PUC

ISSN 0103-9741

Monografias em Ciência da Computação
nº 15/95

Towards a Logical Analysis of Design: Formal Software Engineering and Data Base Concepts

Paulo A. S. Veloso
Antonio L. Furtado

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 15/95

Editor: Carlos J. P. Lucena June, 1995

Towards a Logical Analysis of Design: Formal Software Engineering and Data Base Concepts *

Paulo A. S. Veloso

Antonio L. Furtado

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:

Rosane Teles Lins Castilho

Assessoria de Biblioteca, Documentação e Informação

PUC Rio — Departamento de Informática

Rua Marquês de São Vicente, 225 — Gávea

22453-900 — Rio de Janeiro, RJ

Brasil

Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645

E-mail: rosane@inf.puc-rio.br

TOWARDS A LOGICAL ANALYSIS OF DESIGN:
Formal Software Engineering and Data Base Concepts

Paulo A. S. VELOSO and **Antonio L. FURTADO**
e-mail {veloso, furtado}@inf.puc-rio.br

PUCRioInf MCC 15/95

Abstract. We analyze the various objects and arrows involved in conceptual design of data base applications, aiming at clarifying the roles played by the diverse language dichotomies: application vs. data model, static vs. dynamic, etc. This analysis is suggested by the similarity with encapsulation and implementation of data types in Software Engineering, which in turn leads to interpretation of specifications based on translation of their languages. The central ideas come from the logical approach to formal specifications. We emphasize the process of representing an application concept on a data model.

Key words: Data base architecture, conceptual design, data models, application concepts, external schemas, internal schemas, software engineering, abstract data types, formal specifications, interpretation, representation, translation.

Resumo. Analisam-se os vários objetos e setas involvidos no projeto conceitual de aplicações da bancos de dados, a fim de esclarecer os papéis desempenhados pelas diversas dicotomias de linguagens: aplicação vs. modelo de dados, estático vs. dinâmico, etc. Esta análise é sugerida pela similaridade com encapsulamento e implementação de tipos dados em Engenharia de Software, o que por sua vez conduz a interpretações de especificações com base em traduções de suas linguagens. As idéias centrais têm sua origem no enfoque lógico para especificações formais. Dá-se ênfase ao processo de representação de um conceito de aplicação em um modelo de dados.

Palavras chave: Arquitetura de bancos de dados, projeto conceitual, conceitos de aplicação, esquemas externos, esquemas internos, engenharia de software, tipos abstratos de dados, especificações formais, interpretação, representação, tradução.

CONTENTS

1. Introduction

Data base design involves several objects, such as external schemas and data models, as well as arrows between them, such as representation of application concept on data model. In this paper we use some formal ideas from Software Engineering to put them in perspective: properly regarded all such objects have the same nature, and likewise for the intervening arrows. We aim at clarifying the roles played by the diverse language dichotomies: application vs. data model, static vs. dynamic, etc.

More specifically, we consider the ANSI-X3-SPARC architecture and regard external schemas, application concepts, data models, and internal schemas as abstract data types. Then, mappings between them, such as user interfaces and representations on data models, are naturally seen as interpretations between the associated specifications.

The situation is akin to the one in Software Engineering, where an implementation of an abstract data type on a more concrete one amounts to a translation of the abstract concepts into a cluster-like module over the concrete data type [Guttag '77; Liskov & Zilles '77].

The central ideas come from the logical approach to specifications [Turski & Maibaum '87; Veloso '87], which in turn relies on the simple logical concepts of extensions and interpretations of axiomatic theories [Enderton '72].

2. Data Base and Software Engineering

The ANSI-X3-SPARC architecture suggests regarding analysis and design of data base applications on three levels: external, conceptual and physical. Important guidelines in this proposal are the notions of independence, protection and integrity. To achieve them the idea of data models appears naturally as an intermediate step between application concepts and physical files.

A data model supports concepts purportedly of widespread applicability. Lists and trees are data structures that are applicable in various situations; so their properties have been carefully analyzed and efficient implementations for them are available. Data models may be seen as candidates to a similar role. An application deals with specific objects and properties, whereas a data model is supposed to handle less specific ones. In this manner a general data model may support a variety of applications.

Independence, protection and integrity are important goals in Software Engineering as well. Some conceptual tools towards these aims, such as information hiding and encapsulation, have been put forward [Gehani & McGettrick '86; Liskov & Zilles '77]. These ideas lead naturally to abstract data types and formal specifications. An abstract data type has an encapsulating signature providing access paths to its objects, whose internal representation is hidden. To connect diverse data types some translation between their signatures is required [Turski & Maibaum '87; Veloso '87; Veloso & Maibaum '95].

For definiteness, we shall focus on representation of an application on a data model. The analysis, however, will naturally carry over to user interfaces for external schemas on an application concept, as well as to implementation of a data model by physical files, if properly regarded. We shall also comment on some methodological issues.

3. Informal Presentation and Overview

For the sake of concreteness, we shall center our presentation around an, admittedly simple, example of representation of an application on a data model. The theoretical considerations, however, will be seen to be general. Also, some methodological aspects can be adapted to other levels.

3.1 Example: Informal Presentation

We shall consider as a running example a representation of a simple academic data base on a version of the entity-relationship-attribute data model.

Application: academic data base

As a simple-minded example consider an academic data base with information about students and courses during an academic term. This information concerns regular students, i. e. registered, courses listed as offered, and courses taken by students by enrolling. Also, students may drop courses and courses may be canceled as long as regulations are not violated.

These university regulations are: students can take only courses that are currently listed and an active student must remain active (during the term), where active refers to students taking at least one course. These regulations impose some constraints on the possible evolution of the states. We see that the former constraint refers to a state of the data base, thus being called static, whereas the latter involves its present and future states, thus being dynamic.

Each student is identified by a unique registration number; and a course has as attributes a title and a course code, the latter identifying it.

One wishes a data base for this application, i.e. one storing this information and whose evolution complies with the above constraints.

Data Model: ERA data model

For the sake of simplicity, we shall consider a version of the ERA (short for entity-relationship-attribute) data model [Chen '76], supporting attributes only for entities but not for relationships [Furtado et al. '83].

