Generating an Algebraic Specification from an ER-Model

Ernst-Erich Doberkat
Chair for Software Technology
University of Dortmund
D-44221 Dortmund
Germany

September 19, 1995

Contents

List of Figures

Abstract

Entity-Relationship modelling is a rather intuitive technique for specifying the structure of complex data. The technique is popular in part because the structure of an ER-model is easily grasped, and it is usually supported by diagrams or other visualizing tools. This paper deals with a detailed analysis of ER-modelling with the goal of deriving an algebraic specification for a given ER-model. This is motivated by considerations regarding program specification for data intensive applications. We indicate haw the technique demonstrated here may be combined with formal techniques for specifying the functional behavior of a system.

1 Introduction

Entity-Relationship modelling is a rather intuitive technique for specifying the structure of complex data. The technique is popular in part because the structure of an ER-model is easily grasped, and it is usually supported by diagrams or other visualizing tools.

This paper deals with a detailed analysis of ER-modelling with the goal of deriving an algebraic specification for a given ER-model. This is motivated by considerations regarding program specification for data intensive applications: when such a program system is specified, then its functionality should be described jointly with the data on which the program is to operate. Describing the functionality formally with a Petri net results in a partial description of the system, the semantics being provided by the semantics of Petri nets. Describing the data using an ER-model results in another partial description, albeit one which does not have a formal semantics. So linking these descriptions results in an unbalanced situation: the functional specification is on semantically firm grounds, the specification of the data is not. Providing an algebraic specification for the ER-model helps, since the machinery of algebraic specifications permits the formulation of models through which the semantics of a specification is obtained.

Similarly, the functionality of a program may be described with algebraic techniques (ASL [21], SPECTRUM [2] or π [5, 4] come to mind). Here the data are either described using the framework for the functional specification, or the data are described informally. The former alternative is quite unpractical when it comes to dealing with data the structure of which is complex, probably even violating the accepted principle of *separation of concerns*, here applied to separating functions from data. The latter alternative is unsatisfactory since it provides only a partially formal description, leaving a gap in which mathematical arguments cannot be applied. Thus an algebraic specification of ER-models would close this gap.

In order to study this problem, we deal with a rather simple instance of ER-models which has entities, binary relations, and attributes on entities as well as on relations. The only relation we consider between entities is the IsA-relation, providing a mechanism for simple inheritance. Relations may have some restrictions: among others, they may be total or $N:1$ relations; attributes may also be total, total attributes may be key attributes. This scenario is usually not sufficient for the purposes of semantic data modelling, but it is rich enough to be interesting on its own, and it permits demonstrating the techniques for obtaining an algebraic specification from an ER-model; more complicated models may be delt with in a similar fashion, but of course at the cost of a more involved technical machinery.

Given an ER-model $\mathcal{M}$, the dependecies between the objects in the model may be of a varied and rather subtle kind (e.g. deleting a pair (x, y) from a total relation which has the entity E as its domain and for which the entity E' is in the transitive closure of IsA implies among others deleting x from E' and deleting all pairs having x as the first component from all relations having E' as their domain). We capture this by building up an abstract data type (ADT) from $\mathcal{M}$, the underlying structure of which is a graph with nodes coming essentially from the objects in $\mathcal{M}$ (there are some other nodes, too). This graph contains directed as well as undirected vertices, the directed edges reflecting the structure of the IsA-relation, the undirected ones being generated from relations and from attributes. Each node in the graph is labelled with a sort which is generated or constructed from the corresponding object in $\mathcal{M}$. The ADT operates on this graph and specifies for each operation in $\mathcal{M}$ the effect it has on each node. This gives rise to local axioms which may be thought of decorating each node, the entire decoration constitutes the set of axioms comprising the interesting part of the specification. There are other, basic parts to the specification: since we use operations from set theory for the basic constructions, we have to provide an algebraic specification for that part of set theory relevant here (essentially simple manipulations on finite set, finite Cartesian products, and on finite associative maps). The primitive constructions (e.g. describing the specification $\mathsf{SET}(s)$ of sets over s for a sort s) are parametrized according to their respective basic sorts, so that a simple import mechanism takes care of making all basic specifications available through collecting the corresponding instantiations for the primitive parts. We build our construction on this import by enriching it with new function symbols and new axioms.

Section 2 introduces algebraic specifications and defines the flavor of ER-model we are working with. Section 3 then shows how to construct a graph as the basic data structure from a given ER-model, and the specification is constructed in detail in section 4. We briefly indicate how to combine condition/event nets with (the terms of) an algebraic specification in section 5 and discuss implementation issues in section 6. The final section 7 discusses related work and suggests some extensions. The basic constructions mentioned above are partly done in section 2, partly delegated to the appendices A.1 and A.1.

2 Preliminaries

In this section we provide a brief definition of algebraic specifications and of the version of ER-models which are considered here.

2.1 Algebraic Specifications

We follow the notation in [22] in introducing algebraic specifications. Let Σ be a signature, i.e., $\Sigma = \langle S, \Gamma \rangle$, where S is a set of *sorts* and Γ is a set of *function symbols* disjoint from S; each $f \in \Gamma$ is associated its domain $s_1, \ldots, s_n$ and its co-domain s_{n+1}, so that $f : s_1 \times \ldots \times s_n \to s_{n+1}$. If $n = 0$, f is a constant. Let $X = (X_s)_{s \in S}$ be an S-sorted set of free variables, then the S-sorted set $\mathcal{T}(\Sigma, X) = (\mathcal{T}(\Sigma, X)_s)_{s \in S}$ is the smallest S-sorted set $(\mathcal{V}_s)_{s \in S}$ such that these three conditions hold:

1. $\forall s \in S : X_s \subset \mathcal{V}_s$,
2. whenever $f \in \Gamma$ such that $f : \to s$, then $f \in \mathcal{V}_s$,
3. whenever $f \in \Gamma$ such that $f : s_1 \times \ldots \times s_n \to s_{n+1}$, and $t_1, \ldots, t_n$ are terms such that $t_1 \in \mathcal{V}_{s_1}, \ldots, t_n \in \mathcal{V}_{s_n}$, then $f(t_1, \ldots, t_n) \in \mathcal{V}_{s_{n+1}}$.

Thus $\mathcal{T}(\Sigma, X)$ contains free variables as well as constants, and it is closed under the application of function symbols from Γ. Let $=_s$ be the equality relation on $\mathcal{T}(\Sigma, X)_s$, then the set $\mathrm{wff}(\Sigma, X)$ of *well defined formulas* contains all the equalities $t_1 =_s t_2$ for $t_1, t_2 \in \mathcal{T}(\Sigma, X)_s, s \in S$, and is closed under disjunction, negation and existential quantification (i.e. if $G, H \in \mathrm{wff}(\Sigma, X)$, and $s \in S$ is a sort, then $G \vee H, \neg G, \exists x : s.G$ are members of $\mathrm{wff}(\Sigma, X)$.

An algebraic specification $\langle \Sigma, \Delta \rangle$ is a signature Σ together with a set $\Delta \subset \mathrm{wff}(\Sigma, X)$ of axioms: the signature describes the function symbols together with their syntax, the axioms describe how these functions relate to each other.

Now let $\langle \Sigma, \Delta \rangle$ be an algebraic specification with $\Sigma = \langle S, \Gamma \rangle$. Suppose each sort $s \in S$ is assigned a carrier set A_s, and let each function symbol $f \in \Gamma$ with

$$f : s_1 \times \ldots \times s_n \to s_{n+1}$$

be associated with a map

$$f^{\mathcal{A}} : A_{s_1} \times \ldots \times A_{s_n} \to A_{s_{n+1}}.$$

This association $f \mapsto f^{\mathcal{A}}$ is an interpretation of the function symbol f, it carries over in a natural way to ground terms, i.e. terms without free variables, so that we can talk about the interpretation $t^{\mathcal{A}}$ of a ground term t. Variables are taken care of by *valuations*. A valuation $v = (v_s : X_s \to A_s)_{s \in S}$

$$\begin{array}{rcl} \texttt{if}\ \mathit{true}\ \texttt{then}\ x\ \texttt{else}\ y\ \texttt{fi} & =_s & x \\ \texttt{if}\ \mathit{false}\ \texttt{then}\ x\ \texttt{else}\ y\ \texttt{fi} & =_s & y \end{array}$$

Figure 1: Default axiom for s-sorted branch

---

is an S-sorted familiy of maps assigning values to variables. Putting $v_s^*(x) := v_s(x)$, if $x \in X_s$, $v_s^*(c) := c^{\mathcal{A}}$, if $c :\to s$ is a constant, and setting

$$v_s^*\left(f(t_1, \ldots, t_k)\right) := f^{\mathcal{A}}\left(v_{s_1}^*(t_1), \ldots, v_{s_k}^*(t_k)\right)$$

whenever $f : s_1 \times \ldots \times s_k \to s$ is a function symbol, and $t_1 \in \mathcal{T}(\Sigma, X)_{s_1}, \ldots, t_k \in \mathcal{T}(\Sigma, X)_{s_k}$ are terms, v is extended to an S-sorted map $v^* = (v_s^* : \mathcal{T}(\Sigma, X)_s \to A_s)_{s \in S}$.

A well-formed formula G is valid in $\mathcal{A}$ iff the interpretation of G is a true statement in $\mathcal{A}$ no matter what values are assigned to its free variables ([22] defines this more formally, but this rather intuitive definition will do for our present purposes). The S-sorted set $\mathcal{A} = (A_s)_{s \in S}$ is a *model* for the specification iff all the axioms are valid in $\mathcal{A}$. The model $\mathcal{A}$ is *term-generated* iff each element of a carrier set can be represented as the interpretation of a suitably chosen term. This notion of a term-generated model formalizes the intuitive notion of the implementation of a specification. Let $\mathcal{B}$ be another model. A *homomorphism* $\Psi : \mathcal{A} \to \mathcal{B}$ is an S-sorted set of maps $(\Psi_s : A_s \to B_S)_{s \in S}$ such that for each function symbol $f : s_1 \times \ldots \times s_k \to s$ and each $a_1 \in A_{s_1}, \ldots, a_k \in A_{s_k}$ the equality

$$\Psi_s(f^{\mathcal{A}}(a_1, \ldots, a_k)) = f^{\mathcal{B}}(\Psi_{s_1}(a_1), \ldots, \Psi_{s_k}(a_k))$$

holds. $\mathcal{A}$ is an inital model iff for each model $\mathcal{B}$ there exists a unique homomorphism $\Psi : \mathcal{A} \to \mathcal{B}$, it is a terminal model iff for each model $\mathcal{B}$ there exists a unique homomorphism $\Psi : \mathcal{B} \to \mathcal{A}$. The initial semantics of an algebraic specification is given by all term-generated initial models, similarly for its terminal semantics. Its loose semantics consists of all term-generated models.

