Structuring Database Specifications

Martin Wirsing
Universität Passau
Bayerisches Forschungszentrum für Wissenschaftlichre Systeme
Postfach 25 40
W-8390 Passau

Jack Leszczyński
Institute of Computer Science
Polish Academy of Sciences
00-901 Warszawa PKIN
P.O. Box 22
Poland

Abstract
The construction of specifications, their decomposition into separate modules and the reusability of such modules are key problems for the development of high-quality software. The paper discusses these issues for a small but realistic case-study: the development of a database specification within the framework of an (algebraic) specification language, which allows parametrization and polymorphism. Two database specifications are developed which have a different module structure. The development process of the two specifications is described and analysed.

1. Introduction

Today requirement and design specifications are constructed using informal descriptions. However, for the development of high-quality software informal methods are not enough; they have to be integrated with formal methods in order to provide means for testing, analysing and transforming specifications.

The algebraic/axiomatic approach is a set of the principal candidates for industrializing formal specifications and the development of software. It has influenced the design of procedural languages such as ADA and object oriented programming languages such as Eiffel. Other formal approaches are set- or model-oriented (such as Z [Spivey 89] and VDM [Jones, Shaw 90]) or logic-based [Nordström et al. 90].

The basic idea of the algebraic approach consists in describing data structures by just giving the names of the different sets of data, the names of the basic functions and their characteristic properties (as a set of equational first-order formulas). For the description of large data structures it is necessary to compose specifications from smaller ones. This has led to the introduction of specification languages such as CLEAR, LARCH, OBJ, ACT and ASL (for references see [Wirsing 90]). Having such languages available, one of the difficult tasks during the design of formal specifications is to find the right degree of modularity. In many cases it is easy to write a flat specification having only a few building blocks; but afterwards it becomes difficult to understand and use this specification since the axioms tend to be of too low level.

In order to make formal specifications human readable it is necessary to abstract from low level properties and to split the specification in components whose behaviour can be understood in isolation by looking at the interface to the other components. Having the right degree of modularity, components can be developed independently and modules from a library of components can be used such that development time and costs decrease and the reliability of software increases (cf. e.g. [Wirsing 88]). This paper presents a case study in the modular structuring of a previously "flat" specification. It treats the same example, a database specification, as the excellent paper by C. Jones and C.S. Fitzgerald [Jones, Fitzgerald 90] (cf. also [Walshe 90]). Instead of VDM, we use the algebraic specification framework. Technically we start from an extension of ASL ([Sannella, Wirsing 82], [Wirsing 86], [Sannella, Tarlecki 88]) that allows to integrate three major structuring mechanisms: polymorphism, parameterisation and "high-level" predicates (cf. [Leszczyński, Wirsing 91]).

The paper is built around database examples. We start with a binary relational database in which a "flat" specification is given. In a second step, two modular and more general database specifications are constructed; then by specialising particular components, new (equivalent) specifications of the original database are derived.

The paper is organized as follows: In section 2 an informal specification of the binary database is presented. In section 3 the specification language is introduced by means of a simple example. In section 4 a flat specification of the database is given. In section 5 a first modular specification of an n-ary relational database is developed. In section 6 polymorphism is used to control the typing of the database.

2. Informal specification of a binary database

In this section we give the informal specification of our main example, the relational database system "NDB". "NDB" stood for "Norman's Database" and was described in [Wirth 71]; it eventually became an IBM product, named the "Non-programmer Database" [IBM]. It is based on the binary relational model, i.e. the database consists of units of information called "entities" together with binary relations which describe the associations between entities. The advantage of the relational model (in contrast to other database models such as the hierarchical model or the network model) is that it is based on the mathematically well-defined notion of a relation; the elements of a binary relation are pairs of entities and therefore a relation can be understood as a set of pairs of entities.