One can, within an entity-set, create an entity, which then exists until it is deleted. One can also link entities, provided they exist, via a relationship, whereupon they become related under the relationship, remaining so until they are unlinked. Attributes of entities have values, which can be modified.

Of course, some variations are possible; we choose this version for definiteness.

Representation of Academic Application on ERA Data Model

One usually regards modeling by the ERA model as displaying the application in terms of boxes (for entities), diamonds (for relationships) and ovals (for attribute values).

A representation of an application on a data model explains, by an interpretation, the concepts pertaining to the former in terms of those provided by the latter.

In our case one might, for instance, interpret:
"course c is listed" as "within entity-set 'lstd', c exists",
"offer course c" as "create, within entity-set 'lstd', entity c";
"student s takes course c" as "s is related via relationship 'tks' to c";
"enroll student s in course c" as "link, via relationship 'tks', s to c",
"the code of course c is Math_1305" as "attribute 'cd' at c has value Math_1305".

3.2 Overview

Since the data model is assumed to have available implementations, we expect such a representation of an application on a data model to provide an implementation for the application. Usually the parameter sorts will be given an implementation by means of standard basic sorts once and for all. We will consider an implementation of the data model using this given implementation for the parameter sorts of the application. We then use the representation of the application on the data model to "extract" an adequate realization for the application.

When one takes a closer look at the situation, one notices that distinct languages - oriented to application and to data model - are involved. To clarify this situation we shall examine in more detail the languages and specifications involved.

4. Syntax and Behavior

Applications and data models can be given more structured descriptions by relying on (many-sorted) signatures involving sorts (denoting domains), predicates (for relations) and operations (for functions) [Enderton '72; Ehrig & Mahr '85].

4.1 Application Languages

One can view an application data base at the information level, focusing on its content and queries, or at the manipulation level, which emphasizes how the content is altered [Casanova et al. '84; Veloso & Furtado '86]. Both levels share a common set Srt of sorts.

Application Sorts

In our example we recognize four sorts, namely
Std and Crs, as well as Rg_nbr, Cr_cd and Cr_nm.

We may regard the sorts Rg_nbr, Cr_cd and Cr_nm as parameter sorts of the application.

Query Languages

To express application queries we use predicates and operations, involving the above sorts, forming the application query signature Qry.

As queries we have predicates as follows:
unary predicates rglr, over sort Std, and lstd, over sort Crs,
binary predicates: tks, over sorts Std and Crs, as well as
ttl, over sorts Crs and Cr_nm, and cd, over Crs and Cr_cd;
and as operation query we have id, from Std to Rg_nbr.

Notice that some, not crucial, decisions have been taken concerning which informal connections are operations or predicates. Also, by categorizing these symbols by way of boxes, diamonds or ovals we would be paving the way for an ERA representation.

By adding variables for the sorts we obtain a static query language *Qry*. We may then express some (static) properties of the application queries by means of sentences of this language. For instance, the above static constraint concerning students and courses may be expressed by $(\forall x_2{:}Crs)(\forall x_1{:}Std)\ [tks(x_1,x_2) \rightarrow lstd(x_2)]$.

This language caters to a static view focusing on snapshots of the application. The evolution of the data base involves several states; by extending the language with modalities, such as $\Box$ for always (in every future state), we may express some dynamic properties. For instance, the above dynamic constraint might be expressed as $(\forall x_1{:}Std)\ [actv(x_1) \rightarrow \Box actv(x_1)]$, where $actv(x_1)$ is an abbreviation for $(\exists x_2{:}Crs)\ tks(x_1,x_2)$.

Manipulation Language

Updates may be thought of as transformations on states. Our application languages still do not include symbols for the updates. Our application has a set Stp of updates, generating single-step transitions, such as
unary update rgstr {with rgstr(s) registering student s:Std},
unary update offr {with offr(c) offering course c:Crs};
binary update enrll {with enrll(s,c) enrolling student s:Std in course c:Crs}.
It also has an initialization
constant strt {for initializing the data base}.

By adding such symbols we obtain a language Mnpl for the application at the manipulation level. In this manipulation language we may express some properties of the updates; in particular their effects on predicates, for instance [rgstr(s)>rglr(s) and [offr(c)>lstd(c), in a notation reminiscent of that of dynamic logic [Harel '79]. Notice that in these languages the state is hidden, and implicitly referred to through modalities, such as [] and [rgstr(s)>. (In other words, updates may be thought of as having a hidden global state parameter.)

4.2 Data Model Languages

We shall consider static, repertoire and encapsulation views of our ERA data model, with corresponding signatures and languages.

ERA Sorts

In our ERA data model we have the following sorts:
A (for attributes) and V (for attribute values);
E (for entities), and T (for entity-sets), as well as
R_k (for m-ary relationships), for each positive natural k

These sorts will be shared by the diverse data model signatures and languages. We may regard the non-entity sorts as parameter sorts of the ERA data model.

Static Syntax

We need (predicate) symbols for expressing existence of entities, their entity-sets, as well as their relatedness under relationships. They form the ERA static signature Stat.

The ERA predicates are:
a unary predicate exs, over sort E,
a binary predicate e_st, over sorts T and E,
an k-ary predicate $rltd_k$, over sorts R_k,E,...,E, for each positive natural k.
We also have a ternary predicate hs_vl, over sorts A, E and V. (We could alternatively consider a binary operation val, from A and E to V.)

By adding variables for the sorts we obtain a language. We may then express some (static) properties of the data model by sentences of this static language. For instance, the property that each existing entity is within some entity-set might be expressed by $(\forall x{:}E)\ [exs(x)\rightarrow(\exists y{:}T)e\_st(x,y)]$, and the property that only existing entities can be related by sentences, involving quantification over relationship variables, like $(\forall x_1,x_2{:}E)\ \{[(\exists z{:}R_2)rltd_2(z,x_1,x_2)]\rightarrow[exs(x_1)\wedge exs(x_2)]\}$.

Repertoire syntax

The above language takes a static view regarding the data model instantaneously through snapshots. Its evolution involves several states that we wish to place under control of a repertoire Rprt of operations (symbolizing update and initializations).

To handle evolution, we now wish to refer explicitly to states and transformations on them. So, we add to the preceding sorts an explicit state sort $*$, where these operations take values.