Specifications will be built incrementally, assuming that the primitive specification BOOL for the data type `Boolean` with the usual axioms and two different constants $true$ and $false$ is already provided for. It is understood that each specification contains BOOL. We will assume tacitly that with each sort s a ternary function symbol

$$\texttt{if . then . else . fi} : \mathrm{BOOL} \times s \times s \to s$$

(the conditional) is associated; the — usual — semantics is given in Fig. 1. Moreover we tacitly associate with s an error symbol

$$error_s :\to s$$

indicating that some extraordinary action will have to take place. The occurrence of $error_s$ will be data dependent (it will be found only in the branches of some conditionals) and will take care of integrity constraints; hence we leave it to specific interpretations to deal with $error$.

In general, specifications are built up incrementally:

specification $\langle \Sigma, \Delta \rangle$ **imports** $\langle \Sigma_1, \Delta_1 \rangle, \ldots \langle \Sigma_k, \Delta_k \rangle$ **end**

denotes the specification

$$\langle \Sigma + \Sigma_1 + \cdots + \Sigma_k, \Delta \cup \Delta_1 \cup \cdots \cup \Delta_k \rangle.$$

Here the sum operator $+$ is defined through

$$\langle S', \Gamma' \rangle + \langle S'', \Gamma'' \rangle := \langle S' \cup S'', \Gamma' \cup \Gamma'' \rangle,$$

This requires Γ' and Γ'' being compatible, i.e., each function symbol occurring in their intersection having the same signature. Extending the sum to more than two operands requires that the

$$
\begin{array}{rcl}
\neg(x & \in_s & empty_s) \\
x & \in_s & insert_s(x,b) \\
\neg(x & \in_s & delete_s(x,b)) \\
insert_s(x, insert_s(y,a)) & =_{\mathsf{set}(s)} & insert_s(y, insert_s(x,a)) \\
insert_s(x, delete_s(x,a)) & =_{\mathsf{set}(s)} & insert_s(x,a) \\
(delete_s(x, insert_s(x,a)) =_{\mathsf{set}(s)} a) & =_{\mathrm{BOOL}} & \neg(x \in_s a) \\
empty_s & \subset_s & a \\
insert_s(x,a) \subset_s b & =_{\mathrm{BOOL}} & (a \subset_s b) \wedge x \in_s b \\
a \subset_s b & \rightarrow & a \subset_s insert_s(x,b) \\
a \subset_s delete_s(x,b) & \rightarrow & a \subset_s b \\
a & \subset_s & insert_s(x,a) \\
delete_s(x,a) & \subset_s & a \\
a & \subset_s & a \\
(a =_{\mathsf{set}(s)} b) & =_{\mathrm{BOOL}} & (a \subset_s b \wedge b \subset_s a) \\
a \subset_s b \wedge b \subset_s c & \rightarrow & a \subset_s c
\end{array}
$$

Figure 2: Axioms for sets

operands are mutually compatible. The incremental approach has the advantage that it permits separation of concerns, as may be observed in the following sections.

Let $\mathcal{S}$ be an infinite set of sorts, and fix $s \in S$. Then $\mathsf{set}(s)$ is a fresh member of $\mathcal{S}$, and let the specification of sets of sort s be

$$\mathsf{SET}(s) := \langle \Sigma, \Delta \rangle$$

with

$$\Sigma := \langle \{s, \mathsf{set}(s)\}, \{empty_s, . \in_s ., . \subset_s ., . =_{set(s)} ., insert_s, delete_s\} \rangle$$

such that

$$
\begin{array}{rlcl}
empty_s : & & \rightarrow & \mathsf{set}(s) \\
insert_s : & s \times \mathsf{set}(s) & \rightarrow & \mathsf{set}(s) \\
delete_s : & s \times \mathsf{set}(s) & \rightarrow & \mathsf{set}(s) \\
. \in_s . : & s \times \mathsf{set}(s) & \rightarrow & \mathrm{BOOL} \\
. \subset_s . : & \mathsf{set}(s) \times \mathsf{set}(s) & \rightarrow & \mathrm{BOOL}
\end{array}
$$

($. \in_s .$, $. \subset_s .$ and $. =_{\mathsf{set}(s)} .$ are used as infix operators). The axioms suggested for Δ_s can be found in Fig. 2.

Suppose a model $\mathcal{A}$ for $\mathsf{SET}(s)$ is term-generated, then the usual property of the subset relation

$$a \subset_s^{\mathcal{A}} b \Leftrightarrow \forall x \in A_s : x \in_s^{\mathcal{A}} a \rightarrow x \in_s^{\mathcal{A}} b$$

holds; note that the equality of $A_{\mathsf{set}(s)}$ with the power set $\mathcal{P}(A_s)$ of A_s is not implied.

Similar to the construction $s \mapsto \mathsf{SET}(s)$ describing sets of a sort s we construct a specification $\mathsf{ABB}(s_1, s_2)$ for the description of all partial maps from s_1 to s_2 (see A.1) respectively $\mathsf{CART}(s_1, s_2)$ for all pairs of sort s_1 and s_2 (see A.1).

For the rest of the paper the indication of the sort will be omitted when talking about equality, since the sort under consideration will be clear from the context.

2.2 ER-Models

An entity-relationship model [20, 2.4] consists of entities, relationships on these entities and attributes both on entities and relations. Only binary relations will be considered for the sake of simplicity. Entities may be related by the IsA relation: E_1 IsA E_2 indicates that each instance of E_1 is also an instance of E_2, hence shares all the attributes defined on the latter entity. IsA^+ is the transitive closure of IsA. Multiple inheritance is not permitted (i.e, no entity may be related to more than one other entity via an IsA -relation). Names are supposed to be unique; in particular no attributes may be redefined in an IsA relation. As usual, entities are represented graphically as boxes, attributes as ovals, and relations as diamonds, resp. Mathematically, entities are represented as sets (the *extension* of an entity), relations as subsets of the Cartesian product for the sets representing the corresponding entities. If R relates the entities E_1 and E_2, the entities in E_1 (in E_2) are said to be in the *domain* (in the *co-domain*) of R. In the graphical representation the order of the factors for the product is not immediate, hence we number the corners of the diamond counterclockwise starting in the northern corner, identifying domain and co-domain uniquely. Attributes are usually represented as maps; as usual, an attribute is a *key* for an entity iff it uniquely determines each instance. A relation R is $N:1$ iff $b_1 = b_2$ is true whenever both aRb_1 and aRb_2 hold (i.e. whenever R is a partial map), i.e. iff for each instance a in the domain of R the set $\{b : aRb\}$ contains at most one element. In a similar way $1:N$ relations are characterized: R is an $1:N$ relation iff its inverse R^{-1} is $N:1$. A relation is said to be $N:M$ iff there are no restrictions concerning the domain or the co-domain of the pairs participating in the relation. That a relation is $N:1$ is indicated in the graphical representation by labelling the edge leading to the domain with an $*$, and a 1 as a label for the co-domain.

Fig. 3 displays an example for modelling a simple graphical user interface.

The entities are `window`, `button`, `textfield`, `menu entry`, moreover `trigger` and `text(fixed)`, both of which are related to `menu entry` via the IsA relation, and `output window` and `icon`, for which IsA `window` holds. The relations are `sequence`, which is an $N : M$ relation between `windows`, `residesIn`, a $1:N$ relation between `window` and `button`, `contains` relates `textfield` and `window` as an $N:1$ relation, `inMenu` is an $N:1$ relation between `trigger` and `window`, and finally `invocation` relates `trigger` and `menu entry` $1:N$. Attributes are e.g. `window layout` defined on entity `window` or `button position` defined on relation `residesIn`. As usual, key attributes are underlined, and total relations or attributes carry a dot where they are total.

3 The Graph

This section will construct a graph from an ER-model.

Given an ER-model $\mathcal{M}$, denote by $\mathcal{E}$, $\mathcal{R}$ and by $\mathcal{A}$ the set of entities, of relations, and of attributes, resp.; let $N_\mathcal{E}$, $N_\mathcal{R}$ and $N_\mathcal{A}$ be fresh and disjoint sets of nodes representing $\mathcal{E}$ and the domains and co-domains for the relations in $\mathcal{M}$, resp. $\mathcal{M}$, so that each $E \in \mathcal{E}$ is associated with a unique node $n_E \in N_\mathcal{E}$, similarly for $\mathcal{R}$. Construct a directed edge $n_{E_1} \to n_{E_2}$ iff E_1 IsA E_2 holds in $\mathcal{M}$. If R is a relation in $\mathcal{M}$ with E_1 as domain and E_2 as co-domain, construct non-directed edges between n_{E_1} and n_R and between n_R and n_{E_2}; additionally, generate two fresh nodes $n_{\delta(R)}$ and $n_{\gamma(R)}$ in $N_\mathcal{E}$ which are linked through the directed edges $n_{\delta(R)} \to n_{E_1}$ and $n_{\gamma(R)} \to n_{E_2}$ to their domain and co-domain, resp. (this reflects the fact that the domain and the co-domain of R have to be taken care of when it comes to manipulate the relation). Similarly, add a fresh node $n_\alpha \in N_\mathcal{A}$ for each attribute $\alpha \in \mathcal{A}$ defined on entity E or relation R , and add an undirected edge from n_α to n, where $n \in N_\mathcal{E} \cup N_\mathcal{R}$ is the node in corresponding to E resp. R.

