Towards a Conceptual Model for the Environment of the Algebraic Specification Language ACT ONE*

Ingo Claßen
TU Berlin
Informatik, FG Formale Spezifikation
Franklinstraße 28/29
W-1000 Berlin 10
E-Mail: ingo@opal.cs.tu-berlin.de

Martin Gogolla
TU Braunschweig
Informatik, Abt. Datenbanken
Gaußstraße 12
W-3300 Braunschweig
E-Mail: gogolla@infbs.uucp

TU Berlin, Forschungsberichte Informatik, Nr. 91/11

Abstract

Due to the interactive nature of the specification task, it is desirable and necessary for a specification environment to support a collection of pre-defined or already verified modules on secondary storage. This possibility also enhances the modularity features of specification languages.

The present paper defines a conceptual model for the algebraic specification language ACT ONE and its accompanying specification environment. But the approach chosen is quite general and can be employed for other specification languages as well. The definition of the database schema (including integrity constraints) is done by means of an expressive and semantically well-founded extended entity-relationship model and a respective calculus.

The basic idea is to start with the context free grammar of ACT ONE and to introduce entity types for non-terminal symbols. Productions are modelled by a type construction (a peculiarity of the extended entity-relationship model) and appropriate components (entity-valued attributes). Context sensitive conditions are translated into integrity constraints formulated in the calculus. This approach especially allows to describe various degrees of incompleteness of ACT ONE types, for instance whether an ACT ONE type has already been checked syntactically or whether the flat equivalent of the type is available.

1 Introduction

The development of algebraic specifications [EM85, EGL89, EM90] describing software systems of practical relevance usually results in large sets of related specification units. These units arise from the decomposition of complex specifications into smaller pieces by means of the structuring mechanisms provided by specification languages. Additionally algebraic methods and especially

*This work has been partially supported by the ESPRIT Basic Research Working Group No. 3264, COMPASS

specification languages give rise to a bulk of information like proofs, formal transformation steps, formal relations like signature morphisms, etc., that must be stored to be accessible by various tools.

Existing specification systems like RAP [Hus85], OBJ3 [GW88], ASSPEGIQUE [BC85, Cho88], ACT [Han87] or OBSCURE [Loe89] provide some mechanisms to store and retrieve specifications and to operate on them. But in general (the OBSCURE system seems to be an exception) they employ no systematic approach to the administration of their information. Usually they rely on storage facilities of the underlying programming language and file system.

A solution to the information handling problem is the use of databases [ULL88, ULL89]. They are already accepted as being important components of software development systems [Got88], and, since specification systems can be regarded as parts of general software development systems, the same arguments apply to them.

The application of databases to construct systems for specification languages leads to certain requirements:

- Since specifications are structured entities, the database must be capable to deal with complex objects in a coherent way.
- Since the specification task has an interactive nature, the database must be capable to deal with incomplete information. It must support different degrees of incompleteness and should enable trigger mechanisms for automatic tool invocations if the state of completeness changes.

Using database methodology to describe the information items in a specification system has the advantage of decoupling the logical description of the information elements from respective actions like tool invocations. Usually the logical description of the information is given by a database schema in a suitable data model. Integrity constraints in a suitable calculus may be added to the schema to describe consistent states of the database.

The aim of this paper is to apply results of the database methodology in the area of algebraic specifications. We show how a concrete data model, namely the entity relationship model and calculus of [HG88, GH91], can be used to support the algebraic specification language ACT ONE [EFH83, EM85, Cla88, Cla89]. The concepts used in this paper are, of course, general enough to support other specification languages as well. Therefore, we feel the design of a conceptual schema for ACT ONE is mainly a case study in employing a semantic data model for database support of specification or programming languages.

Section 2 provides a short and informal description of ACT ONE. It presents the formal syntax of the language that serves as the basis for the database schema in section 4. Section 3 introduces the data model and gives a brief description of its important features. Section 4 presents our case study in which the example specification language ACT ONE is modeled by a database schema that provides information to support several tools and that deals with incomplete specifications.

2 The Algebraic Specification Language ACT ONE

The algebraic specification language ACT ONE was developed at the Technical University of Berlin [EFH83], implemented in a first version using equational axioms only and in a revised version using conditional equations and initiality constraints [Cla88, Cla89]. ACT ONE is based on the concept of parameterized specifications (see [EM85]) which includes usual algebraic specifications with initial and loose semantics as special cases. The structuring mechanisms to build up larger specifications from smaller pieces are *extension, union, renaming,* and *actualization.*

The following is an example of a specification of generic sets:

```
type SET is
  extend DATA by
    sorts Set(Data)
    opns  empty: -> Set(Data)
          single: Data -> Set(Data)
          _U_: Set(Data), Set(Data) -> Set(Data)
    eqns  vars s,t,v: Set(Data) ofsort Set(Data):
            s U empty = s;
            s U s = s;
            s U t = t U s;
            (s U t) U v = s U (v U t);
  endext
endtype
```

Type *SET* uses the structuring mechanism extension to import type *DATA* that provides sort *Data*. Note that body sorts (i.e., non-parameter sorts) like *Set(Data)* may have structured names to indicate sort dependencies. Such dependencies are used in actualizations to control origins and implicit renamings (see [Cla88] for details).

```
type DATA is
  extend BOOL initial BOOL by
    parameter
      sorts Data
      opns  _eq_: Data, Data -> Bool
      eqns  ...
  endext
endtype
```

The semantics of type *SET* is roughly a mapping that assigns to each carrier set of sort *Data* the set of all finite subsets of this carrier set. Since parameter parts of parameterized specifications are interpreted loosely, i.e., each *DATA* algebra is considered, an initial constraint (see [EM90] for details) on *BOOL* (in *DATA*) is introduced that provides us with the intended interpretation of the boolean values.

Actualization is the structuring mechanism of ACT ONE that allows us to generate instances of generic (i.e., parameterized) specifications. The following actualization creates, for example, sets of characters.

```
type CHARSET is
  actualize SET by CHAR using
    sortnames Char for Data
    opnames   eq for eq: Data, Data -> Bool
endtype
```

The parameter sort *Data* of *SET* is mapped on sort *Char* (resulting in an implicit renaming of sort *Set(Data)* into *Set(Char)*) and the parameter equality operation is mapped on the equality of characters.

Since the syntax of ACT ONE evolves over the time, we provide the syntax of the language (in BNF-notation) that is used in the database schema in section 4. The following conventions are used:

terminals	underlined
nonterminals	not underlined
::=	definition
|	alternative
[*syntactical entity*]	zero or one occurrences of *syntactical entity*
{*syntactical entity*}$^+$	one or more occurrences of *syntactical entity*
{*syntactical entity*}*	zero or more occurrences of *syntactical entity*

