)(true)$ holds. The initialization S_0 satisfies $wp(S_0)(true) \Leftrightarrow true$.

Determinism. Provided the invariant holds each operation S in the final machine is deterministic, i.e. $\mathcal{I} \wedge wlp(S)^*(\mathcal{R}) \Rightarrow wp(S)(\mathcal{R})$ holds for all well-formed formulae $\mathcal{R}$. For the initialization we must have $wlp(S_0)^*(\mathcal{R}) \Rightarrow wp(S_0)(\mathcal{R})$.

4 Database Program Generation as a Term Rewriting Process

Let us now address our main problem how to transform a final B specification into executable DBPL code. Since we neglected modularization aspects in Section 3, it should be clear that the result will be just one *module* in DBPL. Moreover, this should be a *database module*, i.e. all variables in it are considered to be persistent.

It has been exemplified in [4] that *order-sorted algebra* is useful for the specification of programming languages or even more general for a specification language such as B. Hence the idea to apply this algebraic approach to our problem, i.e. to represent B and DBPL in order-sorted algebra. Moreover, in the previous section we demonstrated that DBPL is representable in B. Thus, each DBPL program may be considered as a syntactic variant of a specific B specification. In order-sorted algebra this can be represented by using a unified algebra with the sorts from DBPL being subsorts of corresponding sorts from B and (conditional) equations that relate DBPL and B expressions. This is our approach here, where we use the OBJ system to accomplish the task.

A collection of (conditional) equations can also be regarded as a term rewriting system as OBJ does. Then the reduction of a complex term that represents a final B specification will result in the required DBPL code.

The rewrite rules that are presented in this section are simpler than the representation of DBPL in B suggests. This simplification is due to the fact that we assume them to be applied only to final specifications.

```
obj AM is protecting ID .
  ...
  sort set .
  subsort identifier enumset rangeset < set .
    op INT        : -> set .
    op POW        : set -> set .
    op _ /\ _     : set set -> set [assoc comm prec 30] .
  sort substitution .
    op (_ := _)   : lambda expression -> substitution [prec 43] .
    op (_ || _)   : substitution substitution
                    -> substitution [assoc comm prec 70] .
  sort command .
  subsort substitution < command .
    op SKIP       : -> command .
    op _ ; _      : command command -> command [assoc prec 70] .
    op _ [] _     : command command -> command [assoc comm prec 70] .
    op _ ==> _    : predicate command -> command [prec 60] .
    op VAR_IN_END : identifier-list command -> command .
    op DO_==>_OD  : predicate command -> command .
  ...
endo
```

Fig. 2. The OBJ module AM

4.1 An Overview on OBJ

OBJ is an algebraic specification language based on order-sorted algebra. The basic building block of an OBJ specification is the *object* or *module*. Each such object consists of a *signature* and *axioms*. In a signature *sorts* and *operators* are declared. Sorts are arranged in a *subsort hierarchy*. Each operator has an arity in $S^* \times S$, where S is the set of sorts. Figure 2 gives an example of an object called AM.

The axioms on an OBJ object are expressed as (conditional) equations with terms built from the constants and operators and an S-indexed family of variables. Associativity, commutativity, idempotency and neutral elements can be specified by specific keywords and need not be specified as axioms.

We may assign an order-sorted algebra A with a signature. Doing this we associate with each sort s a set A_s called the *carrier* of the sort such that subsorts correspond to subsets and the carriers are disjoint otherwise. Moreover, each operator f with arity $s_1 \cdots s_n \rightarrow s$ corresponds to some function $f_A : A_{s_1} \times \cdots \times A_{s_N} \rightarrow A_s$. If all (conditional) equations are satisfied when interpreted in the usual way in A, we may associate A with the specification.

In OBJ the semantics of an object is given by an initial order-sorted algebra. This algebra can be built as a quotient of the order-sorted term algebra constructed from the constants and the operators of the object with respect to the equivalence relation defined by the axioms. See [4] for more details.

OBJ supports modular specifications. An object may be built from others using three different modes called *proctecting*, *extending* and *uses*. The protecting mode

assures that the initial semantics of all old sorts and operators remains completely unchanged, i.e. no additional closed terms occur (*no junk*) and no terms are identified by new equations (*no confusion*). The extending mode only assures that there is no confusion, whereas *uses* assures nothing.

In addition to modularization OBJ also supports parameterization in a sophisticated way. However, this feature will not be used for our problem. Therefore, we dispense with describing it.

OBJ is supported by a software system that is built around a term rewriting engine. For term rewriting each (conditional) equation is interpreted as a rewrite rule in order to replace the left hand side by the right hand side. The OBJ rewrite engine supports AC and ACI rewriting. Rewriting of a term t is done by running $reduce(t)$ in the OBJ system.