The ERA repertoire consists of:

a ternary operation crt:$(T,E,*)\rightarrow *$ {creating, with entity-set t:T, entitity e:E},
a (k+1)-ary operation $lnk_k:(R_k,E,\ldots,E,*)\rightarrow *$ {linking, via $r:R_k$, entities $e_1,\ldots,e_k$:E}, for each positive natural k,
a constant (nullary operation) init:$\rightarrow *$ {initializing the state};
a quaternary operation mdf:$(A,E,V,*)\rightarrow *$ {modifying the value of attribute a:A on entity e:E to value v:V},
a binary operation dlt:$(E,*)\rightarrow *$, {deleting entitity e:E},
a (k+1)-ary operation $unlk_k:(R_k,E,\ldots,E,*)\rightarrow *$ {removing relationship $r:R_k$ among entities $e_1,\ldots,e_k$:E}, for each positive natural k.

By adding variables for the sorts we obtain a language, where some properties of the repertoire may be expressed. A simple example is $crt(x,y,crt(x',y',\eta))\approx crt(x',y',crt(x,y,\eta))$, where $\eta:*$.

A structure $\mathcal{R}$ for the repertoire Rprt provides realizations for its symbols: nonempty domains for its sorts and functions for its operations. Thus, we have an initial state $\mathcal{R}[init]\in\mathcal{R}[*]$; also for each update and values in the non-state domains we have an update invocation, say $\mathcal{R}[crt(t,e)]:\mathcal{R}[*]\rightarrow\mathcal{R}[*]$. Sequences of such update invocations starting with the initial state are called traces [Veloso & Furtado '84] and describe a state. We call such a structure $\mathcal{R}$ *state-named* when its state set $\mathcal{R}[*]$ is finitely generated [Ehrig & Mahr '85] by its traces: each state $\sigma\in\mathcal{R}[*]$ is the value $\mathcal{R}[\tau]$ of some trace τ. Notice that such a state-named structure realizes the idea of encapsulation: its state is manipulated only by the operations provided.

Encapsulation syntax

The static and repertoire signatures, sharing the common sorts, may be combined into an encapsulation signature Encpsl. We can then express properties of updates. For instance, we can write some effects of the update crt in the form: $exs^*(e,crt(t,e,\sigma))$ to mean that entity e exists in the state attained from state σ by creating, with entity-set t, entitity e.

The reason why we have used exs*, instead of the simpler exs, is that the latter is a unary predicate over sort E whereas the former is a binary predicate over sorts E and $*$. We shall later see that this is change is ineffectual, amounting almost to syntactic sugar.

The static signature Stat consists of sorts, such as E and T, and static predicates, such as exs and $rltd_2$. We now wish to regard these predicates as dynamic, in that they now depend also on an explicit state argument. We thus have, for instance, the predicates:

binary predicate exs*, over sorts E and $*$;

a binary predicate e_st*, over sorts T, E and $*$;

(k+2)-ary predicate rltd^*_k, over sorts R_k,E,...,E and $*$, for each positive natural k.

In general, each static m-ary predicate p, over sorts $s_1,\dots,s_m$, in Stat has a dynamic version with explicit state sort: (m+1)-ary predicate p*, over sorts $s_1,\dots,s_m,*$ in Stat*; so each static formula $\varphi(v_1,\dots,v_m)$ has a dynamic counterpart $\varphi^*(v_1,\dots,v_m,\eta)$. For instance, the above static sentence τ concerning entities and entity-sets has as its dynamic counterpart $\tau^*(\eta)$ the formula $(\forall x{:}E)\ [\mathrm{exs}^*(x,\eta)\rightarrow(\exists y{:}T)\mathrm{e\_st}^*(x,y,\eta)]$, with $(\forall\eta{:}*)\ \tau^*(\eta)$ expressing that it always holds.

In this language we can describe some properties of updates, such as their effects on the static predicates and operations. For instance, with implicit universal quantification,

$[\mathrm{exs}^*(x_1,\eta)\wedge\mathrm{exs}^*(x_2,\eta)]\rightarrow\mathrm{rltd}^*_2(z,x_1,x_2,\mathrm{lnk}_2(z,x_1,x_2,\eta))$ and

$\{\mathrm{rltd}^*_2(z,x_1,x_2,\eta)\rightarrow[\mathrm{dlt}(x_1,\eta)\approx\eta\wedge\mathrm{dlt}(x_2,\eta)\approx\eta]\}$.

4.3 Application Behavior

A structure for the application will amount to a possible evolution of the data base; as such, it consists of instantaneous states obtained by invocations of its traces. By a realization for the application we mean such a structure that exhibits the specified behavior.

Static Behavior of the Application

The static behavior of the application concerns each instantaneous state: a structure for the query signature. It is required to satisfy the declared static constraint

Stt: $(\forall x_2{:}\mathrm{Crs})\ [(\exists x_1{:}\mathrm{Std})\mathrm{tks}(x_1,x_2)\rightarrow\mathrm{lstd}(x_2)]$.

The static constraint is to hold at every state; so it should be understood as $\Box$Stt. The states satisfying the static constraint are called (statically) valid, forming a set Val of structures for the query signature.

Dynamic Behavior of the Application

We also have restrictions involving more than one state. For instance, the dynamic constraint concerning active students

Act: $(\forall x_1{:}\mathrm{Std})\ [\mathrm{actv}(x_1)\rightarrow\Box\mathrm{actv}(x_1)]$; where $\mathrm{actv}(x_1)$ abbreviates $(\exists x_2{:}\mathrm{Crs})\ \mathrm{tks}(x_1,x_2)$.

Also, since the parameter sorts will usually be given an implementation in terms of primitive sorts, one would normally wish, say, the registration number of a student to remain stationary during the evolution of the data base. Query operations, such as id, and predicates, such as cd, with this property will be called *stationary*.

These requirements impose some constraints on the behavior of the application updates. For instance, a student should not be allowed to enroll in a course that is not listed.

State-explicit Version of Static Behavior

The languages used so far for describing application behavior rely on hiding the state, implicitly referred to via modalities. Another, more expressive, alternative introduces an explicit state sort, thereby obtaining versions of the application languages with explicit state.

We first extend the sorts of the application by adding an explicit state sort $*$: $\mathrm{Srt}^*:=\mathrm{Srt}\cup\{*\}$. We now wish to replace symbols with implicit state by their explicit-state versions.