(1)	spectext	::=	<u>acttext</u> {type}$^+$ [<u>fromlibrary</u> typenamelist] <u>endtext</u>
(2)	type	::=	<u>type</u> id <u>is</u> pexpr <u>endtype</u>
(3)	pexpr	::=	pspec | construct
(4)	construct	::=	union | extension | renaming | actualization
(5)	union	::=	<u>union</u> typeexpr {<u>,</u> typeexpr}$^+$ <u>endunion</u>
(6)	extension	::=	<u>extend</u> typeexpr [<u>initial</u> typenamelist] <u>by</u> pspec <u>endext</u>
(7)	renaming	::=	<u>rename</u> typeexpr <u>using</u> repl <u>endren</u>
(8)	actualization	::=	<u>actualize</u> typeexpr <u>by</u> typeexpr [<u>using</u> repl] <u>endact</u>
(9)	typeexpr	::=	id | construct
(10)	pspec	::=	<u>parameter</u> spec [<u>body</u> spec] | [<u>body</u>] spec
(11)	spec	::=	[<u>sorts</u> sortname {<u>,</u> sortname}*] [<u>opns</u> {operation <u>;</u>}$^+$]

```
                                [eqns {equationblock}+]
(12)  repl            ::=  [sortnames {sortname for sortname}+]
                           [opnames {id for id[: [argumentlist] -> sortname]}+]
(13)  operation       ::=  opname {, opname}* : [argumentlist] -> sortname
(14)  opname          ::=  id | _ id _ | if_then_else_fi
(15)  argumentlist    ::=  sortname {, sortname}*
(16)  equationblock   ::=  [vars vardecllist] ofsort sortname : {condeqn}+
(17)  condeqn         ::=  [ equation {, equation}* ->] equation
(18)  equation        ::=  term = term
(19)  vardecllist     ::=  {id {, id}* : sortname;}+
(20)  sortname        ::=  id [ ( sortname {, sortname}* )]
(21)  term            ::=  sterm [: sortname]
(22)  sterm           ::=  id [( term {, term }* )] | ( sterm )
                           | if term then term else term fi | term id term
```

The concrete syntax of identifiers is not important in the following.

3 The Extended Entity-Relationship Model and Calculus

This section presents a brief informal description of our approach to data modelling by an example of a simplified geoscientific application. We want to model information about different kinds of bodies of water, about countries, towns, and the leading people of countries and towns. Therefore, we would like to represent the following facts:

- Every TOWN lies in a COUNTRY and may lie at one or several RIVERS.
- RIVERS flow through COUNTRIES and flow into some kind of WATERS.
- SEAS, RIVERS, and LAKES are such WATERS.
- A PERSON may be the mayor of a TOWN, or the boss or a minister of a COUNTRY.

Besides usual properties of entities, like the name of a town, the population of a country, or the list of addresses of a person, we additionally associate with every geographic object a geometry in the world's co-ordinate system:

- Countries are represented by a set of closed polygons (representing their regions),
- towns and lakes by circles,
- seas by closed polygons, and

- rivers by connected, non-overlapping lines.

We describe this situation with an extended entity-relationship model and give a corresponding entity-relationship diagram in figure 1.

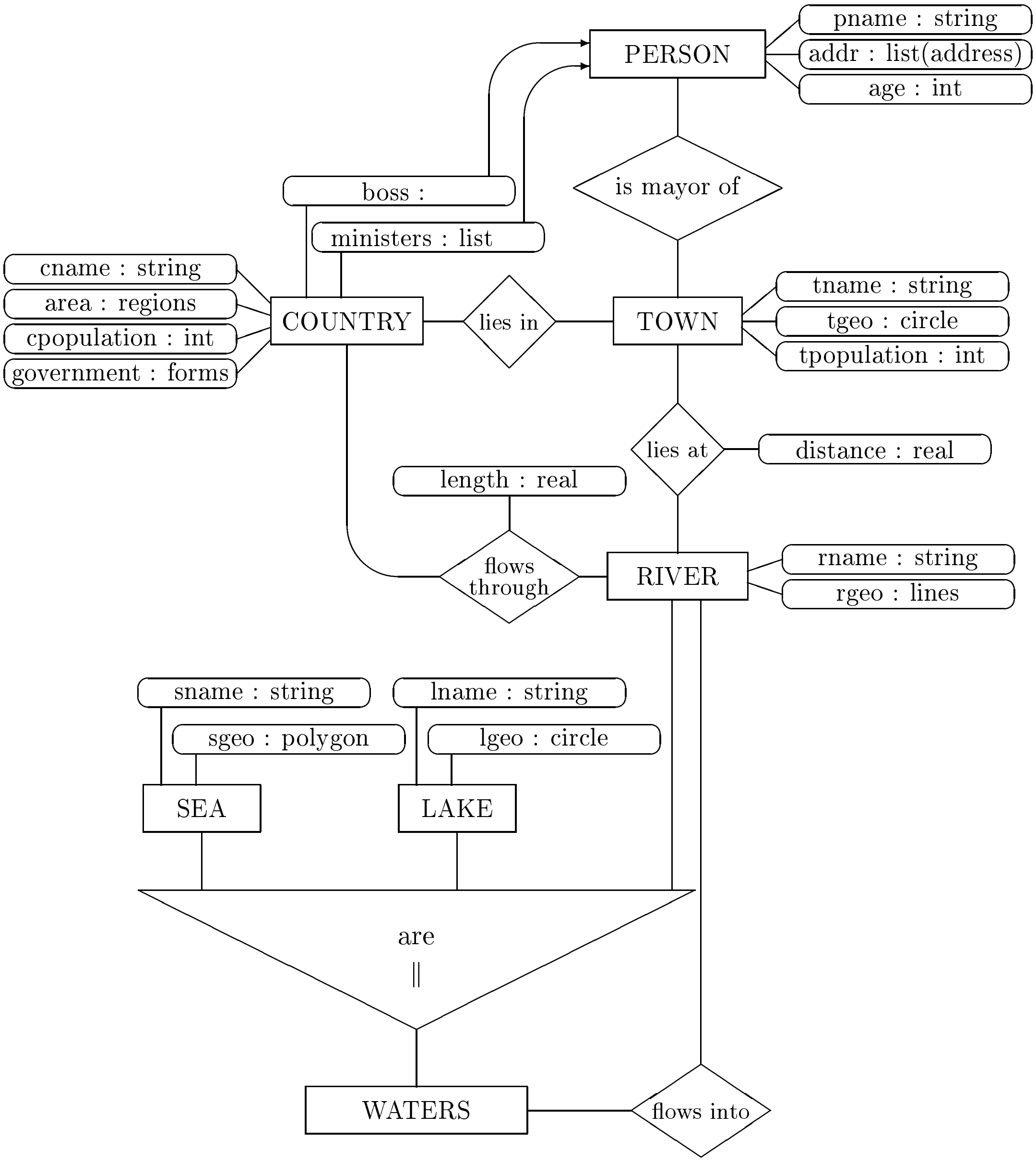


Figure 1: Diagram of the example ER schema

Here, we shortly recapture the fundamental notions of the underlying EER model and calculus. We only present a sketch of the syntax and give examples, because all technical details and the formal semantics can be found in [HG88, GH91]. There, also full motivation for all concepts can be found.

3.1 The EER Model

Here, we present the basic ingredients of our data model. The model allows data types like *int* or *string*, parameterized data types like *list* or *set*, and aggregation functions (for instance to count the number of elements in a set). The central notion here is the extended entity-relationship schema providing names for entity and relationship types, attributes, components, and type constructions.

3.1.1 Data Types

Our EER model does not only allow standard predefined data types like *int*, *real*, or *string*, but also arbitrary application dependent types. For instance, it is possible to define and use geometrical types like *point*, *circle*, or *lines* together with appropriate operations and predicates. Please note, these data types are used for modelling purposes only and are distinct from ACT ONE data types.

Formally, we have a data signature DS consisting of

- finite sets DATA, OPNS, PRED giving the names of the data sorts, operation and predicate symbols, respectively, together with
- functions $\mathit{source} : \text{OPNS} \rightarrow \text{DATA}^*$, $\mathit{destination} : \text{OPNS} \rightarrow \text{DATA}$, and $\mathit{arguments} : \text{PREDS} \rightarrow \text{DATA}^+$ giving the source and destination types of function symbols and argument types of predicate symbols.

Roughly speaking, data signatures are interpreted by sets, (partial) functions, and predicates.