This graph $\mathcal{G}$ contains directed as well as undirected edges, removing the undirected edges will result in a directed graph $\widetilde{\mathcal{G}}$.

Fig. 4 displays the graph constructed from the ER-model given in Fig. 3, where squares, diamonds, and bullets represent entities, relations, and attributes, resp. The directed subgraph is given in Fig. 5.

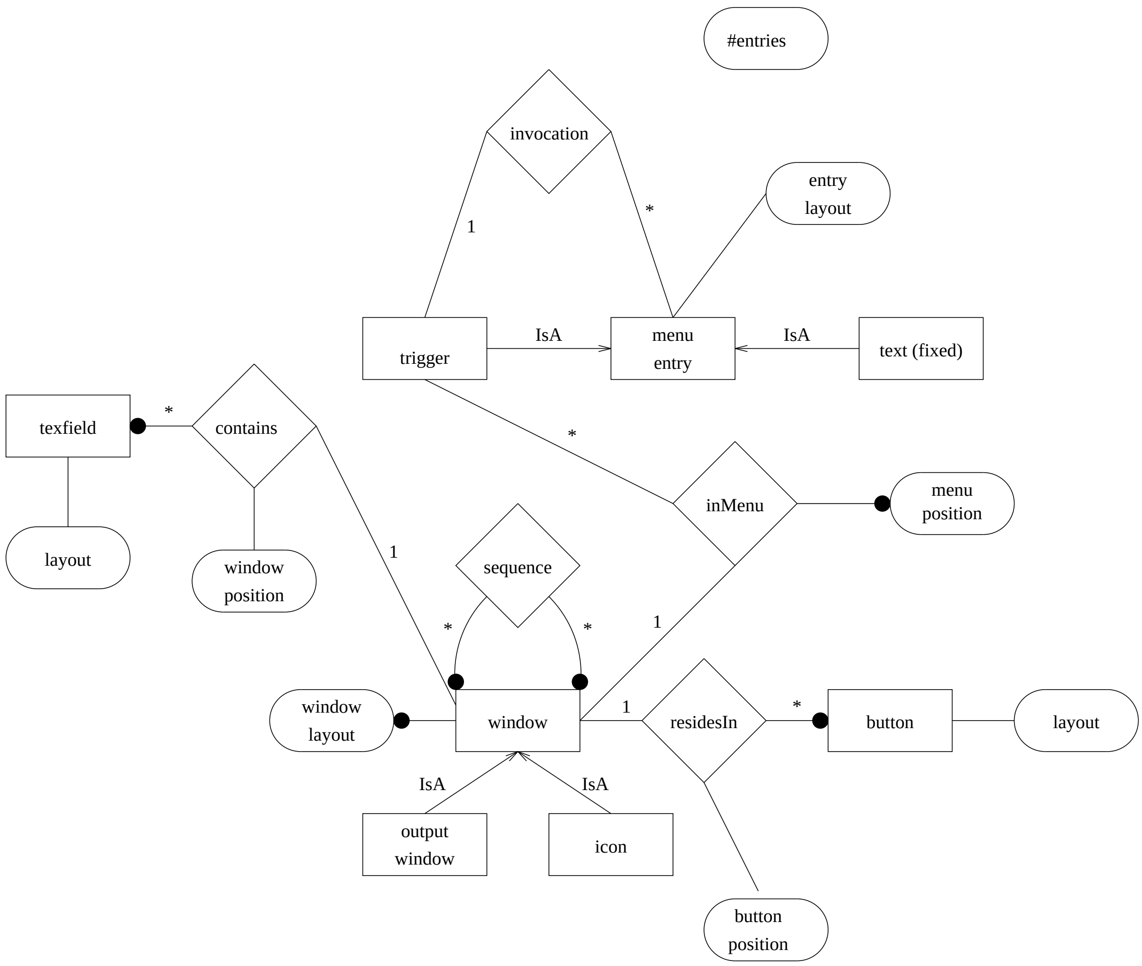

Figure 3: ER-model for a simple graphical user interface

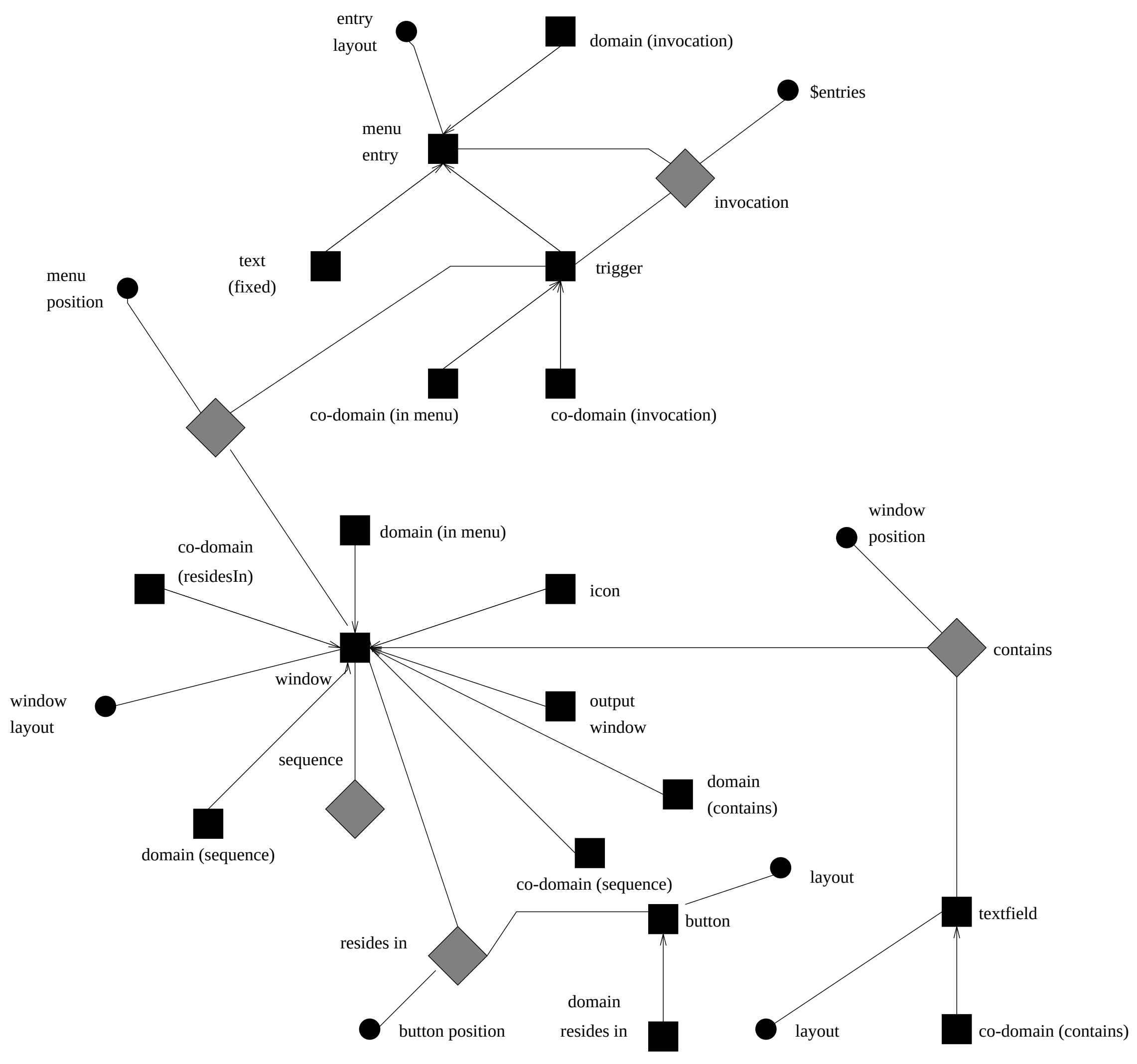

Figure 4: Graph for the ER-model

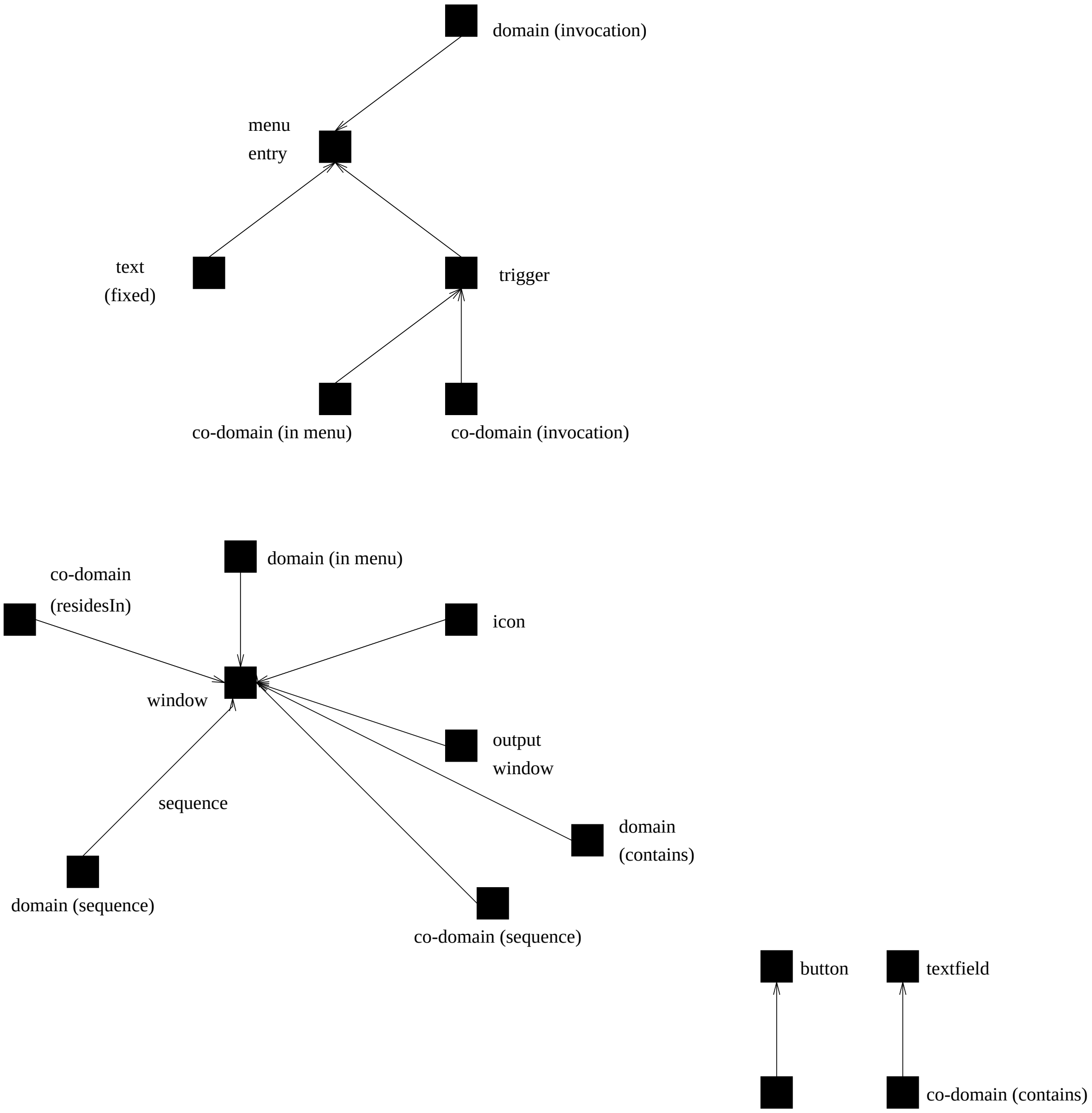

Figure 5: Directed subgraph for the ER-model

4 Constructing the Specification

We will interpret $\mathcal{G}$ as an abstract data type which will be manipulated as a whole, since local operations for an entity or a relation will have side effects on certain other nodes in the graph.

4.1 Basic Constructions

Generate for each root node w in $\widetilde{\mathcal{G}}$ a fresh sort s_w. A path p in $\mathcal{G}$ is called *admissible* iff p is a path in $\widetilde{\mathcal{G}}$ (so p does not contain cycles) such that the first node of p is a leaf or the last node of p is a root in $\widetilde{\mathcal{G}}$. For easier notation, denote by $\upharpoonleft n$ the admissible path in $\widetilde{\mathcal{G}}$ having node n as the first node ending in a root node, and let $\Downarrow n$ be the set of all nodes in $\widetilde{\mathcal{G}}$ which are on an admissible path ending in n.

Let n be a node in $\widetilde{\mathcal{G}}$, hence n comes from an entity. Then there exists a uniquely determined root node $w(n)$ in $\widetilde{\mathcal{G}}$ such that there is an admissible path from n to $w(n)$ (this is so since we do not permit multiple inheritance). Label the node n with sort $s_n := s_{w(n)}$: directed edges come from IsA-relations which correspond to subset-relations in set theory, our model assumes that the instances from entities along an IsA chain all belong to the same set. This implies that all nodes on the path $\upharpoonleft n$ as well as the nodes in $\Downarrow n$ carry the same label.

If R is a relation between entities E and F, label the corresponding node n_R in $\mathcal{G}$ with sort $s_{n_R} := \mathsf{cart}(s_{n_E}, s_{n_F})$. The attribute α for entity E or relation R is modelled as a function symbol $A_\alpha : s^\alpha \to s_\alpha$, with s^α as the sort underlying E resp. R. The corresponding node n_α is labelled with $s_{n_\alpha} := \mathsf{abb}(s^\alpha, s_\alpha)$.

Now put $\sigma_m := \mathsf{set}(s_m)$, if m is a node coming from an entity or from a relation, and set $\sigma_m := s_m$ otherwise (hence m is an attribute node), then set

$$s_{\mathcal{G}} := [\![\sigma_m : m \text{ is a node in } \mathcal{G}]\!] .$$

Define for each node m the projection from $\mathcal{G}$ to m upon renaming

$$proj_m := \pi_{[\![\sigma_m : m \text{ is a node in } \mathcal{G}]\!], \sigma_m}$$

(see A.1 for these constructions).

Intuitively, we have decorated $\mathcal{G}$ so that each node generated from an entity has a set as its label, each relational node is labeled with a subset of the Cartesian product, and each attribute node carries a map. The corresponding operations come from the following specification:

> **specification** $\Pi\langle sigma_m : m \text{ is a node in } \mathcal{G}\rangle$
> **imports** $\{\mathsf{SET}(s_e) : e \text{ is an entity node}\}$
> **imports** $\{\mathsf{SET}(s_r) : r \text{ is a relational node}\}$
> **imports** $\{\mathsf{ABB}(s^\alpha, s_\alpha) : \alpha \text{ is an attribute node with label } \mathsf{abb}(s^\alpha, s_\alpha)\}$
> **end**

This specification will be extended gradually through the following considerations by adding function symbols and axioms.

4.2 Manipulating Entities

If E is an entity in $\mathcal{M}$ with associated node n_E in $\mathcal{G}$, this node is labelled with sort s_{n_E}. So are all nodes on the path $\upharpoonleft n_E$ and in $\Downarrow n_E$. Generate the functions

$$
\begin{array}{rcl}
proj_m(init^E(a)) & = & empty_{s_{n_E}} \text{ for each node } m \in \Downarrow n_E \\