4.2 Language Representation by Term Algebras

In order to apply OBJ to our problem we define four OBJ modules. The first one is called ID and contains general identifier and number symbols. We omit the details.

The module AM which protects ID contains the specification of the B syntax. It is illustrated in part in Figure 2. For a complete description see [5].

In AM we have sorts *enumset, rangeset* and *set* that are used for enumerated sets, for sets defined as subsets of others and for arbitrary (basic) sets. Clearly *enumset* and *rangeset* are subsorts of *set*. Thus, INT occurs as a constant of sort *set* and the intersection $\cap$ as an associative and commutative operator.

In order to represent guarded commands the sorts *substitution* and *command* are introduced with *substitution* being a subsort of *command* containing only the assignment commands.

The module DBPL which also protects ID contains the specification of the DBPL syntax. In this module we have among others the sort *Type* with some subtypes and the sort *Assignment*. *Type* contains the specifications of types. *Assignment* contains the specifications of specific operations on relations. The module DBPL is illustrated in part in Figure 3.

The last module is TRANS without new sorts. It only contains the sorts from AM (extended) and DBPL, some few new operations and the equations representing the equivalence between final B specifications and DBPL. The equations are illustrated in part in Figure 4 with primary focus on record and relation types, access expressions and transactions. Figure 5 illustrates the complete sort hierarchy in TRANS.

5 Conclusion

The work described in this paper is based on a three stage scenario for the development of database application systems.

- The first stage consists in building a conceptual design in some high-level design language. In our case this language is the dynamically enriched semantic data model TDL. A conceptual design should be automatically translatable into an initial formal specification.

```
obj DBPL is protecting ID .
  ...
  sort SimpleType .
  subsort Ident Enumeration SubrangeType < SimpleType .
  sort Type .
  subsort SimpleType < Type .
    op INTEGER    : -> Type .
  sort FieldList .
    op (_ :' _)   : IdentList Type -> FieldList [prec 60] .
    op (_ ;' _)   : FieldList FieldList -> FieldList [assoc prec 70] .
    op RECORD_END : FieldList -> Type .
  sort RelationKey .
  subsort KeyDesignator IdentList < RelationKey .
    op _ ,' _ : RelationKey RelationKey -> RelationKey [assoc prec 40] .
    op RELATION OF_ : Type -> Type .
    op RELATION _ OF_ : RelationKey Type -> Type .
  sort Assignment .
    op (_ :='_) : Designator Expression -> Assignment [prec 60].
    op (_ :+ _) : Designator Expression -> Assignment [prec 60].
    op (_ :- _) : Designator Expression -> Assignment [prec 60].
    op (_ :& _) : Designator Expression -> Assignment [prec 60].
  sort Statement .
  subsort Assignment < Statement .
  ...
endo
```

Fig. 3. The OBJ module DBPL

- The second stage comprises standard refinement rules that direct the completion and modification of the initial formal specification towards an implementation in a specific implementation language. In our case the formal method is B and the implementation language is the relational database programming language DBPL.
- The last stage consists in a characterization of sufficiently refined specifications and their automatic transformation into executable database programs.

While the first two stages were discussed in [13] we concentrate on the last stage. The characterization of implementable specifications could be achieved by representing DBPL language constructs in B which defines a semantic equivalence. The transformation was achieved by representing both the B and the DBPL language in a unified order-sorted algebra such that the semantic equivalences could be used as (conditional) rewrite rules. For this purpose we used OBJ.

Although we made some restrictive assumptions it could be shown that the automatic generation of DBPL programs is possible. However, two severe problems arise.