3.1.2 Parametrized Data Types

For modelling purposes flat data types do not suffice. Certain standard constructors are useful in the context of data types (and also in the context of entity types as we will see later). Therefore, we allow to use tuple-, set-, list-, and bag-valued parameterized types to be applied to already defined types; bags (or multi-sets) allow that one element occurs more than once in the collection of elements. Again, these types are characterized by (the names of) their sorts, operations, and predicates. Given already defined types s, s_1, ..., s_n (or predefined types like *nat*, *int* or *string*) we introduce derived sorts

$$\mathit{list}(\mathrm{s}),\ \mathit{bag}(\mathrm{s}),\ \mathit{set}(\mathrm{s}),\ \text{and}\ \mathit{tuple}(\mathrm{s}_1, ..., \mathrm{s}_n).$$

These constructors are used to define application dependent types, e.g.,

$$\begin{aligned} \mathit{point} &= \mathit{tuple}(\mathit{real},\mathit{real}) \\ \mathit{circle} &= \mathit{tuple}(\mathit{point},\mathit{real}) \\ \mathit{lines} &= \mathit{list}(\mathit{point}) \end{aligned}$$

Additionally, functions and predicates are allowed, e.g.,

$$\begin{array}{lll} x,\ y & : point & \rightarrow real \\ center & : circle & \rightarrow point \\ point\_distance & : point \times point & \rightarrow real \\ circle\_circle\_cut & : circle \times circle & \end{array}$$

3.1.3 Aggregation Functions

For the computation of aggregated values some standard functions are provided:

$$\begin{array}{lll} CNT & : list(\mathrm{s}) & \rightarrow int \\ SEL & : list(\mathrm{s}) \times nat & \rightarrow \mathrm{s} \\ LTS & : list(\mathrm{s}) & \rightarrow set(\mathrm{s}) \\ IND & : list(\mathrm{s}) & \rightarrow set(int) \\ APL_\sigma & : list(\mathrm{d}) & \rightarrow \mathrm{d} \end{array}$$

CNT counts the number of elements in a list, *SEL* selects the i-th element of a list, *LTS* converts a list to a set, *IND* gives the set of indices of a list (e.g., $IND(\mathrm{l})=\{1,2,3\}$ if $CNT(\mathrm{l})=3$), and APL_σ applies an operation σ to a list, where $\sigma : \mathrm{d} \times \mathrm{d} \rightarrow \mathrm{d}$ is a binary operation on a data type $\mathrm{d} \in \mathrm{DATA}$. Most functions (e.g., *CNT* and *APL*) are used analogously for the bag- or set-valued case. The *APL* functions are employed in order to define aggregation functions. For instance, if we want to define the *SUM* of a set x of integers, we simply specify

$$SUM(\mathrm{x}) \equiv APL_+(\mathrm{x})$$

In the same way we use the aggregation functions *MIN*, *MAX*, and *AVG* computing the minimum, maximum, and average of a collection of numerical values. There are also standard predicates like $IN : \mathrm{s} \times list(\mathrm{s})$, which determines whether a given element is element of a given list (analogously for bags and sets).

3.1.4 Extended Entity-Relationship Schemas

The central notion of the EER model is the extended Entity-Relationship schema. Such a schema provides names for the entity (or object) types, for the relationship types between these entity types, and for the attributes of entities. It is also possible to define components (or in other words object-valued attributes). Attributes and components are allowed to be list-, bag-, or set-valued.

Formally, an extended Entity-Relationship schema consists of

- finite sets ENTTYPE, RELTYPE, ATTRIBUTE, COMPONENT, and CONSTRUCTION together with functions
- $participants : \mathrm{RELTYPE} \rightarrow \mathrm{ENTTYPE}^+$,

- *source* : ATTRIBUTE $\rightarrow$ ENTTYPE $\cup$ RELTYPE,
- *destination* : ATTRIBUTE $\rightarrow$ { d, *set*(d), *bag*(d), *list*(d) | d $\in$ DATA },
- *source* : COMPONENT $\rightarrow$ ENTTYPE,
- *destination* : COMPONENT $\rightarrow$ { e, *set*(e), *bag*(e), *list*(e) | e $\in$ ENTTYPE }, and
- *input*, *output* : CONSTRUCTION $\rightarrow 2^{\text{ENTTYPE}} - \emptyset$.

There are several additional conditions these schemas have to fulfil: for two distinct construction types the output types must be disjoint; no cycles are allowed for construction types, i.e., it is not allowed that there is an entity type being constructed from itself. These conditions guarantee a unique construction path for every entity.

Let us explain these notions using our running geographic example. Here, we have the following identities.

- ENTTYPE := { PERSON, COUNTRY, TOWN, RIVER, SEA, LAKE, WATERS }
- RELTYPE := { *is_mayor_of, lies_in, lies_at, flows_through, flows_into* }
- ATTRIBUTE := { *pname* : PERSON $\rightarrow$ *string*, *addr* : PERSON $\rightarrow$ *list*(*address*), ... }

  The above line specifies not only the set ATTRIBUTE but the functions *source* and *destination* for ATTRIBUTE as well.
- COMPONENT :=

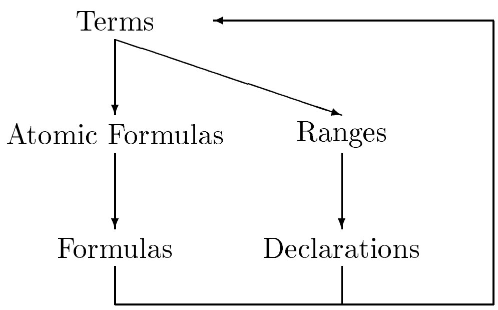
  { *boss* : COUNTRY $\rightarrow$ PERSON, *ministers* : COUNTRY $\rightarrow$ *list*(PERSON) }
- CONSTRUCTION := { *are* (SEA, LAKE, RIVER ; WATERS) }

The interpretation of an EER schema (i.e., an EER database or an instance of an EER schema) associates finite sets of surrogate values to the entity types and finite subsets of the cartesian products (of the corresponding surrogate values) to the relationship types. Attributes are interpreted by finite functions from the surrogate values of the entity types into (the interpretation of) the (possibly multi-valued) data types. Components are finite functions from the surrogate values of the source entity types into surrogate values or bags, lists, or sets of surrogates of the destination entity types. Construction types correspond to injections from the union of the output surrogates into the union of the input surrogates. Thus, for example, the injection for the construction *are* gives the SEA, RIVER, or LAKE corresponding to a given entity of type WATERS. In general, the union of the output surrogates will be a proper subset of the union of the input surrogates. By additionally specifying an equality symbol in the type construction triangle, it can be required that there is a bijection between input and output surrogates. All functions corresponding to attributes and components are partial: apart from data values or surrogates they may yield an undefined value $\perp$.

Let us give a short, explaining remark concerning construction types, which support the well-known abstraction concepts specialization, generalization, and partition. These concepts are covered as special cases of this more general approach. In the case of specialization, we have one input type and one output type, and the injection gives the corresponding entity (in the superclass) of the specialized entity. In the case of generalization, there are more than one input type (like the above SEA, RIVER, and LAKE) and one output type (like the above WATERS). In the case of a partition, we have one input type, e.g., PERSON, and more than one output type, e.g., WOMAN and MAN. The injection from the union of the entities of the types WOMAN and MAN into the entities of type PERSON guarantees that each PERSON is either a WOMAN or a MAN.