proj_m(init^E(a)) & = & proj_m(a) \text{ for each node } m \notin \Downarrow n_E \\
proj_{n_R}(init^E(a)) & = & empty_{s_{n_R}} \\
 & & \text{for each relational node } n_R \text{ the relation of which has } E \\
 & & \text{as its domain or co-domain (cp. 4.3)} \\
proj_m(insert^E(x,a)) & = & insert_{s_{n_E}}(x, proj_m(a)) \text{ for each node } m \in \uparrow n_E \\
proj_m(insert^E(x,a)) & = & proj_m(a) \text{ for each node } m \notin \uparrow n_E \\
test^E(x,a) & = & \bigwedge\{x \in_{s_{n_E}} proj_m(a) : m \in \uparrow n_E\}
\end{array}
$$

Figure 6: Axioms for insertion operations for entities

$$
\begin{array}{rlcl}
init^E: & s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
insert^E: & s_{n_E} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
test^E: & s_{n_E} \times s_{\mathcal{G}} & \to & \text{BOOL} \\
delete^E: & s_{n_E} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}}
\end{array}
$$

with the axioms given in Fig. 6.

The set of axioms concerning $insert^E$ makes sure that insertion is done along IsA links, and along these links only. The axiom concerning $test^E$ checks along the corresponding path. Note that $init^E$ is not formulated as a constant. This is so since the entire ER-model is formulated as one abstract data type, hence initializing an entity has to take the state of this ADT into account, in particular it must not affect other, unrelated entities or relations.

Deleting an element is a bit more complicated: when an element is deleted from entity E, we have to be sure that it is neither contained in E nor in

- any of the entities F such that F IsA^* E,
- any domain or co-domain of any relation R in which E plays this part; this implies that deletion operations have to be triggered for the corresponding relations,
- any domain of an attribute on E.

We will deal with attributes after modelling operations on relations. Let $delDom^R$ and $delCoDom^R$ be the deletion function for the domain, and for the co-domain of relation R between entities E' and F', having the respective signatures $s_{n_{E'}} \times s_{\mathcal{G}} \to s_{\mathcal{G}}$ and $s_{n_{F'}} \times s_{\mathcal{G}} \to s_{\mathcal{G}}$ (to be more precise, $delDom^R$ is used for modelling the relation `DeleteFromDomain(x, Rel)` which consists of the given relation `Rel` having all pairs in which `x` appears in the first component deleted; similarly for $delCoDom^R$). These function symbols will be considered in 4.3. Once they are provided, deletion may be formulated.

The motivation is as follows: if $m = n_{E'}$ is a node coming from relation E' such that E' IsA^+ E holds, deleting an instance from E implies deletion of that instance from E'. If node m, however, is a node of the form $n = n_{\delta(R)}$, so that E is the domain entity for some relation R, then deletion from E implies deletion from the domain of R and from R itself, so that no pair in R having the instance deleted as the first component survives.

Again note that we are defining the behavior of the function symbols under consideration by their projections onto particular nodes (if only specifying for most nodes that nothing changes). This follows from the observation that the ER-model is formulated as one ADT.

$$\begin{array}{rcl}
proj_m(delete^E(x,a)) & = & delete_{s_{n_E}}(x, proj_m(a)) \text{ for all nodes } m \in \Downarrow n_E \setminus M_r \\
proj_m(delete^E(x,a)) & = & proj_m(delDom^R(x, proj_m(a))) \\
& & \text{for all nodes } m = n_{\delta(R)} \in M_\delta, m = n_R \\
& & \text{for relation } R \\
proj_m(delete^E(x,a)) & = & proj_m(delCoDom^R(x, proj_m(a))) \\
& & \text{for all nodes } m = n_{\gamma(R)} \in M_\gamma, m = n_R \\
& & \text{for relation } R \\
proj_m(delete^E(x,a)) & = & proj_m(a) \text{ for all nodes } m \notin \Downarrow n_E
\end{array}$$

(here

$$\begin{array}{rcl}
M_\delta & := & \{m \in \Downarrow n_E : m = n_{\delta(R)} \text{ for some relation } R\} \\
M_\gamma & := & \{m \in \Downarrow n_E : m = n_{\gamma(R)} \text{ for some relation } R\} \\
M_r & := & M_\delta \cup M_\gamma
\end{array}$$

denote leaves coming from domains or co-domains of relations.)