We obtain Qry* from Qry by replacing each static query by its explicit-state version, as follows:

we replace each non-stationary query,
such as rglr over sort Std, by rglr*, over sorts Std and $*$;

we keep the stationary queries, such as id and cd, as they are.

We thus obtain the explicit-state version of the query language of the application. Notice that the stationary behavior of id and cd is already built onto this syntax. But, in this language we can express stationary behavior; for instance, if one later wishes course titles to become sationary, by Sttnr[ttl*]:

$(\forall x:\mathrm{Crs})(\forall u:\mathrm{Cr\_nm})(\forall \xi,\zeta:*)\ [\mathrm{ttl}^*(x,u,\xi)\leftrightarrow \mathrm{ttl}^*(x,u,\zeta)]$.

State-explicit Manipulation Behavior of the Application

In a similar spirit we introduce explicit state argument for the updates and initialization, thereby obtaining their explicit-state versions. We shall have, for instance, update operations, such as $\mathrm{rgstr}^*:(\mathrm{Std},*)\rightarrow *$ and $\mathrm{enrll}^*:(\mathrm{Std},\mathrm{Crs},*)\rightarrow *$, as well as a constant initialization strt* of sort $*$.

We thus obtain an explicit-state version Mnpl* of the manipulation signature of the application. In its language we can express properties of updates and queries. For instance,

$\mathrm{lstd}^*(c,\eta)\rightarrow(\exists x:\mathrm{Std})\mathrm{tks}^*(x,c,\mathrm{enrll}^*(s,c,\eta))$ for

$\mathrm{lstd}(c)\rightarrow[\mathrm{enrll}(s,c)>(\exists x:\mathrm{Std})\mathrm{tks}(x,c)]$.

4.4 Data Model Specification

A data model specification describes its possible behaviors by means of sentences of its languages. A structure for the signature satisfying these sentences is a data model realization.

Manipulation Specification of the Data Model

The manipulation specification Dt_Mdl of the data model consists of sentences of the manipulation language, with explicit state sort, describing properties of its symbols. For instance, inertia, effect and frame axioms like (with implicit universal quantification):

$\neg[\mathrm{exs}^*(x_1,\eta)\wedge \mathrm{exs}^*(x_2,\eta)]\rightarrow \mathrm{lnk}_2(z,x_1,x_2,\eta)\approx\eta$;

$[\mathrm{exs}^*(x_1,\eta)\wedge \mathrm{exs}^*(x_2,\eta)]\rightarrow \mathrm{rltd}^*_2(z,x_1,x_2,\mathrm{lnk}_2(z,x_1,x_2,\eta))$;

$[\mathrm{lnk}_2(z,x_1,x_2,\xi)\approx\zeta\wedge\neg(x'_1\approx x_1\wedge x'_2\approx x_2)]\rightarrow[\mathrm{rltd}^*_2(z,x'_1,x'_2,\xi)\leftrightarrow \mathrm{rltd}^*_2(z,x'_1,x'_2,\zeta)]$,

$\mathrm{lnk}_2(z,x_1,x_2,\xi)\approx\zeta\rightarrow[\mathrm{exs}^*(x,\xi)\leftrightarrow \mathrm{exs}^*(x,\zeta)]$.

From this manipulation specification one can derive some properties involving sentences of its sub-languages: constraints supported or imposed by the ERA data model.

Dynamic Constraints supported by ERA

Some consequences of the manipulation specification can be regarded as constraints on the operations of its repertoire: dynamic constraints supported by the data model. For instance
$(\forall\eta:*)(\forall z:R_2)(\forall x_1,x_2:E)\ \{rltd^*_2(z,x_1,x_2,\eta)\rightarrow[dlt(x_1,\eta)\approx\eta\wedge dlt(x_2,\eta)\approx\eta]\}$.

Static Constraints supported by ERA

Sentences of the static language that follow from the manipulation specification can be viewed as static constraints supported by the data model. Among these we have, for instance
$[exs^*(x,\eta)\rightarrow(\exists y:T)e\_st^*(x,y,\eta)]$ as $[exs(x)\rightarrow(\exists y:T)e\_st(x,y)]$, and
$\{rltd^*_2(z,x_1,x_2,\eta)\rightarrow[exs^*(x_1,\eta)\wedge exs^*(x_2,\eta)]\}$ as
$\{rltd_2(z,x_1,x_2)\rightarrow[exs(x_1)\wedge exs(x_2)]\}$.

Sort Constraints imposed by ERA

As expected, the above specification places no requirement on the nature of the ERA parameter sorts. For instance, given arbitrary nonempty sets W and S, there exist realizations D for its static signature Stat with these sets realizing value and entity-set sorts: $D[V]=W$ and $D[T]=S$.

5. Translating Application on Data Model

Now that we have languages for application and data model, we may describe in more detail the representation informally hinted at above. Our suggestion already uses the semantics of the data model to enforce part of the behavior intended for the application.

This suggestion represents attributes in terms of hs_vl. As examples, for c:Crs: ttl(c,n), with n:Cr_nm, is represented by hs_vl('ttl',c,n), and cd(c)≈d, with d:Cr_cd, by hs_vl('cd',c,d).

The other predicates are represented in terms of existence and relationships. For instance, with c:Crs, lstd(c) is represented by st_ex('lstd',c), where st_ex(y,x) abbreviates exs(x)∧e_st(y,x); whereas tks(s,c) is represented by $rltd_2$('tks',s,c), and rglr(s) by $rltd_1$('rglr',s), where s:Std.

The updates of the application are represented in terms of those of the data model. For instance, offr(c) is represented by crt('Crs',c,σ), enrll(s,c) by lnk_2('tks',s,c,σ), and rgstr(s) by lnk_1('rglr',s,σ).

This representation was naturally described in parts: query and update representations. It can also be regarded as consisting of three steps: first classify application with respect to the data model, then instantiate the latter for the former, as a preparation for the third step, namely translation (and interpretation). We now turn to a closer look at these stages.

5.1 Classification of Vocabularies

We represent the application updates by means of the repertoire of operations of the data model. It remains to represent the query signature of the application in terms of the static syntax of the data model. For this purpose, we first categorize the symbols of the application query signature Qry by means of boxes, diamonds and ovals. This involves classifying application sorts and queries (predicates or operations) in terms of the ERA categories. Our classification will mirror the preceding informal suggestions.