3.2 The EER Calculus

The EER calculus defined for the EER model introduced above allows to formulate queries against an EER database. But it can also be employed to formulate integrity constraints. Its main building blocks are terms, atomic formulas and formulas analogously to the classical domain or tuple calculus.



However, the main difference is that our calculus is not hierarchical: we additionally introduce ranges, which are special set-valued terms, and we allow to bind variables in declarations to these ranges; on the one hand ranges and formulas can again be used for the formulation of bag-valued terms, on the other hand there are special aggregation functions converting multi-valued terms into single-valued ones. This will be explained by an example at the end of this section. We now introduce the different syntactic categories of the calculus and give some examples.

3.2.1 Simple Declarations

We start with a first simple form of declarations binding variables to finite sets. The allowed forms are $v : e$ with $e \in$ ENTTYPE or $v : r$ with $r \in$ RELTYPE. Thus, p : PERSON or li : *lies_in* are correct declarations. Throughout the following examples, we assume variables for the entity types of our running example consisting of a single lower case letter like the above p and variables for the relationship types consisting of at least two lower case letters like the above li.

3.2.2 Terms

There are various possibilities for the formulation of terms.

- Variables and application of operation symbols are terms. Operation symbols in this context are: data operations, standard operations of the parameterized types (like *CNT* or *LTS*), attributes and components. For instance, *age*(*p*)+42 is a term of sort *int*, *CNT*(*addr*(*p*)) is of sort *int*, *boss*(*c*) is of sort PERSON, and *point_distance*($point_1$,$point_2$) is of sort *real* (with $point_1$ and $point_2$ variables of data type *point*).

- If we have a type construction, then $e_{in}(t_{out})$ and $e_{out}(t_{in})$ are terms of sort e_{in}, respectively e_{out} assuming e_{in} is an input type and e_{out} an output type of the construction. For example, RIVER(*w*) is a term of sort RIVER assuming a variable *w* of type WATER.

- If *r* is a relationship type with *n* argument types and *v* is a relationship variable for *r*, then *v.i* is a term for $i \in 1..n$. For instance, *li.1* and *li.2* are terms of sort TOWN and COUNTRY, respectively. Additionally, role names are also allowed in this place instead of numbers (e.g., *li.*TOWN or *li.*COUNTRY).

- The most important form of term is the following.

  -[t_1, ..., t_n | d_1, ..., d_k $\wedge$ ϕ]-

  The value of this expression is a bag (or multi-set) allowing multiple occurrences of elements. The term has the sort *bag*(*tuple*(sort(t_1),...,sort(t_n))). Here, the t_is are the result terms, the d_is are variable declarations and ϕ is a qualifying formula. In the case n=1, the sort of the term is simply *bag*(sort(t_1)). For example,

  -[*age*(*p*) | *p* : PERSON]-
  -[*tname*(*t*), *cname*(*c*) | *t* : TOWN, *c* : COUNTRY $\wedge$ *lies_in*(*t*,*c*)]-

  are terms of sort *bag*(*int*) and *bag*(*tuple*(*string*,*string*)), respectively. They compute the multiset of ages of currently existing persons and the multiset of pairs (town name, country name) such that the town lies in the country. If the town name is not unique within one country then the pair (town name, country name) appears more than once in the bag. Only declared variables are allowed to appear in result terms and the cardinality of the resulting bag is mainly determined by the different bindings for these declared variables.

Please note, terms are allowed to have free variables.

3.2.3 Formulas

Most of the rules for building formulas in the calculus are quite standard.

- The application of predicate symbols to terms yields correct formulas. Predicate symbols in this context are data predicates, standard predicates of the parameterized types, or names of relationship types. For example, *circle_circle_cut*($circle_1$,$circle_2$) with $circle_1$ and $circle_2$ variables of data type *circle*, *p IN ministers*(*c*), or *lies_in*(*t*,*c*) are formulas.

- For a term t, *UNDEF*(t) is a formula testing the definedness of the term. *UNDEF*(*boss*(c)) is a correct formula and yields true, if the *boss* of country *c* is unknown (in the current interpretation). Analogously we use *DEF*.

- In the context of type constructions t_{in} *IS* t_{out} is a formula. For instance, r *IS* w yields true, if the value of the variable r of type RIVER is equal to the value of the variable w of type WATERS.
- Apart from the possibilities mentioned above for formulas, the calculus also allows equality and inequality between terms with equal types and the usual logical connectives ($\neg$, $\wedge$, $\vee$, $\otimes$ for exclusive or, $\Rightarrow$, $\Leftrightarrow$) and quantifiers ($\forall$, $\exists$). Thus for instance,

  $LTB(ministers(c))$ = -[p | p : PERSON $\wedge$ $age(p) > 50$]-
  ($\forall$ t : TOWN) ($\exists$ c : COUNTRY) $lies\_in(t,c)$

  are formulas (*LTB* is another standard operation: it converts a List To a Bag).

Please note, like terms, formulas are allowed to have free variables.

3.2.4 Ranges and Final Form of Declarations

Ranges will be used to restrict the domain of a variable to a finite set of currently existing values.

- The first two simple forms of ranges have already been introduced in the context of simple declarations: e with $e \in$ ENTTYPE or r with $r \in$ RELTYPE are ranges.
- The other possibility for ranges are set-valued terms. For example,

  $LTS(ministers(c))$
  BTS -[$age(p)$ | p : PERSON $\wedge$ $age(p) \geq 21$]-

  are ranges (recall *LTS* converts a List To a Set and *BTS* a Bag To a Set).
- The last possibility for ranges allows the construction of powersets over a given range: If ρ is a range, then set(ρ) is also allowed as a range. For example, each element of set(PERSON) is a finite set of currently existing persons. Recall that (the interpretation of) PERSON is finite. Therefore, (the interpretation of) set(PERSON) will be finite as well.

We have started with a simple form of declaration for the ease of demonstration. The final form of declaration now employs ranges in order to bind variables to arbitrary finite domains, not just entity sets or relationships as in the preliminary form of declaration.

- Variables can be bound to the union of a finite number of ranges with equal types. For example,

  p : $LTS(ministers(c))$
  $string_1$: *BTS* -[$rname(r)$ | r : RIVER]- $\cup$ *BTS* -[$sname(s)$ | s : SEA]-

  are declarations.

- Sequences of declarations of the first form are allowed as well. The declared variables of declarations on the right may occur free in declarations on the left. Thus for instance,

  t : -[t' | t' : TOWN $\wedge$ *lies_in*(t',c)]- ; p : *LTS*(*ministers*(c)) ; c : COUNTRY

  is a correct declaration of the variables c, p and t. First, the variable c is bound to a currently existing country (surrogate), then the variable p is bound to a minister of this country and the variable t to a town lying in this country.

Please note, like terms and formulas, ranges and declarations are allowed to have free variables.

3.2.5 Queries, Integrity Constraints, and Transitive Closure

Queries in our calculus are now simply terms over data types without free variables and integrity constraints are formulas without free variables. As a simple example, consider the following integrity constraint requiring that the average age of a country's ministers has to be less than or equal to 65.

$$\forall \text{ (c : COUNTRY) (} \mathit{AVG} \text{ -[} \mathit{age}(p) \mid p : \text{PERSON} \wedge p \ \mathit{IN} \ \mathit{ministers}(c) \text{]- } \leq 65 \text{)}$$

This example reveals the non-hierarchical nature of the calculus: the formula '*p IN ministers*(c)' appears inside the term '-[*age*(p) | p : PERSON $\wedge$ *p IN ministers*(c)]-'; this term in turn is part of the complete formula.