Figure 7: Axioms for deletion operations from an entity

$$\begin{array}{rcl}
proj_{n_R}(init^R(a)) & = & empty_{s_{n_R}} \\
proj_{n_{\delta(R)}}(init^R(a)) & = & empty_{s_{n_E}} \\
proj_{n_{\gamma(R)}}(init^R(a)) & = & empty_{s_{n_F}} \\
proj_m(init^R(a)) & = & proj_m(a) \text{ for each node } m \notin \{n_R, n_{\delta(R)}, n_{\gamma(R)}\} \\
proj_{n_R}(insert^R(x,y,a)) & = & insert_{s_{n_R}}(pair_{s_{n_E}, s_{n_F}}(x,y), proj_{n_R}(a)) \\
proj_m(insert^R(x,y,a)) & = & insert_{s_{n_E}}(x, proj_m(a)) \\
& & \text{for each node } m \in \uparrow n_{\delta(R)} \\
proj_m(insert^R(x,y,a)) & = & insert_{s_{n_F}}(y, proj_m(a)) \\
& & \text{for each node } m \in \uparrow n_{\gamma(R)} \\
proj_m(insert^R(x,y,a)) & = & proj_m(a) \text{ for each other node } m \text{ in } \mathcal{G} \\
test^R(x,y,a) & = & pair_{s_{n_E}, s_{n_F}}(x,y) \in_{s_{n_R}} proj_{n_R}(a) \wedge \\
& & test^E(x,a) \wedge test^F(y,a)
\end{array}$$

Figure 8: Axioms for the insertion operations for relations

4.3 Manipulating Relations

Let r be a node in $\mathcal{G}$ generated from relation R with domain E and co-domain F, then label n_R with $s_{n_R} := \mathsf{cart}(s_{n_E}, s_{n_F})$, modelling the fact that relations correspond to subsets of Cartesian products.

4.3.1 The General Case

The following function symbols are generated for initialization, insertion, and testing, resp.:

$$\begin{array}{rlcl}
init^R : & s_{\mathcal{G}} & \rightarrow & s_{\mathcal{G}} \\
insert^R : & s_{n_E} \times s_{n_F} \times s_{\mathcal{G}} & \rightarrow & s_{\mathcal{G}} \\
test^R : & s_{n_E} \times s_{n_F} \times s_{\mathcal{G}} & \rightarrow & \text{BOOL}
\end{array}$$

together with the axioms given in Fig. 8. Note that inserting a pair into a relation implies the insertion of the first component and the second one into the domain and the co-domain, resp.

$$
\begin{array}{rcl}
proj_{n_R}(delete^R(x,y,a)) & = & delete_{s_{n_R}}(pair_{s_{n_E},s_{n_F}}(x,y), proj_{n_R}(a)) \\
proj_{n_{\delta(R)}}(delete^R(x,y,a)) & = & \texttt{if } \exists y_1.s_{n_F} : \neg(y = y_1) \wedge test^R(x, y_1, a) \\
 & & \texttt{then} \\
 & & proj_{n_{\delta(R)}}(a) \\
 & & \texttt{else} \\
 & & delete_{s_{n_E}}(x, proj_{n_{\delta(R)}}(a) \\
 & & \texttt{fi} \\
proj_{n_{\gamma(R)}}(delete^R(x,y,a)) & = & \texttt{if } \exists x_1.s_{n_E} : \neg(x = x_1) \wedge test^R(x_1, y, a) \\
 & & \texttt{then} \\
 & & proj_{n_{\gamma(R)}}(a) \\
 & & \texttt{else} \\
 & & delete_{s_{n_F}}(y, proj_{n_{\gamma(R)}}(a)) \\
 & & \texttt{fi}
\end{array}
$$

Figure 9: Axioms for the deletion operations from relations

Again, initialization takes the whole ADT into account, and inserting a pair into a relation must not touch unrelated nodes.

Deletion is a bit more complicated, when it comes to delete from the domain and from the co-domain of R:

$$
\begin{array}{rlcl}
delete^R : & s_{n_E} \times s_{n_F} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
delDom^R : & s_{n_E} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
delCoDom^R : & s_{n_F} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}}
\end{array}
$$

with the axioms given in Fig. 9 and obtained from the patterns in Figs. 10, and 11, resp. The axioms in Fig. 9 specify that deleting a pair from a relation means for the node carrying that relation that the pair has to be removed from the corresponding set of pairs sitting there. The pair's first component has to be deleted from the domain node only if no other pair with the same first component exists in the relation; similarly for the second component. Fig. 10 is instantiated with $M_d := \{n_{\delta(R)}\}$ and describes what happens when all pairs having a given first component are being deleted from a relation. This is essentially described for relations which may be built up through a sequence of insertions and deletions (since only term-generated models are of interest here), and it is described how the values decorating the nodes for the relation proper, its domain, and its co-domain are affected. All other nodes remain untouched. Symmetric considerations apply to instantiating Fig. 11 with $M_c := \{n_{\gamma(R)}\}$ to the analogous situation of deleting all pairs from a relation when the second component is given as an argument.

The axioms in Figs. 10, and 11 are a bit involved, since they are to describe the effect of the domain resp. the co-domain of a relation. Their complexity is due to the fact that these deletions may have remote side effect, e.g., deleting from the domain affects the set decorating the co-domain (and vice versa). Moreover, there are some degrees of freedom through the respective sets M_c and M_d: they specify on which nodes a plain deletion takes place, but these nodes may vary from the constrain for the relation under consideration.

4.3.2 Constraints

Relation R between entities E and F is called *left-total* if each instance of E is related to some instance of F, i.e., iff given an instance e of E there exists an instance f of F such that e is related to f via R; R is called *right-total* iff R^{-1} is left-total, it is called *total* iff it is both left- and right-total.

$$
\begin{array}{rcl}
delDom^R(x, init^R(a)) & = & init^R(a) \\
proj_{n_R}(delDom^R(x, insert^R(x_1, y_1, a))) & = & \texttt{if } x = x_1 \\
 & & \texttt{then} \\
 & & proj_{n_R}(delDom^R(x, a)) \\
 & & \texttt{else} \\
 & & proj_{n_R}(insert^R(x_1, y_1, delDom^R(x, a))) \\
 & & \texttt{fi} \\
proj_{n_R}(delDom^R(x, delete^R(x_1, y_1, a))) & = & \texttt{if } x = x_1 \\
 & & \texttt{then} \\
 & & proj_{n_R}(delDom^R(x, a)) \\
 & & \texttt{else} \\
 & & proj_{n_R}(delete^R(x_1, y_1, delDom^R(x, a))) \\
 & & \texttt{fi} \\
proj_m(delDom^R(x, a)) & = & delete_{s_{n_E}}(x, proj_m(a)) \\
 & & \text{for each node } m \in M_d \\
proj_{n_{\gamma(R)}}(delDom^R(x, insert^R(x_1, y_1, a))) & = & \texttt{if } x = x_1 \\
 & & \texttt{then} \\
 & & proj_{n_{\gamma(R)}}(delDom^R(x, a)) \\
 & & \texttt{else} \\
 & & proj_{n_{\gamma(R)}}(insert^R(x_1, y_1, delDom^R(x, a))) \\
 & & \texttt{fi} \\
proj_{n_{\gamma(R)}}(delDom^R(x, delete^R(x_1, y_1, a))) & = & \texttt{if } x = x_1 \\
 & & \texttt{then} \\
 & & proj_{n_{\gamma(R)}}(delDom^R(x, a)) \\
 & & \texttt{else} \\
 & & proj_{n_{\gamma(R)}}(delete^R(x_1, y_1, delDom^R(x, a))) \\
 & & \texttt{fi} \\
proj_m(delDom^R(x, a)) & = & proj_m(a) \text{ for each node } m \notin \left(\{n_R, n_{\gamma(R)}\} \cup M_d\right)
\end{array}
$$

Figure 10: Axiom patterns for the deletion operations from relations (domain)

Left-Total Relations Now suppose that E has a left-total relation R among the relations having E in its domain with F as the entity for the co-domain for R. Then only the function symbols $test^E$ with the same signature as above is generated, but we do without the function symbols $insert^E$ and $delete^E$, since manipulation of E (initialization, insertion, deletion) is done as a side effect through R. Hence the axioms in Fig. 6 are modified accordingly, the axioms given in Fig. 7 are not needed in this case. Fig. 9 is augmented by the set of axioms

$$
\begin{array}{rcl}
proj_m(delete^R(x, y, a)) & = & \texttt{if } \neg\left(\exists x_1 . s_{n_E} : \neg(x = x_1) \wedge test^R(x_1, y, a)\right) \\
 & & \texttt{then} \\
 & & delete_{s_{n_E}}(x, proj_m(a) \\
 & & \texttt{else} \\
 & & proj_m(a) \\
 & & \texttt{fi} \\
 & & \text{for all nodes } \; m \in \Downarrow n_E
\end{array}
$$

Since the domain of a left-total relation is somewhat tightly coupled to the relation proper, we instantiate the templates from Fig. 10 and Fig. 11 by putting $M_d := \Downarrow n_E$ and $M_c := \Downarrow n_E$, resp.