Sort Classification

We classify application sorts in Srt as value or entity sorts:
sorts Rg_nbr, Cr_cd and Cr_nm as value sorts, and
sorts Std and Crs as entity sorts.

Query Classification

We classify the query predicates and operations of Qry as valued, entity or relationship:
queries ttl as well as id and cd as valued attributes,
query lstd as entity predicate,
queries rglr and tks as relationship predicates.

Notice that this classification mirrors the preceding informal suggestions.

5.2 Instantiation of Data Model for Application

Instantiation is a preparatory step for translation. The application concerns specific relationships, whereas the data model deals with arbitrary ones. Thus, we extend the data model signature by adding constants and predicates, to code, and be able to recover, information expressed in the application language. For this purpose, we add to the signature of the data model appropriate symbols so as to capture the syntax of the application.

Static Query Instantiation

We extend the data model signature with symbols according to this classification. We add relativization predicates corresponding to the value sorts and then constants corresponding to the other application symbols.

We first instantiate the data model for application sorts, according to the sort classification, by adding relativization predicates and constants.

Corresponding to the value sorts we add unary relativization predicates over sort V:
value relativization predicates "Rg_nbr", "Cr_cd" and "Cr_nm" over sort V;

and corresponding to the entity sorts we add constants to sort T:
entity-set constants 'Std' and 'Crs' to sort T.

We now extend the sort instantiation for application queries, according to the query classification, by adding constants into appropriate ERA sorts.

Corresponding to the valued attributes we add constants to sort A:
attribute constants 'ttl', 'id' and 'cd' to sort A;

corresponding to the entity predicates we add constants to sort E:
entity constant 'lstd':E for entity query lstd;

and corresponding to the relationship predicates we add constants into proper sorts R_k's:
relationship constant 'rglr':R_1 for unary relationship predicate rglr;
relationship constant 'tks':R_2 for binary relationship predicate tks.

This instantiation process also carries other application symbols, if any, in a natural way; for instance, it would carry application constants John_Doe, of entity sort Std, and Math_1305, of value sort Cr_cd, to instantiation constants John_Doe, of sort E, and Math_1305, of sort V. So, instantiation adds symbols (mainly constants and unary relativization predicates) to the data model signature, whereby an instantiated signature $\mathrm{Stat}^{\mathrm{Qry}}$ arises.

Since relativization predicates are intended to represent application sorts, one wishes them to be nonempty. A structure *I* for the instantiated signature will be called *proper* when the realization of each relativization predicate is nonempty, e. g. I["Cr_cd"]$\neq\emptyset$. This requirement can be expressed by a set of sentences, like $(\exists u{:}V)$ "Cr_cd"(u), of the instantiated language.

A simple property of this instantiation process is:
any structure *D* for the signature Stat of the data model can be expanded to a structure D^{A} for the instantiated signature $\mathrm{Stat}^{\mathrm{Qry}}$ which is proper.

Dynamic Query Instantiation

We now obtain a dynamic version of the instantiated signature $\mathrm{Stat}^{\mathrm{Qry}}$ by replacing each static predicate or operation p by its dynamic version p*, as above. But, since we wish the added relativization predicates to be stationary, we keep them as they are, adding state sort only to the remaining predicates. For instance, we have a predicate exs*, over sorts E and $*$, but keep relativization predicate "Cr_nm" over sort V.

In other words, we may regard the dynamic instantiation as extending the dynamic version Stat* of the static signature of the data model with respect to the state-explicit query signature Qry* of the application. We then have a dynamic instantiated signature $\mathrm{Stat}^{\mathrm{Qry}*}$ with symbols for the state-explicit query signature of the application. A structure M^* for the dynamic instantiated signature $\mathrm{Stat}^{\mathrm{Qry}*}$ has a reduct *M* to its static sub-signature $Stat^{Qry}$ with the same realization for each relativization predicate.

5.3 Translation of Application into instantiated Data Model

Having prepared the data model by instantiation, we can now translate the application concepts in terms of those of the instantiated data model. We can proceed by levels: first information (translating queries), then manipulation (translating updates and initialization).

Static Query translation

We translate from Qry to $Stat^{Qry}$, by relying on the categorization of application symbols.

We translate valued attributes in terms of hs_vl, e. g.

valued predicate query ttl(x,u), with x:Crs and u:Cr_nm, to hs_vl('ttl',x,u),

valued operation query $id(x) \approx u$, with x:Std and u:Rg_nbr, to hs_vl('id',x,u).

We translate entity predicates in terms of st_ex(y,x), i. e. $exs(x) \wedge e\_st(y,x)$:

entity predicate query lstd(x), with x:Crs, to st_ex('lstd',x).

We translate relationship predicates in terms of appropriate $rltd_k$:

unary relationship predicate rglr(x), with x:Std, to $rltd_1$('rglr',x);

binary relationship predicate $tks(x_1,x_2)$, with x_1:Std and x_2:Crs, to $rltd_2('tks',x_1,x_2)$.

This translation process relies on a translation of sorts $s:Srt \rightarrow Stat^{Qry}$ with relativization:

we translate value sorts to V with the given relativization predicate, e. g.

(u:Cr_nm) to (u:V) with relativization "Cr_nm"(u).

we translate entity sorts to E with the relativizations defined via e_st(y,x),:

(x:Std) to (x:E) with relativization e_st('Std',x),

(x:Crs) to (x:E) with relativization e_st('Crs',x).

Our query translation $q:Qry \rightarrow Stat^{Qry}$ maps the symbols in the application signature Qry to corresponding ones in the instantiated signature $Stat^{Qry}$ of data model. By translating variables in the natural manner and relativizing quantifiers, we obtain a translation q mapping a formula $\varphi(v_1,\dots,v_m)$ of the application language to formula $q[\varphi(v_1,\dots,v_m)]$ of the instantiated data model language. For instance, application formula $(\forall x:Crs)(\exists u:Cr\_nm)ttl(x,u)$ is translated to formula $(\forall x:E)\{e\_st('Crs',x) \rightarrow (\exists u:V)["Cr\_nm"(u) \wedge hs\_vl('ttl',x,u)]\}$.