Informal specification of a binary database

The database deals with "entities" as basic objects about which information is stored. Each entity is represented by a unique "entity identifier" (Eid) which is associated with a certain value. Each binary relation is identified by a name (Rid); its components can be addressed by attributes (Attr). Moreover, each attribute is associated with a "typing constraint", i.e. the set of those entities which are allowed to occur as values of the attribute. Each relation can be restricted to satisfy a "database dependency": one-to-one (1), one-to-many (1M), etc.

The database consists of the description of the entities, the typing information and the database relations. A state of the database is consistent if the above mentioned properties are satisfied and, moreover, each entity occurs in one of the "typing constraints". The characteristic operations over a database are:

- entity addition and deletion
- database relation addition and deletion
- tuple addition and deletion
- construction of the "empty" database.

Note that due to space limitations we do not consider an important part of the database system, the query language.

3. Polymorphic typed specifications

The specification language we use in the paper is an extension of ASL [Wirsing 86] by polymorphism and typing of specifications following the ideas of [Leszczyński, Wirsing 91]. Let us recall briefly some of its important features. The main ideas are constructed in two steps: First, the signature is given (i.e. the sorts and names of the characteristic functions and predicates); second, the characteristic properties of the elements of the signature are defined by axioms. Structured specifications are constructed using three "specification operators": extend sp by Δ (extension of a specification sp by a set Δ of new declarations and axioms).

CH3001-5/91/0000/0406/$01.00 © 1991 IEEE
sp * sp ("putting two specifications together").
sp by r ("renaming of sp by a renaming of r of symbols"). In our extension of
ASL, signatures consist of a set of declarations and equations
between sorts. Each specification sp has a type Spec(Σ) where Σ
denotes the signature of sp. Because of this type information,
declarations are not explicitly given in the body of the
specification. Moreover, auxiliary symbols used in the body are
automatically hidden if they are not mentioned in the signature.
Thus the type of sp gives an exact description of the export-
interface and therefore replaces the export-operator of ASL.
Let us consider the example of finite sets. There is a predefined
global specification SET0. It contains a "sort operator" Set that for
any given sort D returns a new sort Set(D) which describes all
finite subsets of D. Moreover, there are basic set operations such as
∅ ("empty set"). ∪ ("union"), ∩ ("intersection"), { } ("singleton set"), \ ("deletion") and one predicate . ∈ ("is-
element to our language.

m+

4. A flat specification of the database

Following the informal specification, the signature of the NDB
database consists of eight sorts: Eid ("Entity identifiers"). Value,
5. A first modularisation

It is obvious that the notion of relation used in the definition of NDB can be isolated and specified separately. In this case it seems natural to generalise pairs of entities and the consistency of relations cannot be specified; this requires more information about the actual structure of RInf.

5.1. Tuples

We define tuples as finite mappings from attributes to entities with three characteristic functions for creating a tuple ("create"), retrieving the value of an attribute ("value") and for determining "okdep(as,d,s)" which holds if the set of tuples s satisfies the dependency d and is consistent with the attributes as (of s).

val DEP_SIG : Signature = extend DEP_SIG by Dependency by Set(Set(Attr)) x Attr end

val REL_DER : Spec(DEP_SIG) = extend RasFun by Dependency by Set(Set(Attr)) x Attr

We define two different specifications for dependencies: REL_DEP1 describes the dependencies of NDB, REL_DEP2 describes the more general case where a set of functional dependencies is associated with each relation.

5.2. Functional dependencies

The polymorphic predicates FUN and RFUN are generalized to a predicate RasFun(s, [a1, ..., an], a) which holds if a1, ..., an → a is a functional dependency of the set of tuples s.

val RasFUN_SIG : Signature = extend TUPLE by

This specification uses a predefined specification of finite sequences with a sort operator seq. If s is a sequence, then s_i denotes the i-th component of s; by elems(s) we denote the set of all elements of s.