Right-Total Relations The case that entity F has a relation R which is right-total is now rather symmetric: neither $init^E$, $insert^E$ nor $delete^E$ are generated, the axioms from Fig. 9 are

$$
\begin{array}{rcl}
delCoDom^R(y, init^R(a)) & = & init^R(a) \\
proj_{n_R}(delCoDom^R(y, insert^R(x_1, y_1, a))) & = & \texttt{if } y = y_1 \\
& & \texttt{then} \\
& & proj_{n_R}(delCoDom^R(y, a)) \\
& & \texttt{else} \\
& & proj_{n_R}(insert^R(x_1, y_1, delCoDom^R(y, a))) \\
& & \texttt{fi} \\
proj_{n_R}(delCoDom^R(y, delete^R(x_1, y_1, a))) & = & \texttt{if } y = y_1 \\
& & \texttt{then} \\
& & proj_{n_R}(delCoDom^R(y, a)) \\
& & \texttt{else} \\
& & proj_{n_R}(delete^R(x_1, y_1, delCoDom^R(y, a))) \\
& & \texttt{fi} \\
proj_{n_{\gamma(R)}}(delCoDom^R(y, a)) & = & delete_{s_{n_F}}(y, proj_{n_{\gamma(R)}}(a)) \\
& & \text{for each node } \ m \in M_c \\
proj_{n_{\delta(R)}}(delCoDom^R(y, insert^R(x_1, y_1, a))) & = & \texttt{if } y = y_1 \\
& & \texttt{then} \\
& & proj_{n_{\delta(R)}}(delCoDom^R(y, a)) \\
& & \texttt{else} \\
& & proj_{n_{\delta(R)}}(insert^R(x_1, y_1, delCoDom^R(y, a))) \\
& & \texttt{fi} \\
proj_{n_{\delta(R)}}(delCoDom^R(y, delete^R(x_1, y_1, a))) & = & \texttt{if } y = y_1 \\
& & \texttt{then} \\
& & proj_{n_{\delta(R)}}(delCoDom^R(y, a)) \\
& & \texttt{else} \\
& & proj_{n_{\delta(R)}}(delete^R(x_1, y_1, delCoDom^R(y, a))) \\
& & \texttt{fi} \\
proj_m(delCoDom^R(y, a)) & = & proj_m(a) \text{ for each node } m \notin \big(\{n_R, n_{\gamma(R)}\} \cup M_c\big)
\end{array}
$$

Figure 11: Axiom patterns for the deletion operations from relations (co-domain)

augmented by a very similar set of axioms, viz.,

$$
\begin{array}{rcl}
proj_m(delete^R(x, y, a)) & = & \texttt{if } \neg\big(\exists y_1 . s_{n_E} : \neg(y = y_1) \wedge test^R(x, y_1, a)\big) \\
& & \texttt{then} \\
& & delete_{s_{n_F}}(x, proj_m(a)) \\
& & \texttt{else} \\
& & proj_m(a) \\
& & \texttt{fi} \\
& & \text{for all nodes } \ m \in \Downarrow n_F
\end{array}
$$

and the axiom pattern from Fig. 10 and Fig. 11 by putting $M_d := \Downarrow n_F$ and $M_c := \Downarrow n_F$, resp.

Total relations are treated as the combination of the cases considered so far and need not be discussed further.

N:1 Relations Let R be an $N:1$-relation for the entities E and F; we do not assume R to be a left- or right-total relation (the modifications are obvious). All function symbols dealing with insertion into R have to be modified according to the following pattern

$$
\begin{array}{rcl}
proj_{n_\alpha}(InitAtt_\alpha(a)) & = & init_{s_{n_E},s_\alpha} \\
proj_m(InitAtt_\alpha(a)) & = & proj_m(a) \text{ for each node } \ m \neq n_\alpha \\
proj_{n_\alpha}(PutAtt_\alpha(x,y,a)) & = & put_{s_{n_E},s_\alpha}(x,y,A_\alpha) \\
proj_m(PutAtt_\alpha(x,y,a)) & = & put_{s_{n_E},s_\alpha}(x,a) \text{ for each node } \ m \in \uparrow n_E \\
proj_m(PutAtt_\alpha(x,y,a)) & = & proj_m(a) \text{ for each node } \ m \notin \uparrow n_E \cup \{n_\alpha\} \\
GetAtt_\alpha(x,a) & = & get_{s_{n_E},s_\alpha}(x, proj_{n_\alpha}(a)) \\
proj_{n_\alpha}(UnPutAtt_\alpha(x,a)) & = & unput_{s_{n_E},s_\alpha}(x, proj_{n_\alpha}(a)) \\
proj_m(UnPutAtt_\alpha(x,a)) & = & proj_m(a) \text{ for each node } \ m \neq n_\alpha
\end{array}
$$

Figure 12: Manipulating attributes (general case)

$$
\begin{array}{rcl}
proj_{n_R}(insert^R(x,y,a)) & = & \texttt{if } \neg\left(\exists x_1.s_{n_E} : test^R(x_1,y,a) \wedge \neg(x = x_1)\right) \\
& & \texttt{then} \\
& & insert_{s_{n_R}}(pair_{s_{n_E},s_{n_F}}(x,y), proj_{n_R}(a)) \\
& & \texttt{else} \\
& & error_{n_R} \\
& & \texttt{fi}
\end{array}
$$

The conditions guard the insertion from inserting a pair the second component of which is already related to another instance.

4.4 Specifying Attributes

We discuss only the case of attributes defined on entities; attributes defined on relations are delt with *cum grano salis* in a similar way.

4.4.1 The General Case

Let α be defined on entity E such that the corresponding function symbol A_α has the signature $s_{n_E} \to s_\alpha$. The following function symbols are defined

$$
\begin{array}{rlcl}
InitAtt_\alpha : & s_\mathcal{G} & \to & s_\mathcal{G} \\
PutAtt_\alpha : & s_{n_E} \times s_\alpha \times s_\mathcal{G} & \to & s_\mathcal{G} \\
GetAtt_\alpha : & s_{n_E} \times s_\mathcal{G} & \to & s_\alpha \\
UnPutAtt_\alpha : & s_{n_E} \times s_\mathcal{G} & \to & s_\mathcal{G}
\end{array}
$$

Informally stated, $InitAtt_\alpha(a)$ initializes the attribute, $PutAtt_\alpha(x,y,a)$ sets the attribute value for the instance x to y, $GetAtt_\alpha(x,a)$ retrieves the value for the instance x, and finally $UnPutAtt_\alpha(x,a)$ removes the value for x from the attribute. Note that we always take the global state of the ADT into account.

The axioms are given in Fig. 12; they are essentially an adaption of the axioms for $\mathsf{abb}(s_{n_E}, s_\alpha)$ to the graph $\mathcal{G}$.

4.4.2 Special Cases: Total Attributes

Let $Total_E$ and $NonTotal_E$ be the set of all total resp. non-total attributes defined on E. Hence $\alpha(e)$ is defined for each instance e of E and each $\alpha \in Total_E$. Then removing the value for an attribute implies removing the corresponding instance, inserting an instance into E requires defining the values for all the attributes in $Total_E$. Thus the situation is similar to N:1-relations.

In fact, if $Total_E \neq \emptyset$, we need not generate the function symbols $init^E$, $insert^E$ and $delete^E$, since initializing, inserting and deleting is done as a side effect through the corresponding operations for

$$
\begin{array}{rcl}
proj_{n_{\alpha_i}}(InitTotAtt^E(a)) & = & init_{s_{n_E}, s_{\alpha_i}} \text{ for } i = 1, \ldots, k \\
proj_m(InitTotAtt^E(a)) & = & \texttt{if } empty_{s_{n_E}}(proj_{n_E}(a)) \\
 & & \texttt{then} \\
 & & empty_{s_{n_E}} \\
 & & \texttt{else} \\
 & & error_{s_{n_E}} \\
 & & \texttt{fi} \\
 & & \text{for all nodes } m \in \Downarrow n_E \\
proj_{n_R}(InitTotAtt^E(a)) & = & \texttt{if } empty_{s_{n_E}}(proj_{n_E}(a)) \\
 & & \texttt{then} \\
 & & empty_{s_{n_R}} \\
 & & \texttt{else} \\
 & & error_{s_{n_R}} \\
 & & \texttt{fi} \\
 & & \text{for each relational node } n_R \\
 & & \text{such that } R \text{ has } E \text{ as its domain or co-domain} \\
proj_m(InitTotAtt_\alpha(a)) & = & proj_m(a) \text{ for each other node } m \\
proj_{n_{\alpha_i}}(PutTotAtt^E(x, y_1, \ldots, y_k, a)) & = & put_{s_{n_E}, s_{\alpha_i}}(x, y_i, A_{\alpha_i}) \\
 & & \text{for } i = 1, \ldots, k \\
proj_m(PutTotAtt^E(x, y_1, \ldots, y_k, a)) & = & proj_m(insert^E(x, a)) \\
 & & \text{for all nodes } m \in \uparrow n_E \\
proj_m(PutTotAtt^E(x, y_1, \ldots, y_k, a)) & = & proj_m(a) \text{ for each node} \\
 & & m \notin \uparrow n_E \cup \{n_{\alpha_1}, \ldots, n_{\alpha_k}\} \\
proj_{n_{\alpha_i}}(UnPutTotAtt^E(x, a)) & = & unput_{s_{n_E}, s_{\alpha_i}}(x, proj_{n_{\alpha_i}}(a)) \\
 & & \text{for } i = 1, \ldots, k \\
proj_m(UnPutTotAtt^E(x, a)) & = & proj_m(delete^E(x, a)) \\
 & & \text{for all nodes } m \in \Downarrow n_E \\
proj_m(UnPutTotAtt^E(x, a)) & = & proj_m(a) \text{ for each node} \\
 & & m \notin \Downarrow n_E \cup \{n_{\alpha_1}, \ldots, n_{\alpha_k}\}
\end{array}
$$

Figure 13: Modified axioms for total attributes on entity E

---

the attributes in $Total_E$; $test^E$ is generated, however. Let $Total_E = \{\alpha_1, \ldots, \alpha_k\}$ and generate the function symbols $InitAtt_\alpha$, $PutAtt_\alpha$, $UnPutAtt_\alpha$ for each attribute $\alpha \in NonTotal_E$, and $GetAtt_\alpha$ for each attribute α defined on E. Instead of generating the missing function symbols individually for each $\alpha \in Total_E$, we do it for the whole collection:

$$
\begin{array}{rlcl}
InitTotAtt^E: & s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
PutTotAtt^E: & s_{n_E} \times s_{\alpha_1} \times \ldots s_{\alpha_k} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}} \\
UnPutTotAtt^E: & s_{n_E} \times s_{\mathcal{G}} & \to & s_{\mathcal{G}}
\end{array}
$$

The axioms for this case are given in Fig. 13.