One would also desire (backward) induction of structures: a proper structure *I* for the instantiated signature inducing a structure I_q by using translation q.

For a value sort, say Cr_nm, I_q[Cr_cd]:=I["Cr_cd"], a subset of I[V], whereas for an entity sort, say Std, we have I_q[Std]:=I[e_st('Std',x)], a subset of I[E]. For a valued predicate, such as ttl, we have a subset I[hs_vl('ttl',x,u)] of I[E] x I[V], and we take I_q[ttl]:=I[hs_vl('ttl',x,u)]$\cap$(I_q[Std]xI_q[Cr_cd]); for an entity predicate, say lstd, we have a subset I[exs]$\cap$ I[e_st('lstd',x)] of I[E] and set I_q[lstd]:=(I[exs]$\cap$ I[e_st('lstd',x)])$\cap$ I_q[Crs]; finally for a relationship predicate, such as tks, we have a subset I[rltd$_2$('tks',x_1,x_2)] of I[E]xI[E], so we put I_q[tks]:=I[rltd$_2$('tks',x_1,x_2)]$\cap$(I_q[Std]xI_q[Crs]).

We wish this backward induction to produce a structure for the application signature Qry. For this, I_q[Crs]:=I[e_st('Crs',x)] should be a nonempty set. Also, for valued operation id this process produces a relation I_q[id]$\subseteq$$I_q$[Std]x$_q$[Rg_nbr], which should be a total function from I_q[Std] into I_q[Rg_nbr]. The situation is similar to the case of interpretation with relativization predicates [Enderton '72]: these requirements for syntax preservation can be expressed by a set Cls^{Qry} of closure sentences of the instantiated language. For instance, ($\exists$ x:E) e_st('Crs',x) and ($\forall$x:E) {e_st('Std',x)$\rightarrow$($\exists$!u:V) ["Rg_nbr"(u)$\wedge$hs_vl('id',x,u)]}. A proper structure for signature Stat^{Qry} satisfying these closure requirements will be called *properly closed.*

We then have backward induction, as desired:

every structure I for the instantiated signature Stat^{Qry} that is properly closed induces a structure I_q for the application signature Qry.

Notice that, by construction, the induced structure exhibits the expected behavior, e.g. I_q satisfies ($\forall$$x_1$,$x_2$:E) {rltd$_2$('tks',$x_1$,$x_2$)$\rightarrow$[e_st('Std',$x_1$)$\wedge$e_st('Crs',$x_2$)]}.

A consequence of these processes is a translation lemma (much as in [Enderton 72, p. 161]):

given an assignment $a_1,\ldots,a_m$ in the domains of properly closed I
$I_q \rhd \varphi(v_1,\ldots,v_m)$ $[a_1,\ldots,a_m]$ iff $I \rhd q[\varphi(v_1,\ldots,v_m)]$ $[a_1,\ldots,a_m]$;
where we use $I \rhd \varphi(\underline{v})$ $[\underline{a}]$ for satisfaction.

The above mapping q from the application signature to the instantiated signature is injective. So, we also have forward induction of structures:

every structure A for the application signature Qry induces a structure A^q for the instantiated signature Stat^{Qry}, which is properly closed; moreover forward and backward inductions are inverses to each other: $(A^q)_q=A$.

This situation is similar to the familiar reduction of many-sorted logic to unsorted logic with relativization predicates [Enderton '72]. As such, we have a faithful translation preserving definability. This means that we can replace the application language by that of the instantiated data model without any loss of deductive or expressive powers.

For instance, the application uses formula $(\exists x_2{:}Crs)\ tks(x_1,x_2)$ to talk about the application concept active students; we can use instead its translation $(\exists x_2{:}E)\ [e\_st('Crs',x_2) \wedge rltd_2('tks',x_1,x_2)]$, because they define the same sets $A[(\exists x_2{:}Crs)\ tks(x_1,x_2)] = I[(\exists x_2{:}E)\ [e\_st('Crs',x_2) \wedge rltd_2('tks',x_1,x_2)]]$ whenever *A* and *I* correspond to each other.

Another property of this instantiation process is:

given an arbitrary nonempty set W, there exists a data model realization *D* with *D*[V]=W whose expansion D^A to the instantiated signature $Stat^{Qry}$ is properly closed.

Dynamic Counterpart of Static Query Translation

The application deals with the state implicitly via modalities, whereas the ERA data model has an explicit state sort. So, we wish to adapt our query translation $q{:}Qry \rightarrow Stat^{Qry}$ to a version with explicit state sort $*$.

Recall that a dynamic counterpart is obtained by replacing each static predicate or operation p by its version p* with explicit state. But, we now have two such versions: explicit-state version Qry* on the application side, and dynamic version $Stat^{Qry*}$ for the instantiated data model.

The query translation q induces a natural dynamic translation q*: given application query a, with query translation $a^q=i$, translate its explicit-state version $a^*(\eta)$ to $i^*(\eta)$. By construction, applying translation q* to the application explicit-state version is the same as applying the static translation q and then obtaining its dynamic counterpart in the instantiated data model. For instance, consider an application formula θ, say $(\exists x_2{:}Crs)tks(x_1,x_2)$, its query translation $q[\theta]$ is formula $(\exists x_2{:}E)[e\_st('Crs',x_2) \wedge rltd_2('tks',x_1,x_2)]$. Now, the explicit-state version θ^* is $(\exists x_2{:}Crs)tks^*(x_1,x_2,\eta)$, which q* maps to the dynamic version $(q[\theta])^*$ of $q[\theta]$, namely $(\exists x_2{:}E)[e\_st('Crs',x_2,\eta) \wedge rltd^*_2('tks',x_1,x_2,\eta)]$. Similarly, since $q[id(x) \approx u] = [hs\_vl('id',x,u)]$, q* maps it to $hs\_vl^*('id',x,u,\eta)$, and thus $q^*[id(x) \approx u \rightarrow (\exists x_2{:}Crs)tks^*(x_1,x_2,\eta)]$ becomes $hs\_vl^*('id',x,u,\eta) \rightarrow (\exists x_2{:}E)[e\_st('Crs',x_2,\eta) \wedge rltd^*_2('tks',x_1,x_2,\eta)]$.