We define two different specifications for dependencies: REL_DEP1 describes the dependencies of NDB, REL_DEP2 describes the more general case where a set of functional dependencies is associated with each relation.

val REL_DER : Spec(DEP_SIG) = extend RasFun by Dependency by Set(Set(Attr)) x Attr

5.3. Typing of attributes

For the typing of attributes we also define a separate specification where the range of each attribute is not global as in NDB but local to each relation. Thus each typing consists of a sequence of attributes and a "setmap".

val TYP_SIG : Signature = extend TUPLE by

As a consequence, we can develop a specification of dependencies by extending RasFUN by a sort "Dependency" and a predicate "okdep(as,d,s)" which holds if the set of tuples s satisfies the dependency d and is consistent with the attributes as (of s).

val DEP_SIG : Signature = extend RasFUN_SIG by Dependency by Set(Attr)

5.4. A relation module

The specification of relations is given in two steps. First, a specification RBASE is defined that abstracts from the particular structure of the typing and dependency information and introduces a sort "RInf" instead. The signature of RBASE also provides function symbols for adding and deleting tuples and entities from relations. At this stage the handling of entities and the consistency of relations cannot be specified; this requires more information about the actual structure of RInf.

val RBASE_SIG : Signature = extend TUPLE by
RInf, Relation : SORT

new : RInf → Relation

in_tup, out_tup : Relation*Tuple → Relation

put_ent : Set(Attr) * Eid* → Relation = Relation → Relation

take_ent : Eid * Relation → Relation

okrel : Pred( Relation)

end

val RBASE : Spec(RBASE_SIG) =

extend TUPLE by

Vi: RInf, s: Set(Tuple), t: Tuple.

{okrel(i, s) =

new(i) = (i, 0)

in_tup((i,s), t) = okrel(i, s U t)

out_tup((i,s), t) = okrel(i, s) - t}

end

In the second step, we define RInf as Cartesian product of Typing and Dependency. The consistency predicate "okrel" of a relation is given as the conjunction of the consistency predicates of TYP_SIG and DEP_SIG. The function symbols put_ent and take_ent can now be defined by the corresponding functions of SETMAF'.

By adding the two particular dependency specifications we obtain two variants of the relation module: RELATION1 admits only binary relations, RELATION2 admits relations of arbitrary size.

val REL1_SIG : Signature = TD_REL_SIG + DEP1_SIG

val RELATION1 : Spec(REL1_SIG) = TD_REL + REL_DEP1

val REL2_SIG : Signature = TD_REL_SIG + DEP2_SIG

val RELATION2 : Spec(REL2_SIG) = TD_REL + REL_DEP2

5.5. A database module

The database module extends RBASE by the sorts Value, Eid, Db

and the characteristic function symbols of NDB extended to arbitrary tuples. The consistency predicate "ok" is not yet completely specified; it needs additional information which is provided by the auxiliary predicate "okdom".

val DB1_SIG : Signature = DB1_SIG U REL1_SIG

val DBASE1 : Spec(DB1_SIG) = DB1 + RELATION1 + OKDOM

Specialisation with RELATION2 yields specification DBASE3 of a database with arbitrary n-ary relations.

val DB3_SIG : Signature = DB1_SIG U REL2_SIG

val DBASE3 : Spec(DB3_SIG) = DB1 + RELATION2 + OKDOM

where OKDOM denotes the specification

spec ∀V: EntMap, Rs: DbRels.

okdom(V, Rs) =

dom(V) =

{e|Eid} |n:Rd, as:Seq(Attr), tm:Attr → Set(Eid), a:Attr.

Rs(n) = ((as,tm), phi) A a ∈ elms(as) A e ∈ tm(a)}

end

Since the typing of attributes is local to each relation the states of Db are pairs consisting of an entity mapping Sort "Entmap" and a relation mapping (sort "DbRels").
6. A polymorphic modularisation

In this section we construct a second modularization of the database specification which uses the polymorphism of the language.