Apart from all calculus constructs mentioned above, we will also use the transitive closure R^+ of a binary relationship R(E,E) with E$\in$ENTTYPE, R$\in$RELTYPE, and participants(R)=<E,E>. Recall that $R(v_1,v_2)$ is a valid formula provided v_1 and v_2 are variables for entity type E. We will also allow the formula $R^+(v_1,v_2)$ as a shorthand for the following complex formula with free variables v_1 and v_2:

$$\begin{array}{l} \exists \text{ (es:set(E)) } \Big[_A \ \forall \text{ (v:E) [v } \mathit{IN} \text{ es } \Leftrightarrow \text{ [} R(v_1,v) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(} \exists \text{ (v':E) v' } \mathit{IN} \text{ es } \wedge \text{ R(v',v))]] } \Big]_A \\ \qquad\qquad \wedge \\ \qquad\qquad [\ v_2 \ \mathit{IN} \text{ es]} \end{array}$$

Apart from parenthesis (and) we use brackets [and] to point out the structure of formulas more clearly. The A-part of the formula determines that for a given entity v_1 the set es contains exactly those entities that are directly or indirectly reachable from v_1 via relationship R.

4 A Conceptual Model for ACT ONE

This section presents a database schema for ACT ONE formulated in the EER model and introduces some integrity constraints in the EER calculus using the notations of section 3. The integrity constraints are mainly intended to model context conditions of ACT ONE as given in [Cla88]. Some of the constraints are introduced to deal with incomplete specifications. The aim of this section is to give a concrete example of how specification systems can be supported by database methodology. To keep the presentation in a reasonable size, we have selected only some of the context conditions of [Cla88]. Additionally, we have omitted the modeling of *ACT texts* and of *initial constraints*.

The main questions when providing a database schema are: what is to be modelled and how is it to be modelled? Usually there are different possibilities to describe the items of interest. We have decided to concentrate on ACT ONE types [1] because they are the main specification units a specifier focuses on. The database schema for types provides information about their

- syntactical construction,
- flat representation, and
- state of incompleteness.

The syntactical construction of types is mainly given by the part of the schema that models some kind of abstract syntax tree.

The flat representation [2] is given by the part of the schema that models parameterized specifications with additional information to cope with shared subspecifications. Parts of this schema are also used for the syntactical representation of types, because parameterized specifications are the basic types of ACT ONE upon which structured types, i.e., types using structuring mechanisms like extension, union, etc., are constructed. Flat representations of types are used in this paper to formulate integrity constraints corresponding to context conditions of the language. Additionally, they will be used by various tools and should therefore by stored in the database. Note that a flat representation itself has to be provided by some tool.

When dealing with larger specifications one usually employs a mixture of bottom-up and top-down development. Such a strategy results in a stepwise construction of types that may have different degrees of incompleteness (see [BCC90] for a more detailed discussion of this topic). To support such a development strategy the entity type TYPE in the schema has some attributes that are intended to support incomplete situations.

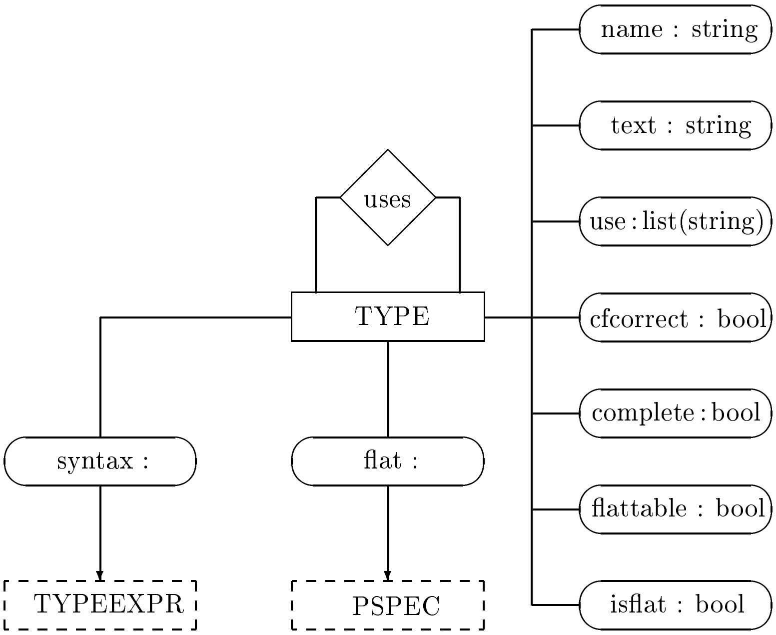
Section 4.1 presents the database schema of types by providing the entity type TYPE together with its attributes and relationships. Some integrity constraints, mainly concerning the use relationship and the handling of incomplete situations, are stated there. Section 4.2 deals with the syntactical construction of types by providing the part of the schema that models abstract syntax trees. We have decided to state nearly no integrity constraints in this section, because they would pertain to the context free part of the syntax, and we considered them as being not of so much importance in our presentation. Section 4.3 then presents the schema for flat representations. A lot of integrity constraints are given there corresponding to context conditions of ACT ONE.

[1] Note that we now have two notions of type, viz. ACT ONE types and EER entity types. In the following we use *type* for ACT ONE types and *entity type* when dealing with EER entity types.

[2] Flat representations of ACT ONE specifications correspond roughly to the so called first level of semantics (see [EM85] for details).

4.1 ACT ONE Types

An ACT ONE type is modelled by the entity type TYPE:



Entity type TYPE has the following attributes:

- *syntax:* This is an object valued attribute (i.e., a component) describing the syntactical appearance of the corresponding type. This attribute may be undefined and will be defined after successful context free analysis.
- *flat:* This component describes the flat representation of types. It may be undefined and will be defined after successful context sensitive analysis.
- *name:* Name of the type.
- *text:* Textual representation of the type; provided by some editor.
- *use:* List of names of used types.
- *cfcorrect:* Indicates whether the textual representation has been checked syntactically, i.e., is context free correct, and a syntax tree has been built.
- *complete:* Indicates whether all types in the *use* list are in the database.
- *flattable:* Indicates whether a flat representation of the type can be computed.
- *isflat:* Indicates that flat representation is available.

In order to keep the entity-relationship diagram in a reasonable size, we split it into several parts. An entity type in a dashed box constitutes an interface and this entity type is described in detail elsewhere.

There is one relationship for entity type TYPE, *uses*, that describes the use relation of types. In contrast to attribute *use* that contains a list of types or more precise type names that are necessary for context sensitive analysis, but may not be in the database, the relationship *uses* can connect only those type that are actually in the database.

The following integrity constraints can be stated.