- B does not provide any notion of orthogonal persistence. Our approach here made everything persistent, which is possible, but not useful in practice. Therefore,

```
obj TRANS is extending AM . extending DBPL . extending TRUTH .
  var X Y Z : Ident . var E E1 E2 : Expression . var F F1 F2 : Designator .
  var K : KeyDesignator . var T : Type . var A A1 A2 : AccessExpressionList .
  ...
  op subdesignator : Designator -> KeyDesignator .
  eq subdesignator(X [E]) = [E] .
  eq subdesignator(F [E]) = (subdesignator(F) [E]) .
  eq subdesignator(X . Y) = Y .
  eq subdesignator(F . Y) = (subdesignator(F) . Y) .
  eq (POW(T)).set = (RELATION OF T).Type .
  eq { X | (X : RELATION OF T) & ALL X1 IN X (ALL X2 IN X ((F1 =' F2) <='
     (X1 =' X2))) } = { X | (X : RELATION subdesignator(F1) OF T) } .
  eq { X | (X : RELATION OF T) & ALL X1 IN X (ALL X2 IN X ((F1 =' F2) AND
     E <=' (X1 =' X2))) } = { X | (X : RELATION subdesignator(F1) OF T)
     & ALL X1 IN X (ALL X2 IN X (E <=' (X1 =' X2))) } .
  eq { X | (X : RELATION K OF T) & ALL X1 IN X (ALL X2 IN X ((F1 =' F2) AND
     E <=' (X1 =' X2))) } = { X | (X : RELATION K ,' subdesignator(F1) OF T)
     & ALL X1 IN X (ALL X2 IN X (E <=' (X1 =' X2))) } .
  eq { X | (X : RELATION K OF T) & ALL X1 IN X (ALL X2 IN X ((F1 =' F2) <='
     (X1 =' X2))) } = { X | (X : RELATION K ,' subdesignator(F1) OF T) } .
  eq ({ X | X : RELATION OF T }).set = (RELATION OF T).Type .
  eq ({ X | X : RELATION K OF T }).set = (RELATION K OF T).Type .
  eq RELATION subdesignator(X) OF T = RELATION OF T .
  ...
  eq SOME X IN E1 (E2 =' X) = E2 IN E1 .
  eq ALL  X IN E1 (X IN E2) = E1 <=' E2 .
  eq { X | (X =' E1) AND E2 & (Y : E).predicate } =
     { X | (X =' E1) AND E2 & (EACH Y IN E).ElementDenotation } .
  eq { X | (X =' E1) AND E2 & Q & (Y : E).predicate } =
     { X | (X =' E1) AND E2 & Q & (EACH Y IN E).ElementDenotation } .
  eq (B1 & B2).predicate = (B1 ,' B2).ElementDenotation-list .
  eq { X | (X =' E1) AND E2 & B } = { E1 OF B :' E2 } .
  eq { X OF EACH X IN E1 :' E2 } = { EACH X IN E1 :' E2 } .
  eq { A1 } \/ { A2 } = { (A1 ,' A2).AccessExpressionList } .
  eq (% X . (X : Z | X) { A }).lambda = (Z{ A }).Expression .
  ...
  eq (F := E).substitution = (F :=' E).Assignment .
  eq (F :=' (Z{ EACH Y  IN F :' NOT (Y IN E) }).Expression) = (F :- E) .
  eq (F :=' (Z{ EACH Y  IN F :' TRUE ,'
                EACH Y1 IN E :' NOT (Y1 IN F) }).Expression) = (F :+ E) .
  eq (F :=' (Z{ EACH Y1 IN F :' NOT (Y1 IN E) ,'
                EACH Y2 IN E :' (Y2 IN F) }).Expression) = (F :& E) .
  ...
endo
```