The dynamic translation q* maps the explicit-state version Qry* of the application query signature into the dynamic version $Stat^{Qry*}$ of the instantiated data model signature. Thus, the previous induction of structures carries over to the dynamic case:

we have bijections between structures *M* for the dynamic instantiated signature $Stat^{Qry*}$, whose static reducts M_* are properly closed, and *A* for the explicit-state version Qry* of the application query signature, which are inverses to each other.

Also, if *M* and *A* correspond to each other then:

1. $M[*]=A[*]$ and their static reducts M_* and A_* correspond to each other as well;

2. $M \rhd \varphi^*(\underline{v},\eta)\ [\underline{a},\sigma]$ iff $A \rhd \psi^*(\underline{v},\eta)\ [\underline{a},\sigma]$ whenever $q[\varphi(\underline{v})]=\psi(\underline{v})$.

Manipulation translation

We can now complete the translation of the application to the instantiated data model by explicating updates of the former in terms of those of the latter. In accordance with our informal suggestion, we extend the dynamic counterpart s* of our sort translation s:Srt→$Stat^{Qry}$.

We translate application state-manipulating operations as follows:

initialization strt to init;
update offr(x), i. e. offr*(x,η), to crt('lstd',x,η);
update rgstr(x), i. e. rgstr*(x,η), to lnk_1('rglr',x,η),
update enrll(x_1,x_2), i. e. enrll*(x_1,x_2,η), to lnk_2('tks',x_1,x_2,η).

We thus have a translation u:Mnpl→$Encpsl^{Mnpl*}$, or equivalently u*:Mnpl*→$Encpsl^{Mnpl*}$, mapping query and update symbols of the application manipulation signature to corresponding ones in the full instantiated signature $Encpsl^{Mnpl*}$ of data model. By translating variables as before, we achieve translation of formulas: an application formula, such as $lstd^*(x_2,\eta)\rightarrow(\exists x{:}Std)tks^*(x,x_2,enrll^*(x_1,x_2,\eta))$, is translated to a formula like $st\_ex^*('lstd',x_2,\eta)\rightarrow(\exists x{:}E)[st\_ex^*('Std',x)\wedge rltd^*_2('tks',x,x_2,lnk_2('tks',x_1,x_2,\eta))]$. As for the peceding translations, we have backward induction of structures:

every structure *M* for the full instantiated signature $Encpsl^{Mnpl*}$, with properly closed static reduct, induces a structure M_u for the state-explicit application signature Mnpl*.

Notice that $M_u[*]=M[*]$ and, e. g. $M_u[Crs]=M_q[Crs]$. Thus, we have initialization $M_u[strt^*]:=M[init]\in M[*]$, and updates, such as $M_u[offr^*]:M_q[Crs]\times M[*]\rightarrow M[*]$, defined by $M_u[offr^*](c,\sigma):=M[crt](M['lstd'],c,\sigma)$ and $M_u[rgstr^*]:M_q[Std]\times M[*]\rightarrow M[*]$ by $M_u[rgstr^*](s,\sigma):=M[lnk_1](M['rgstr'],s,\sigma)$.

Since our extension u of q* remains injective, we still have forward induction of structures:

every structure *A* for the state-explicit signature Mnpl* of the application induces a structure A^u for the full instantiated signature $Encpsl^{Mnpl*}$, with properly closed static reduct; moreover forward and backward inductions are inverses to each other: $(A^u)_u=A$; also, if *M* and *A* correspond to each other and $u[\varphi(\underline{v},\eta)]=\psi(\underline{v},\eta)$:
$M \rhd \varphi(\underline{v},\eta)\ [\underline{a},\sigma]$ iff $A \rhd \psi(\underline{v},\eta)\ [\underline{a},\sigma]$.

5.4 Application Realizations from Translation on Data Model

Let us now examine how one can extract a realization for the application from one for the instantiated data model by combining the preceding constructions.

Consider nonempty sets Rgn, Crd and Cnm, as possible realizations for the application parameter sorts Rg_nbr, Cr_cd and Cr_nm of the application.

As mentioned, we have a data model realization D with $D[\mathrm{V}]=\mathrm{Rgn}\cup\mathrm{Crd}\cup\mathrm{Cnm}$ and properly closed expansion D^{A} to the instantiated signature $\mathrm{Stat}^{\mathrm{Qry}}$. Notice that, e. g. D^{A}["Cr_cd"]=Crd.

Now, structure D^{A} for the full instantiated signature $\mathrm{Encpsl}^{\mathrm{Mnpl}^*}$ induces a structure $(D^{A})_{u}$ for the state-explicit application signature Mnpl*.

Notice that for an application parameter sort, say Rg_nbr, we have $(D^{A})_{u}$[Cr_cd]=$(D^{A})_{q}$[Cr_cd]=D^{A}["Cr_cd"]=Crd.

Thus, given possible realizations $A[\mathrm{v}]$ for the application parameter sorts, we have a realization $(D^{A})_{u}$ for the state-explicit application signature with $(D^{A})_{u}[\mathrm{v}]=A[\mathrm{v}]$.

Moreover, since forward and backward inductions are inverses to each other, for any application formula $\varphi(\underline{v},\eta)$, with translation $u[\varphi(\underline{v},\eta)]=\psi(\underline{v},\eta)$, we have $(D^{A})_{u} \rhd \varphi(\underline{v},\eta)\ [\underline{a},\sigma]$ iff $D^{A} \rhd \psi(\underline{v},\eta)\ [\underline{a},\sigma]$. Thus, any specifiable behavior for the application can be described on the instatiated data model. For instance, for a course $c\in(D^{A})_{u}[\mathrm{Crs}]$, application behavior $(D^{A})_{u} \rhd (\forall\xi,\zeta:*)(\forall u:\mathrm{Cr\_nm})\ [\mathrm{ttl}^*(x,u,\xi)\leftrightarrow\mathrm{ttl}^*(x,u,\zeta)]\ [c]$ is equivalent to
$D^{A} \rhd (\forall\xi,\zeta:*)(\forall u:\mathrm{V})\ \{\text{"Cr\_nm"}(u)\rightarrow[\mathrm{hs\_vl}^*(\text{'ttl'},x,u,\xi)\leftrightarrow\mathrm{hs\_vl}^*(\text{'ttl'},x,u,\zeta)]\}$ $[c]$.