6.1. Tuples

Since tuples have domains of arbitrary sort a polymorphic specification is more appropriate than the one of section 5.1. Instead of a sort Tup we define a sort operator Tup whose argument describes exactly the domain of a tuple. The restriction to finite mappings and the operation "dom" are not necessary anymore.

```
val P_TUP_SIG : Signature =
  sig
    Eid : SORT
    Tup : SORT -> SORT
    value : poly A : SORT. Tup(A) x A -> Eid
  end

val P_TUP : Spec(P_TUP_SIG) =
  spec
    VA : SORT, t : A -> Eid, s : A.
    value(create(t), a) = (a(t))
  end
```

Similarly, RasFun can be defined as polymorphic predicate. The specification P_RasFun can be obtained from RasFun by introducing an additional quantification "VA: SORT" and by substituting Tup(A) for Tuple and A for Attr.

```
val P_RasFUN_SIG : Signature =
  extend P_TUP_SIG by
    RasFun : poly A : SORT. Pred(Set(Tup(A)) x Set(A) x A)
  end

val P_RasFUN : Spec(P_RasFUN_SIG) =
  extend P_TUP by
    VA : SORT, s : Set(Tup(A)), D : Set(A), a : A.
    RasFun(d, a) =
      [\forall u1, u2 : Tup(A). u1 \in s \land u2 \in s ->
       \forall v : Attr. b : D = value(u1, b) = value(u2, b)]
      \land
      value(u1, a) = value(u2, a)
  end
```

6.2. A relation module

The specifications of the dependency and typing information can be easily transformed to polymorphic specifications. The typing information which was recorded in the first arguments of the consistency predicates oktyp and okdep is shifted to the corresponding monomorphic ones and can be combined to a polymorphic specification P_RINF of the relation information.

```
val P_RINF_SIG : Signature =
  extend P_RasFUN_SIG \ SETMAP by
    okinf : poly A : SORT.
    Pred((A -> Set(Attr)) \ Set(Set(A) x Attr) \ Set(A))
  end

val P_RINF : Spec(P_RINF_SIG) =
  extend P_RasFUN \ SETMAP by
    VA : SORT.
    RInf(A) = (A -> Set(Attr)) \ Set(Set(A) x Attr) \ Set(A)
    \forall m : Attr. RInf(A) \ x \ Set(Attr) \ x Set(A)
  end
```

In order to define the sort relation in the relation module we have to combine the different tuple types. This can be done by using a dependent sum construct:

```
[s] 
```
val DBM : | Rel:Spec(RBASE_SIG).Spec(DB1_SIG) =
DB1[Rel/RBASE]

Note that the dependent product operator \( \Pi \) must be used because DB1_SIG depends on RBASE_SIG. Then DB1 to DBASE4 can be written in terms of DBM:

\[
\begin{align*}
DB1 &= DBM(RBASE) \\
DBASE2 &= DBM(RBASE) + RELATION1 + OKDOM \\
DBASE3 &= DBM(RBASE) + RELATION2 + OKDOM \\
DBASE4 &= DBM(RBASE1) + RELATION3 + OKDOM1
\end{align*}
\]

7. Conclusion

The shape of the right hand sides of the above equations suggests a further parameterisation, but actually (apart from OKDOM and OKDOM1) this cannot be accomplished even via parameter passing morphisms (cf. [Wirsing 90]) due to the different structure of e.g., RELATION1 and RELATION3. The fact that RELATION3 is polymorphic, whereas RELATION1 is not, only helps to make the position worse. However, this is not surprising at all since the RELATION specifications contain different implementation decisions. The example proves the usefulness of structured specification languages during the development process. Its modularity supported the design of several (equivalent) database specifications. Moreover, the transition from common specifications to polymorphic ones (with dependent types) seems to be a very elegant implementation tool and should therefore be subject of further investigations.

References

[Fitzgerald, Jones 90]

[IBM]
IBM: Data mapping program: user's guide. SB 11 - 5340.
[Jones, Shaw 90]

[Leszczylowski, Wirsing 91]

[Nordström et al. 90]

[Sannella, Tarlecki 88]

[Sannella, Wirsing 83]

[Spivey 89]

[Walshe 90]
A. Walshe: NDB: The formal specification and rigorous design of a single user database system. In: [Jones, Shaw 90].

[Winterbottom, Sharmar 79]

[Wirsing 86]

[Wirsing 88]