4.4.3 Special Cases: Keys

Let α be a key attribute on entity E, then α corresponds to a total injective map. Assume that $Total_E = Key_E \cup NonKey_E$ is partitioned into key and non-key attributes,

$$
\begin{array}{rcl}
Key_E & = & \{\alpha_1, \ldots, \alpha_\ell\}, \\
NonKey_E & = & \{\alpha_{\ell+1}, \ldots, \alpha_k\},
\end{array}
$$

and assume that $\ell > 0$ holds. Hence each element in Key_E is a key attribute on its own. The axioms for the function symbol $PutTotAtt^E$ given in Fig. 13 are modified so that a value for a key attribute is set only if it did not occur before, hence injectivity is preserved. To be more specific,

$$proj_m(PutTotAtt^E(x, y_1, \ldots, y_k, a))$$

is guarded for each node m by the condition

$$\bigwedge_{j=1}^{j=\ell} \neg \left(\exists z_j.s_{\alpha_j} : GetAtt_{\alpha_j}(z_j, proj_{n_{\alpha_j}}(a)) = y_j \right).$$

If the condition is true, the insertion indicated in Fig. 13 is returned, and an *error* value otherwise. The obvious details are left to the reader.

5 Application to Petri Nets

The ER-model $\mathcal{M}$ generates an algebraic specification $\langle \Sigma_{\mathcal{M}}, \Delta_{\mathcal{M}} \rangle$ according to the constructions outlined above. We indicate how this mechanism may be put to use in the context of Petri nets by discussing the general situation in which valid formulas from the algebraic specification are used as conditions, and labels for flows, resp.

It is well known that Petri nets may be used for the functional specification of concurrent systems (cp. e.g. [11, 5.5.3]), and that the conceptual description of data and their relations may in many cases be formulated using ER-diagrams. These specifications are usually done separately. Information systems, however, require the joint description of functional properties and conceptal properties of data, hence a formalism specifying both the functional and the data view could make use of a tight coupling of Petri nets and ER-diagrams. In this section we propose such a marriage through condition-event nets [18, 13]. The following definition is an adaptation of Reisig's definition of this class of nets, see [18, Sec. 8.2].

Formally, a condition-event net consists of an underlying bipartite graph (P, T, Φ), where P is the set of *places*, T is the set of *transitions*, and $\Phi \subset (P \times T) \cup (T \times P)$ is the flow relation. As usual, put for $a \in P \cup T$

$$\begin{aligned} {}^{\bullet}a &:= \{b \in P \cup T : (b, a) \in \Phi\} \\ a^{\bullet} &:= \{b \in P \cup T : (a, b) \in \Phi\} \end{aligned}$$

Now let $\langle \Sigma, \Delta \rangle$ be an algebraic specification with the S-sorted set X of free variables, where $\Sigma = \langle S, \Gamma \rangle$. Fix a model $\mathcal{A} = (A_s)_{s \in S}$ for the specification, and assume that

$$\begin{aligned} \lambda &: \Phi \to \mathrm{wff}(\Sigma, X) \\ c &: P \to \mathrm{wff}(\Sigma, X) \end{aligned}$$

are maps such that for each flow φ and each place p all terms in $\lambda(\varphi)$ and in $c(p)$ are valid in $\mathcal{A}$. c is called a *condition.* A transition $t \in T$ is *c-activated* iff for each $s \in S$ the following holds:

$$\begin{aligned} \forall p \in {}^{\bullet}t : \lambda_s(p, t) &\subseteq c_s(p) \\ \forall p \in t^{\bullet} : \lambda_s(t, p) &\cap c_s(p) = \emptyset \end{aligned}$$

Thus the condition imposed by c holds for each label on the flow (p, t), and no label on (t, p) satisfies the condition. In this case, the transition may fire, and a new condition c' is defined upon setting

$$c'_s(p) := \begin{cases} c_s(p) \setminus \lambda_s(p, t), & p \in {}^{\bullet}t \setminus t^{\bullet} \\ c_s(p) \cup \lambda_s(t, p), & p \in t^{\bullet} \setminus {}^{\bullet}t \\ (c_s(p) \setminus \lambda_s(p, t)) \cup \lambda_s(t, p), & p \in {}^{\bullet}t \cap t^{\bullet} \\ c_s(p) & otherwise \end{cases}$$

The formalism outlined here permits the joint modelling of functions and data: functional modelling may be done through a Petri net, the flows of which are annotated with formulas coming from the model for an algebraic specification of an ER-model, where free variables are interpreted in a specific way. Places are marked with conditions pertaining to the model.

6 Implementation Issues

The ER-specification given here is based on specifications related to set theory. This makes constructing a model rather straightforward: suppose $\mathcal{M}_s, \mathcal{C}_{s_1,s_2}$, and $\mathcal{F}_{s_1,s_2}$ are models for the respective specifications $\mathsf{SET}(s)$, $\mathsf{CART}(s_1, s_2)$, and $\mathsf{ABB}(s_1, s_2)$ (cp. 2.1). Then label each node in $\mathcal{G}$ correspondingly: a node generated for an entity which is assigned sort s_{n_E} obtains the label $\mathcal{M}_{s_{n_E}}$, a node for a relation which is assigned sort $\mathsf{cart}(s_{n_E}, s_{n_F})$ is labeled with $\mathcal{C}_{s_{n_E},s_{n_F}}$, similarly for attribute nodes: a label $\mathsf{abb}(s^\alpha, s_\alpha)$ is interpreted $\mathcal{F}_{s^\alpha,s_\alpha}$. The operations carry over in a natural way, e.g. $insert^E$ is interpreted for an entity E as set insertion in $\mathcal{M}_{s_{n_E}}$ along the admissible path from n_E to the corresponding root node, and as the identity on all nodes of $\mathcal{G}$ outside $\uparrow n$. Consequently, the operations on $\mathcal{M}_{s_{n_E}}$ may be extended in a natural way to operations on the decorated graph, similar extensions hold for the operations on $\mathcal{C}_{s_{n_E},s_{n_F}}$ and on $\mathcal{F}_{s^\alpha,s_\alpha}$. The discussion above disregards e.g. constraints on relations or attributes, taking them into account will make notation clumsier but does not change the argumentation. So we lock them out.

It is plain that this *procedere* induces an interpretation, and that the decorated graph is in fact a model for the specification. Moreover, if $\mathcal{M}_s, \mathcal{C}_{s_1,s_2}$, and $\mathcal{C}_{s_1,s_2}$ are all chosen to be initial models, an initial model is obtained, similarly, selecting only terminal models for the basic specifications translates into a terminal model. These considerations provide initial and terminal semantics for the ER-specification, loose semantics is obtained from selecting arbitrary (reachable) models for the basic specifications. Thus we have provided a formal semantics for the class of ER-models considered in this paper.

We have implemented a generator for the specification, which takes a textual description for an ER-model as an input. The results are somewhat discouraging: the specification generated for the simple user interface from Fig. 3 included 14 specifications to be imported, 83 function symbols, and 2232 axioms, all this on top of the basic parametrized specifications. Most of the axioms are identity axioms, stating that nothing changes on most nodes in $\mathcal{G}$; omitting these axioms, 220 axioms are left, which is a considerable reduction, but still too large to be maintained without a supporting tool. Thus the proposal put forth in section 5 of using terms coming from an ER-specification as labels for Petri nets should be complemented by a suitable tool assisting in manipulating the particulars of the specification (Petri nets for realistic applications tend to be large, hence tool support is mandatory anyway).

The model theoretic observations make it easy to implement the ER-specification, at least in a language like SETL ([19, 8]) or ProSet [9] supporting directly the dictions of finite set theory (in fact, a package implementing set operations would be sufficient, see the experiment reported about in [7]). Taking these operations as a basic layer, the operations for the ER-specifications are translated directly into expressions of the programming language. The graph $\mathcal{G}$ is not constructed explicitly but may rather be maintained implicitly through the flow of control (e.g. by grouping all the insert statements which are generated for $insert^E$ along the nodes in $\uparrow n_E$ into one insertion procedure and omitting the identity axioms altogether). This reduces the size of the implementation considerably, and a generator may easily be derived from the one indicated above. Thus it is possible to adequately model data using ER-models within the context of software prototyping with persistent data [6, 10], one of the goals of the ProSet project. An ER-model translates into a module having the state of the current instances for the entities, relations, and attributes as its internal state and exporting all the operations on the model generated from the ER-specification, so that the ER-model is manipulated entirely through this module. Having first class citizen rights, modules may be made persistent, thus we end up with a persistent implementation of the ER-model.

7 Related Work and Further Study

The problem of formally describing ER-models has been undertaken by severals authors with different kinds of ER-models and different motivations in mind.

Hettler [14] focusses on a formulation in the specification language SPECTRUM, modelling entities as records with attributes as entries. Dependencies between entities via IsA are not formulated using a graph or a comparable structure. It is noted that these dependencies have to be taken care of, and an example shows how to do it. The motivation for this work is to demonstrate that this data modelling technique may be made accessible in SPECTRUM.

The report [3] proceeds in two steps: transformation of the ER-diagram into an attributed graph signature and transformation of the integrity constraints into first-order logic formulas. The ER-model used does not take the IsA relation explicitly into account. The paper has its focus not on providing a static semantics but rather formalizing the dynamic aspect — transactions — through Σ-homomorphisms, hence showing how transactions may be caught in an algebraic framework.

Gogolla & Hohenstein [12] and a bit later Hohenstein [15] deal with the formal semantics of an extended ER-model form a data base point of view. The goal is to provide a mathematical semantics of EER-models and to propose a calculus for their manipulation ("a well founded calculus taking into account data operations on arbitrary user defined types and aggregate functions“ is the formulation in [12]). These authors propose the semantics of a data base signature as the set of all interpretations ([15], p. 63) and work within that framework; algebraic specifications are not used explicitely.

Further Study The basic ER-model may be extended to support semantic data modelling techniques [16]. This would be done along the lines provided here by first adding some primitive operations to the specification $\mathsf{SET}(s)$ (e.g., the disjoint union has to be formulated), then extending the specification in section 4 correspondingly. Graphical support through an ER-editor would make working with the machinery proposed here more pleasant; work on such an editor is under way. Combining the algebraic specification with functional specifications using Petri nets as suggested in section 5 should be supported by a visually oriented tool. In a similar way, the problems arising in interfacing an ER-editor with a graphical editor e.g. for the specification language π (cp. [1]) do not appear to be entirely trivial.