A type must have at least a name and a textual representation:
$(\forall t : \text{TYPE})[\mathit{DEF}(\mathit{name}(t)) \land \mathit{DEF}(\mathit{text}(t))]$

Type names must be unique:
$(\forall t_1, t_2 : \text{TYPE})[\ t_1 \neq t_2 \Longrightarrow \mathit{name}(t_1) \neq \mathit{name}(t_2)]$

The use relationship must be acyclic:
$(\forall t : \text{TYPE})[\neg\ \mathit{uses}^{+}(t,t)]$

If a type has been checked syntactically its syntax tree and use list must be available:
$(\forall t : \text{TYPE})[\ \mathit{cfcorrect}(t) = \mathit{TRUE} \Longrightarrow \mathit{DEF}(\mathit{syntax}(t)) \land \mathit{DEF}(\mathit{use}(t))]$

A type is complete if all used types are in the database:

$$
\begin{array}{l}
(\forall t : \text{TYPE})[\\
\quad \mathit{complete}(t) = \mathit{TRUE} \\
\quad \Longleftrightarrow \\
\quad [\ \mathit{cfcorrect}(t) = \mathit{TRUE} \land \\
\quad\quad (\forall n : \mathit{LTS}(\mathit{use}(t)))(\exists \text{t'}: \text{TYPE})[\mathit{name}(t') = n \land \mathit{uses}(t,t') \land \mathit{cfcorrect}(t') = \mathit{TRUE}]] \\
]
\end{array}
$$

A type is flattable if all its used types are already flat:

$$
\begin{array}{l}
(\forall t : \text{TYPE})[\\
\quad \mathit{flattable}(t) = \mathit{TRUE} \\
\quad \Longleftrightarrow \\
\quad [\mathit{complete}(t) = \mathit{TRUE} \land [(\forall t' : \text{TYPE})\ \mathit{uses}(t,t') \Longrightarrow \mathit{isflat}(t') = \mathit{TRUE}] \\
]
\end{array}
$$

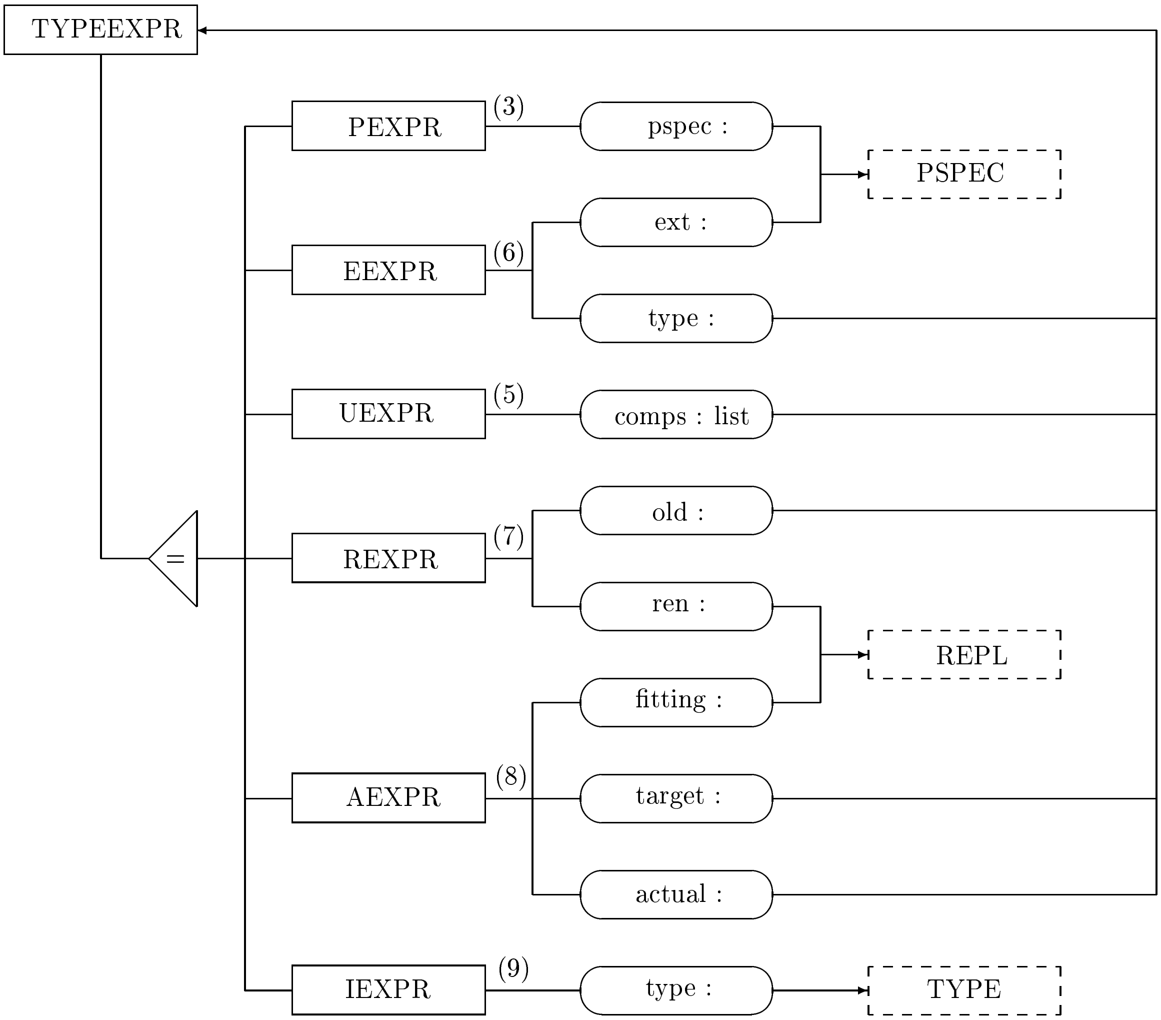
Note that states of incompleteness are indicated by several flags. These flags will be set by appropriate tools and may also lead to tool invocations.

4.2 Syntactical Construction

The syntactical construction of types is given by a schema modeling some kind of abstract syntax tree. The structure of the schema is derived from the following:

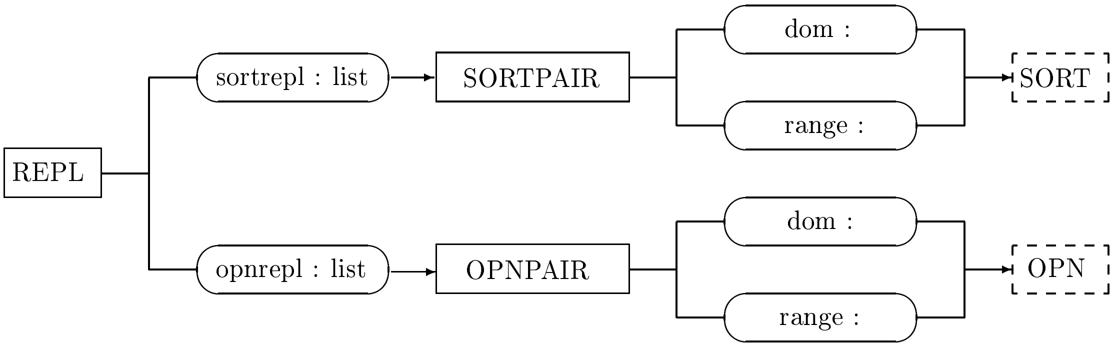
- Nonterminals are represented by entity types.

- Productions are represented by components. The numbers in the diagram reference the production of the ACT ONE grammar.
- The recursive nature of the grammar is captured by a type construction.



Please note that there are two components called type in the above diagram. We consider the first occurrence of type to be a shorthand for EEXPR.type and the second one to stand for IEXPR.type. Analogously we will do for other component names further below.

The entity type REPL represents sort and operation replacements. Its component sortrepl gives a list of sortpairs representing the given sortname and the new one. Analogously, there is a component opnrepl for operation replacements.



Here, we state only one integrity constraint for syntactical constructions. A lot of further constraints may be stated that require the definedness of attributes and components, for example.