Fig. 4. The OBJ module TRANS

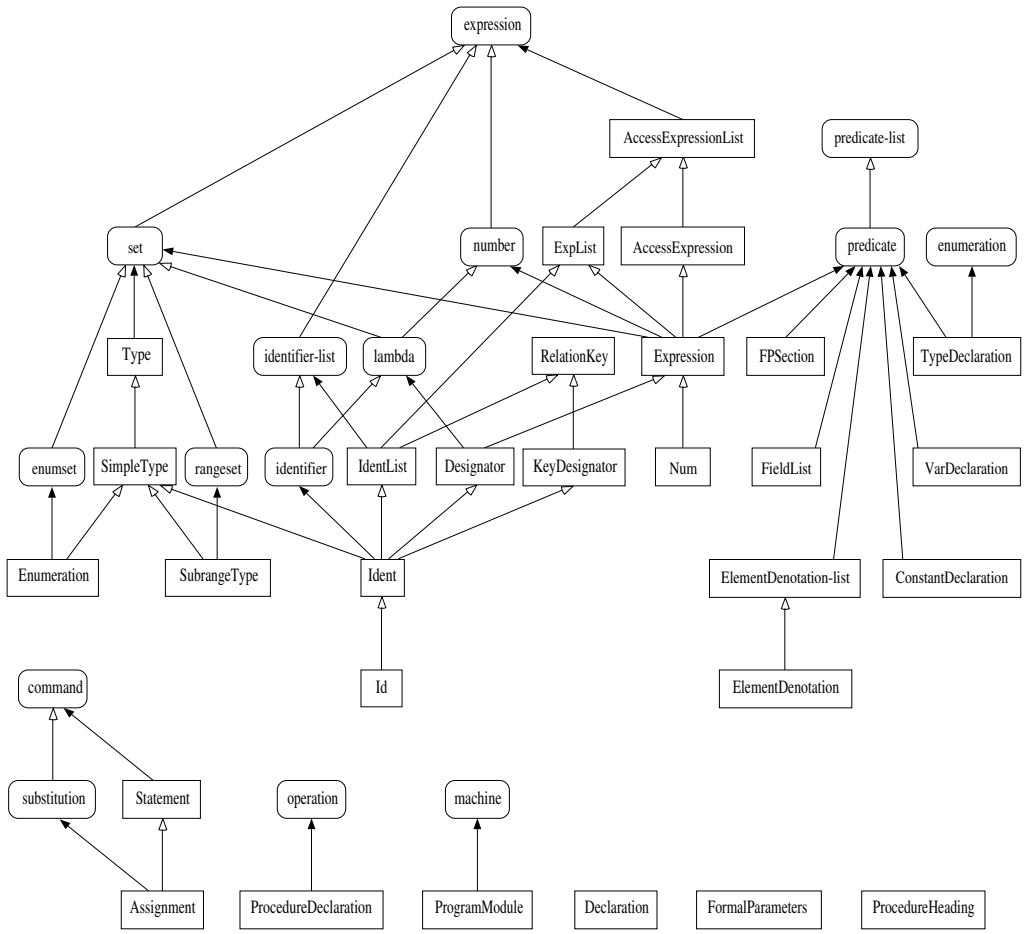

Fig. 5. The sort hierarchy in the OBJ module TRANS

formal methods that are suitable for database application systems should at least offer the user to declare parts of the state description being persistent.

– While the operations in final specifications could be nicely characterized by abstract logical formulae involving predicate transformers, the characterization of final data structures is cumbersome. The reason for this is that DBPL is strongly typed, whereas B is untyped. Further investigations in formal methods should solve this mismatch by introducing typed specifications as proposed in [12]. Otherwise we risc to specify on a lower level of abstraction than we implement which would make formal methods inacceptable.

Despite these outstanding problems it has been made apparent that formal methods combined with conceptual design as a "front-end" and sophisticated implementation

as a "back-end" enhances the benefits of formal methods, eases their use and at the same time fills the gap between the conceptual design and the implementation on a safe mathematical basis.

References

1. J. R. Abrial: *A Formal Approach to Large Software Construction*, in J.L.A. van de Snepscheut (Ed.): *Mathematics of Program Construction*, Proc. Int. Conf. Groningen, The Netherlands, June 89, Springer LNCS 375, 1989
2. A. Borgida, J. Mylopoulos, J. W. Schmidt, I. Wetzel: *Support for Data-Intensive Applications: Conceptual Design and Software Development*, Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon, June 1989
3. E. W. Dijkstra, C. S. Scholten: *Predicate Calculus and Program Semantics*, Springer-Verlag, 1989
4. J. A. Goguen, T. Winkler: *Introducing OBJ3*, SRI International, Technical Report, August 1988
5. T. Günther: *Charakterisierung und Transformation in DBPL implementierbarer Abstrakter Maschinen* (in German), Master Thesis, University of Hamburg, August 1992
6. R. Hull, R. King: *Semantic Database Modeling: Survey, Applications and Research Issues*, ACM Computing Surveys, vol. 19(3), September 1987
7. M. Jarke, J. Mylopoulos, J. W. Schmidt. Y. Vassiliou: *DAIDA: An Environment for Evolving Information Systems*, ACM ToIS, vol. 10 (1), January 1992, pp. 1 – 50
8. C. B. Jones: *Systematic Software Development using VDM*, Prentice-Hall International, London 1986
9. F. Matthes, J. W. Schmidt: *DBPL Rationale and Report*, FIDE technical report, 1992
10. J. Mylopoulos, P. A. Bernstein, H. K. T. Wong: *A Language Facility for Designing Interactive Database-Intensive Applications*, ACM ToDS, vol. 5 (2), April 1980, pp. 185 – 207
11. G. Nelson: *A Generalization of Dijkstra's Calculus*, ACM TOPLAS, vol. 11 (4), October 1989, pp. 517 – 561
12. K.-D. Schewe, J. W. Schmidt, I. Wetzel, N. Bidoit, D. Castelli, C. Meghini: *Abstract Machines Revisited*, FIDE technical report 1991/11
13. K.-D. Schewe, J. W. Schmidt, I. Wetzel: *Specification and Refinement in an Integrated Database Application Environment*, in S. Prehn, H. Toetenel (Eds.): Proc. VDM 91, Noordwijkerhout, October 1991, Springer LNCS
14. J. M. Spivey: *Understanding Z, A Specification language and its Formal Semantics*, Cambridge University Press, 1988

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