It should be possible to automatically derive an implementation in a procedural or object-oriented language for the specification of an ER-model, given the results reported e.g. by Lin [17]. It is plain that what we called above *identity axioms* will serve as pre- and as postconditions, and that most of the other axioms may be read from left to right, yielding procedure calls. Augmenting the model may change the picture, and investigating such an automatic derivation may be interesting.

ER-models tend to become large and unmanageable, so they are modularized. We deal here with flat ER-models. A modular ER-specification should of course reflect the modular structure of the underlying model, so adequate techniques for modularizing algebraic specifications are needed, cp. [22, 9.3,9.2].

A Appendix

This appendix provides specifications for the Cartesian product, and for the associative maps of set theory.

$$\begin{array}{rcl}
\pi_{[\![s_1,\ldots,s_k]\!],s_i}\left(tup_{[\![s_1,\ldots,s_k]\!]}(x_1,\ldots,x_k)\right) & =_{s_i} & x_i \\
 & & \text{for } i=1,\ldots,k \\
z_1 =_{[\![s_1,\ldots,s_k]\!]} z_2 & =_{\text{BOOL}} & \bigwedge_{i=1}^{i=k}\left(\pi_{[\![s_1,\ldots,s_k]\!],s_i(z_1)} =_{s_i} \pi_{[\![s_1,\ldots,s_k]\!],s_i(z_2)}\right) \\
z & =_{[\![s_1,\ldots,s_k]\!]} & tup_{[\![s_1,\ldots,s_k]\!]}(\pi_{[\![s_1,\ldots,s_k]\!],s_1(z)},\ldots\pi_{[\![s_1,\ldots,s_k]\!],s_k(z)})
\end{array}$$

Figure 14: Axioms for the Cartesian product

A.1 The Cartesian Product

Let $s_1,\ldots,s_k$ be sorts, with $. =_{s_i} .$ as the equality relation on s_i for $i = 1,\ldots,k$. The sorts need not be different. Generate a fresh sort $[\![s_1,\ldots,s_k]\!]$, and put

$$S := \{s_1,\ldots,s_k,[\![s_1,\ldots,s_k]\!]\}$$

as the signature for the Cartesian product. The set Γ of function symbols consists of

$$\begin{array}{rlcl}
tup_{[\![s_1,\ldots,s_k]\!]}: & s_1\times\ldots\times s_k & \to & [\![s_1,\ldots,s_k]\!] \\
\pi_{[\![s_1,\ldots,s_k]\!],s_i}: & [\![s_1,\ldots,s_k]\!] & \to & s_i \\
 & & & \text{for } i=1,\ldots,k
\end{array}$$

Intuitively, the *tup*-function builds a tuple from its argument, and the π-functions are the corresponding projections. The set Γ of axioms (which are not really surprising) is enumerated in Fig. 14. The specification

$$\Pi\langle s_1,\ldots,s_k\rangle := \langle\langle S,\Delta\rangle,\Gamma\rangle$$

denotes then the specification for the Cartesian product. If we have only two sorts s_1, s_2, the rather clumsy notation may be alleviated somewhat: the specification is then denoted by $\mathsf{CART}(s_1,s_2)$, the tuple function and the projections are renamed to $pair_{s_1,s_2}$ and π_{s_1},π_{s_2}, resp., and the new sort $[\![s_1,s_2]\!]$ is denoted by $\mathsf{cart}(s_1,s_2)$ subsectionAssociative Maps Given the sorts s_1, s_2 with $. =_{s_1} ., . =_{s_2} .$ as the corresponding equality relations, we generate a fresh sort $\mathsf{abb}(s_1,s_2)$ and the set Γ function symbols

$$\Gamma := \{init_{s_1,s_2}, put_{s_1,s_2}, get_{s_1,s_2}, unput_{s_1,s_2}, undef_{s_1,s_2}\}$$

with these signatures:

$$\begin{array}{rlcl}
init_{s_1,s_2}: & & \to & \mathsf{abb}(s_1,s_2) \\
put_{s_1,s_2}: & s_1\times s_2\times\mathsf{abb}(s_1,s_2) & \to & \mathsf{abb}(s_1,s_2) \\
get_{s_1,s_2}: & s_1\times\mathsf{abb}(s_1,s_2) & \to & s_2 \\
unput_{s_1,s_2}: & s_1\times\mathsf{abb}(s_1,s_2) & \to & \mathsf{abb}(s_1,s_2) \\
undef_{s_1,s_2}: & & \to & s_2
\end{array}$$

Intuitively, the initialization of a partial map happens through $init_{s_1,s_2}$ which produces a map that is undefined everywhere, i.e. that has the value $undef_{s_1,s_2}$ for each argument. Setting an argument to a value happens through put_{s_1,s_2}: the function takes the argument x, the value y and the map f, producing a new map which behaves exactly as f does, except that it assigns the value y to x. The function get_{s_1,s_2} retrieves the function value, and $unput_{s_1,s_2}$ reverses the effect of put_{s_1,s_2}.

The axioms are given in Fig. 15.

References

[1] S. Alker. Ein hybrider graphischer Syntaxeditor für Konfigurationsspezifikationen in der π-Sprache. Master's Thesis, University of Dortmund, Chair for Software Technology, 1993.

$$
\begin{array}{rll}
get_{s_1,s_2}(x, init_{s_1,s_2}) & =_{s_2} & undef_{s_1,s_2} \\
get_{s_1,s_2}(x, put_{s_1,s_2}(x,y,f)) & =_{s_2} & y \\
get_{s_1,s_2}(x, unput_{s_1,s_2}(x,f)) & =_{s_2} & undef_{s_1,s_2} \\
get_{s_1,s_2}(x', put_{s_1,s_2}(x,y,init_{s_1,s_2})) & =_{s_2} & \texttt{if } x' =_{s_1} x \texttt{ then } y \texttt{ else } undef_{s_1,s_2} \texttt{ fi} \\
unput_{s_1,s_2}(x, put_{s_1,s_2}(x,y,f)) & =_{\mathsf{abb}(s_1,s_2)} & unput_{s_1,s_2}(x,f) \\
unput_{s_1,s_2}(x, init_{s_1,s_2})) & =_{\mathsf{abb}(s_1,s_2)} & init_{s_1,s_2} \\
put_{s_1,s_2}(x,y, put_{s_1,s_2}(x,y',f)) & =_{\mathsf{abb}(s_1,s_2)} & put_{s_1,s_2}(x,y,f) \\
\left(f =_{\mathsf{abb}(s_1,s_2)} g\right) & =_{\text{BOOL}} & \forall x.s_1 : (get_{s_1,s_2}(x,f) =_{s_2} get_{s_1,s_2}(x,g)) \wedge \\
 & & \left(unput_{s_1,s_2}(x,f) =_{\mathsf{abb}(s_1,s_2)} unput_{s_1,s_2}(x,g)\right)
\end{array}
$$

Figure 15: Axioms for operations on partial maps

---

[2] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, Slotosch O., and K. Stølen. The requirements and design specification language SPECTRUM — an informal introduction. Technical Report `TUM I9311 - 12`, Technische Universität München, 1993.

[3] I. Claßen, M. Löwe, S. Waßerroth, and J. Wortmann. Static and dynamic semantics of Entity–Relationship models based on algebraic methods. Technical report, Technische Universität Berlin, Fachbereich Informatik, 1994.

[4] J. Cramer. *Interconnecting and Reusing Component Specifications.* PhD thesis, University of Dortmund, Chair for Software Technology, November 1994.

[5] J. Cramer, W. Fey, M. Goedicke, and M. Große-Rhode. Towards a formally based component description language - a foundation for reuse. *Structured Programming*, 12:91 – 110, 1991.

[6] E.-E. Doberkat. Integrating persistence into a set-oriented prototyping language. *Structured Programming*, 13:137 – 153, 1992.

[7] E.-E. Doberkat, E. Dubinsky, and J.T. Schwartz. Reusability of design for complex programs: an experiment with the SETL optimizer. In *Proc. IT & T Workshop on Reusability of Software*, pages 106 — 108, Providence, R.I., 1983.

[8] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL.* Teubner – Verlag, 1989, Stuttgart.

[9] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — A Language for Prototyping with Sets. In N. Kanopoulos, editor, *Proc. Third International Workshop on Rapid System Prototyping*, pages 235–248, Research Triangle Park, NC, June 1992. IEEE Computer Society Press.

[10] E.-E. Doberkat, W. Franke, U. Kelter, and W. Seelbach. Verwaltung persistenter Daten in einer Prototyping–Umgebung. In H. Zuellighoven, W. Altmann, and E.-E. Doberkat, editors, *Requirements Engineering '93: Prototyping*, pages 147 – 164. Teubner-Verlag, Stuttgart, 1993.

[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, 1991.

[12] M. Gogolla and U. Hohenstein. Towards a semantic view of an extended Entity–Relationship model. *ACM Transactions on Database Systems*, 16:369 – 416, 1991.

[13] X. He and J. A. N. Lee. A methodology for constructing predicate transition net specifications. *Software - Practice and Experience*, 21:845–875, August 1991.

[14] R. Hettler. Zur Übersetzung von E/R–Schemata nach SPECTRUM. Technical Report `TUM I9333`, Technische Universität München, 1993.

[15] U. Hohenstein. *Formale Semantik eines erweiterten Entity-Relationship-Modells.* B. G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig, 1993.

[16] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19:201 – 261, 1987.

[17] H. Lin. Procedural implementation of algebraic specifications. *ACM Trans. Prog. Lang. Syst.*, 15:876 – 895, 1993.

[18] W. Reisig. *Petrinetze.* Springer–Verlag, Berlin, Heidelberg, New York, 1982.

[19] J.T. Schwartz, J. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets: An Introduction To SETL.* Springer – Verlag, 1986, New York.

[20] J. D. Ullman. *Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville, MD, 1988.

[21] M. Wirsing. Structures algebraic specifications: a kernel language. *Theoretical Computer Science*, 43:123 – 250, 1986.

[22] M. Wirsing. Algebraic specifications. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B: Formal Models and Semantics*, pages 675 – 788. Elsevier, Amsterdam, 1990.