All renamed sorts in a renaming must be disjoint:

$(\forall rn$:REXPR$)(\forall i, j$:$IND(sortrepl(ren(rn))))[$

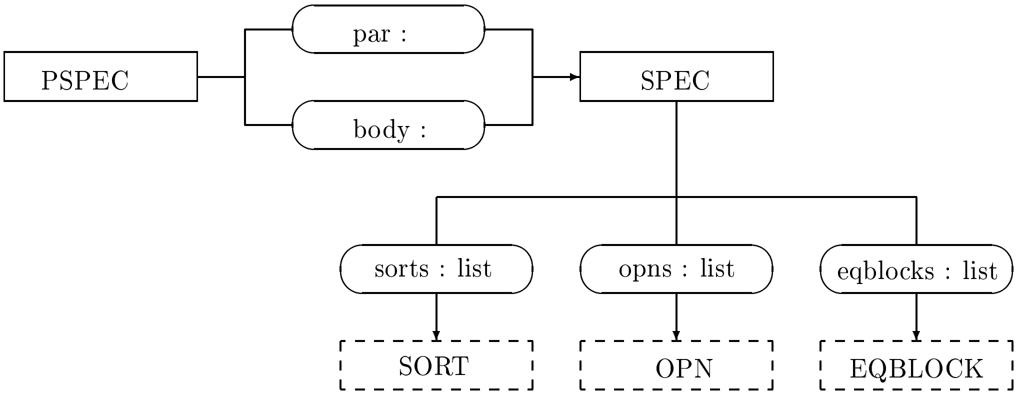
$i \neq j \Longrightarrow name(dom(SEL(sortrepl(ren(rn)),i))) \neq$
$name(dom(SEL(sortrepl(ren(rn)),j)))$

$]$

4.3 Flat Representations

Flat representations correspond to parameterized specifications with additional information. To handle subspecifications in an appropriate way, sorts and operations get associated places of definition that describe the origins of these items. Additionally sort dependencies are recorded to associate origins and to control implicit renamings when using the actualization mechanism. Entity type PSPEC represents flat specifications.

The entity type PSPEC consists of two components of entity type SPEC that describe the parameter part and body part of a parameterized specification. Entity type SPEC defines usual algebraic specifications with sorts, operations, and equations.



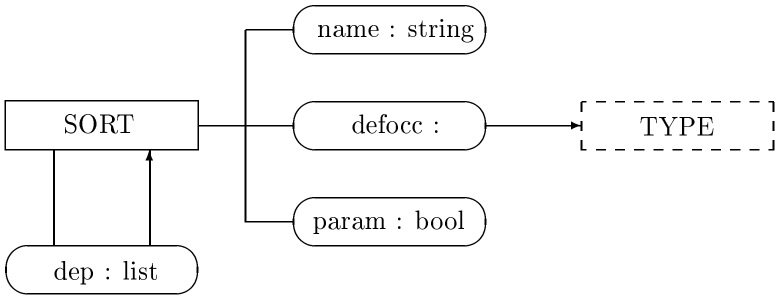
Sort names must be unique within one specification:

$(\forall s$:SPEC$)(\forall i,j$:$IND(sorts(s)))[$
$\quad i \neq j \Longrightarrow name(SEL(sorts(s),i)) \neq name(SEL(sorts(s),j))$
$]$

Parameter and body sorts must be disjoint:
$(\forall ps$:PSPEC$)(\forall i$:$IND(sorts(par(ps))))$ $(\forall j$:$IND(sorts(body(ps))))[$
$\quad name(SEL(sorts(par(ps)),i)) \neq name(SEL(sorts(body(ps)),j))$
$]$

Entity type SORT has the following attributes and components:

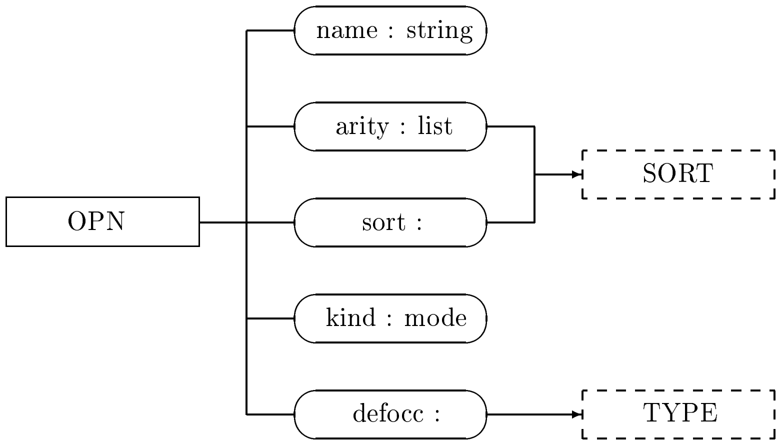
- *name:* String of characters denoting the name of the sort.
- *param:* Flag for parameter sorts.
- *defocc:* Reference to the type that introduces this sort (definition occurrence, origin).
- *dep:* List of sorts this sort depends upon. Is used in actualizations for implicit renaming (e.g., Set(Data) → Set(Char), see Section 2) and to generate new origins.



Parameter sorts must depend only on themselves:
$(\forall s$:SORTS$)[param(s) = TRUE \Longrightarrow CNT(dep(s)) = 1 \wedge SEL(dep(s),1) = s]$

Entity type OPN has the following attributes and components:

- *name:* String of characters denoting the name of the operation.
- *arity:* List of sorts on which the operation is defined.
- *sort:* Result sort of the operation.
- *kind:* Indicates the way the operation may be used in terms. Employs a special enumeration data type mode = {prefix, infix, if-then-else}.
- *defocc:* Reference to the type that introduces this opn.



The target sort of an operation must exist in the flat representation of the type that introduces this operation:

$(\forall opn : \mathrm{OPN})[$

$\quad DEF(flat(defocc(opn)))$

$\quad \Longrightarrow$

$\quad [sort(opn)\ IN\ sorts(body(flat(defocc(opn)))) \vee sort(opn)\ IN\ sorts(par(flat(defocc(opn))))]$

$]$

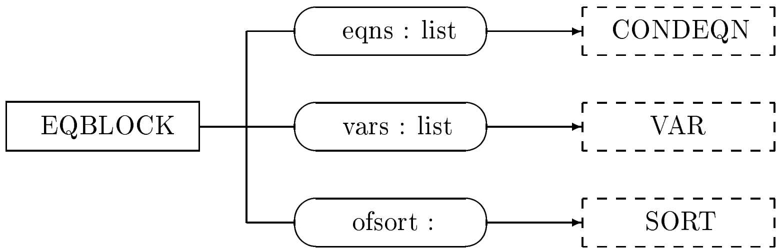
A similar constraint holds for sorts given by the arity component.

Infix operations must have two arguments:

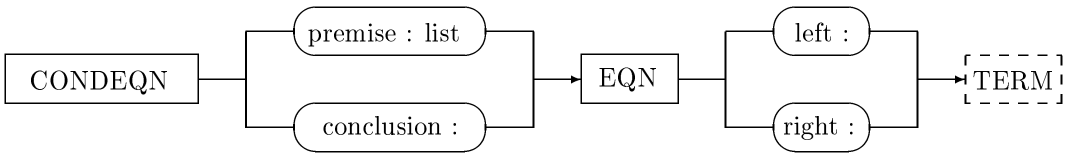
$(\forall opn : \mathrm{OPN})[\ kind(opn) = infix \Longrightarrow CNT(arity(opn)) = 2\ ]$

An analogous integrity constraint like the above one can be formulated for the conditional if-then-else.

An equation block consists of a list of conditional equations all having the same sort, a list of variables that are used within the equations of this block, and a sort indication.

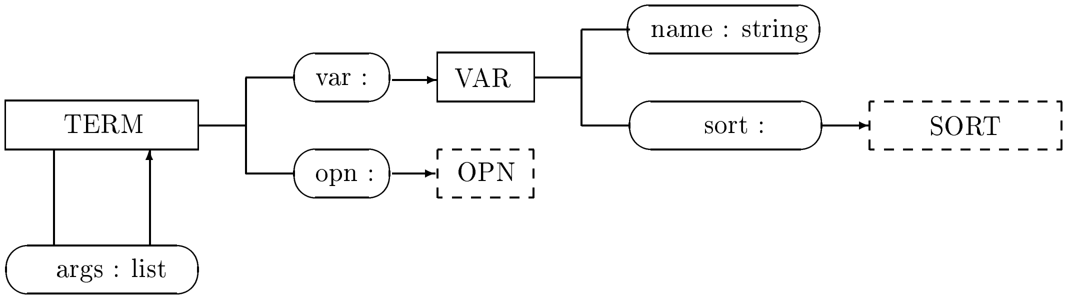


As usual, a conditional equation consists of a list of premise equations and a conclusion equation.



The entity type TERM is characterized by the following components.

- *var:* If the term consists of a single variable, this component returns the variable.
- *opn:* If the term has at least one operation symbol, the component opn returns the top level operation symbol. Recall that components are allowed to be optional. Therefore exactly one of these first two components must be null and the other one non-null.
- *args:* For a proper term, i.e., a term with at least one operation symbol, the component args gives the list of subterms of the given one.



A term is either a variable or it has a top-level operation symbol:
$(\forall t : \mathrm{TERM})[DEF(var(t)) \otimes DEF(opn(t))]$

Variables have no arguments:
$(\forall t : \mathrm{TERM})[DEF(var(t)) \Longrightarrow UNDEF(args(t))]$

The number of arguments (on top-level) of a proper term must coincide with the number of arguments of the top-level operation symbol:
$(\forall t : \mathrm{TERM})[DEF(opn(t)) \Longrightarrow CNT(args(t)) = CNT(arity(opn(t)))]$

The sorts of arguments (on top-level) of a proper term must coincide with the sorts given by the arity of the top-level operation symbol:

$$
\begin{aligned}
&(\forall t : \mathrm{TERM})[\\
&\quad DEF(opn(t)) \Longrightarrow\\
&\qquad (\forall i : IND(args(t)))[\\
&\qquad\quad DEF(opn(SEL(args(t),i))) \Longrightarrow sort(opn(SEL(args(t),i))) = SEL(arity(opn(t)),i)\\
&\qquad\quad \wedge\\
&\qquad\quad DEF(var(SEL(args(t),i))) \Longrightarrow sort(var(SEL(args(t),i))) = SEL(arity(opn(t)),i)\\
&\qquad]\\
&]
\end{aligned}
$$

5 Conclusion

The main contribution of this paper is a case study in applying database methodology in the realm of specification languages. We have provided a concrete database schema for the algebraic specification language ACT ONE. This schema is intended to provide a logical description of data items to be stored in a specification environment for ACT ONE.

The aim of this paper was to investigate how specification languages can be supported by database systems. We hope that the development of a concrete example may give insight on how to model an environment for a specification language.

Taking a closer look at our example, the following observations can be made:

- Two separate aspects pertaining to system support for specification languages have been described: data items to be stored and the state of incompleteness a specification may have.
- The state of incompleteness has been mainly modelled by several flags. How these flags are set and reset has not been described.
- The whole model has been centered around ACT ONE types, that have been identified as main items of interest. Especially incompleteness is associated with ACT ONE types.
- The context free part of the specification language has been translated into an appropriate schema.
- The context conditions have been translated into integrity constraints.
- Additional information (in our case flat representations) has been modelled by additional entity types with appropriate attributes.

Of course, the present proposal is rather incomplete compared with the information that is needed in a practical specification environment. There, we would like to store documentations, proofs, compilation results, etc. as well. Further aspects that have not been modelled are version control and dynamic database behaviour.

Although our paper was inspired by the work of [BCC90] and we tried to describe the same problems, our approach is quite different. In [BCC90] the design of the specification database of the ASSPEGIQUE environment is described by use of the algebraic specification language PLUSS. [BCC90] has tried to describe static aspects as well as dynamic ones. Because they employed a general specification language they presented a rather long winded specification, the sole aim of which was to describe certain states of incompleteness of specifications. Because we have employed a powerful data model we were able to describe the same affair in a few lines. But in contrast to them we have not yet specified dynamic aspects of the database.

Concerning questions of implementation of our design, we think one should not develop its own database, but should use existing database systems in order to avoid unnecessary programming efforts. This raises the question of how existing database systems could be used to implement the described design. One more or less conservative solution is to translate this extended Entity-Relationship schema into a relational one as has been suggested by [Hoh89]. Another approach is to employ an object-oriented database system like [Pro91]. This is subject to future research.

Bibliography

[BCC90] M. Bidoit, F. Capy, and C. Choppy. The design and specification of the ASSPEGIQUE database. *Proc. 1st Int. Symposium on Design and Implementation of Symbolic Computation Systems (DISCO '90)*, Springer, 1990.

[BC85] M. Bidoit, and C. Choppy. ASSPEGIQUE: An integrated environment for algebraic specifications. *Proc. Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'85), LNCS 186*, Springer, 1985, pp. 246–260.

[Cho88] C. Choppy. ASSPEGIQUE user's manual. Technical Report 552, L.R.I., 1988.

[Cla88] I. Claßen. Semantik der revidierten Version der algebraischen Spezifikationssprache ACT ONE. Technical Report 88/24, Technische Universität Berlin, 1988.

[Cla89] I. Claßen. Categorical Constructions for an Algebraic Specification Language. In *H.Ehrig, H. Herrlich, H.–J. Kreowski, G. Preuss (Eds.), Categorical Methods in Computer Science with Aspects from Topology. LNCS 393*, Springer, 1989, pp.124–141.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two level of semantics. Technical Report 83/03, Technische Universität Berlin, 1983.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. Algebraische Spezifikation Abstrakter Datentypen – Eine Einführung in die Theorie. Teubner Verlag, Stuttgart, 1989.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2*, volume 21 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1990.

[Got88] W. Gotthard. Datenbanksysteme für Software-Produktionsumgebungen. *Informatik Fachberichte 193.* Springer, 1988.

[GH91] M. Gogolla and U.Hohenstein. Towards a semantic view of an extended entity-relationship model. To appear in *ACM Transactions on Database Systems.*

[GW88] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI, August 1988.

[Han87] H. Hansen. The ACT-System: experiences and future enhancements. In *Recent Trends in Data Type Specification, LNCS 332*, pages 113–130. Springer, 1987.

[Hus85] H. Hussmann. Rapid Prototyping for Algebraic Specifications - RAP-System User's Manual. Technical Report MIP 8505, University of Passau, 1985.

[HG88] Uwe Hohenstein and Martin Gogolla. A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. Proc. 7th Int. Conf. on Entity-Relationship Approach, Carlo Batini (ed.), pp. 129 -148, 1988.

[Hoh89] Uwe Hohenstein. Automatic Transformation of an Entity-Relationship Query Language into SQL. *Proc. 8th Int. Conf. on Entity-Relationship Approach, F. Lochovski (ed.)*, 1989.

[Loe89] J. Loeckx. Obscure: Eine Spezifikations- und Verifikationsumgebung. Technical Report 89/06, Universität des Saarlandes, 1989.

[Pro91] D. Krömker, H. Steusloff, and H.-P. Subel. PRODIA und PRODAT – Dialog- und Datenbankschnittstelle für Systementwurfswerkzeuge. *Beiträge zur Graphischen Datenverarbeitung*, Springer-Verlag, 1989.

[ULL88] J. D. Ullman. Principles of Database and Knowledge-Base Systems: Vol. I. Computer Science Press, 1988.

[ULL89] J. D. Ullman. Principles of Database and Knowledge-Base Systems: Vol. II. Computer Science Press, 1989.