Hence, $(D^{A})_{u}$ is a realization for the state-explicit application signature Mnpl*, obtained from possible realizations for the application parameter sorts, where we can faithfully reason about the application properties, as expressed by application formulas $\varphi(\underline{v},\eta)$, via their translations $u[\varphi(\underline{v},\eta)]$.

6. Application Representation and External Schemas

In the preceding section we have examined the process of translating an application on a data model in terms of their languages. It involves instantiating the signature of the data model with respect to that of the application and then translating the application signature into the instantiated signature. We have noticed that each such process has two dimensions: static, for queries, and dynamic, for updates. The combined effect of is a process, whereby each properly closed realization for the data model signature induces one for the application signature.

In representing an application on a data model we wish the induction process to provide a realization for the application, i. e. the induced realization should exhibit the required behavior.

6.1 Interpretation of Application on Data Model

A data model realization D gives rise to structure $(D^{A})_u$ for the state-explicit application signature Mnpl*, which has a state-named substructure $D^{\dagger}$. We wish the latter structure $D^{\dagger}$ to be a realization for the application. One way to guarantee this goal is by checking preservation of properties. This involves examining their specifications: the instantiated data model should be specified so that the translation becomes an interpretation of specifications [Shoenfield '67; Turski & Maibaum '87].

More specifically, what is required is that for each application constraint, static or dynamic, its translation will be satisfied in the realization $D^{\dagger}$ induced by the chosen data model realization.

For instance, consider the application static constraint Stt. Its translation q[Stt] is (equivalent to) the conjunction of the following two sentences of the instantiated language

ε: $(\forall x_1,x_2{:}E)\ \{[\text{e\_st}(\text{'Std'},x_1)\wedge \text{e\_st}(\text{'Crs'},x_2)\wedge \text{rltd}_2(\text{'tks'},x_1,x_2)]\rightarrow \text{exs}(x_2)\}$

λ:

$(\forall x_1,x_2{:}E)\ \{[\text{e\_st}(\text{'Std'},x_1)\wedge \text{e\_st}(\text{'Crs'},x_2)\wedge \text{rltd}_2(\text{'tks'},x_1,x_2)]\rightarrow \text{e\_st}(\text{'lstd'},x_2)\}$.

Now, our data model already supports some constraints, a static one being $(\forall z{:}R_2)(\forall x_1,x_2{:}E)\ [\text{rltd}_2(z,x_1,x_2)\rightarrow \text{exs}(x_1)\wedge \text{exs}(x_2)]$. From its particularization $(\forall x_1,x_2{:}E)\ [\text{rltd}_2(\text{'tks'},x_1,x_2)\rightarrow \text{exs}(x_1)\wedge \text{exs}(x_2)]$ we see that ε is a consequence of the the static constraints supported by the data model. To guarantee λ, one may, for instance, refine the translation of $\text{tks}(x_1,x_2)$ to $\text{rltd}_2(\text{'tks'},x_1,x_2)]\wedge \text{e\_st}(\text{'lstd'},x_2)$ (or, equivalently, refine the translation of update enrll). Then, the application structure induced by a properly closed data model realization will satisfy the application static constraint.

On the other hand, consider the expected behavior that id remains stationary, expressed in translation say by $[\text{e\_st}(\text{'Std'},x)\wedge \text{"Rg\_nbr"}(u)]\rightarrow[\text{hs\_vl*}(\text{'id'},x,u,\xi)\leftrightarrow \text{hs\_vl*}(\text{'id'},x,u,\zeta)]$. Since the latter formula is not a constraint supported by the data model, we have to add its universal closure as one of the axioms to obtain the specification of the instantiated data model.

We shall leave the dynamic constraint for later.

6.2 Extending Application and Representation

So far we have not paid much attention to the application updates cncl, to cancel a course, and drp, for a student to drop a course. Thus, we may say that we have represented only part of the application. We now wish to complete the representation by including them. Alternatively, we could view this stage as extending an application and a representation.

The previously examined instantiation of data model for application extends the signature of the former by adding constants and relativization predicates corresponding to the symbols of the latter.

Application Extension

Imagine that we extend our restricted application by adding two new updates: cncl, to cancel a course, and drp, for a student to drop a course.

Since we still wish the static constraint to hold, we would impose requirements on the new updates, such as: $(\forall x_2\text{:Crs})\ [(\exists x_1\text{:Std})\text{tks}^*(x_1,x_2,\eta)\rightarrow \text{cncl}^*(x_2,\eta)\approx\eta]$ and

$[\neg(\exists x_1\text{:Std})\text{tks}^*(x_1,x_2,\xi)\wedge \text{cncl}(x_2,\xi)\approx\zeta]\rightarrow$
$\rightarrow\{\neg\text{lstd}^*(x_2,\zeta)\wedge(\forall x\text{:Crs})[\neg x\approx x_2\rightarrow(\text{lstd}^*(x,\xi)\leftrightarrow\text{lstd}^*(x,\zeta)]\}$.

Extension of Representation

An extension of our translation so as to cover the extended application can be obtained by translating the new updates as follows:

$\text{cncl}^*(x,\eta)$ to $\text{dlt}(x,\eta)$,
$\text{drp}^*(x_1,x_2,\eta)$ to $\text{unlk}_2(\text{'tks'},x_1,x_2,\xi)$.

To guarantee that this extended translation still preserves properties, one should check the translations of the new axioms added as requirements.

For instance, consider the first axiom χ about cncl. Its translation $q[\chi]$ is (equivalent to) the following sentence of the instantiated language
$(\forall x_2\text{:E})\ \{[\text{e\_st}(\text{'Crs'},x_2)\wedge(\exists x_1\text{:E})(\text{e\_st}(\text{'Std'},x_1)\wedge\text{rltd}^*_2(\text{'tks'},x_1,x_2,\xi))]\rightarrow\text{dlt}(x_2,\xi)\approx\xi\}$.

A dynamic constraint supported by our data model, as seen in 4.4, is $(\forall z\text{:}R_2)(\forall x_1,x_2\text{:E})\ \{\text{rltd}^*_2(z,x_1,x_2,\xi)\rightarrow[\text{dlt}(x_1,\xi)\approx\xi\wedge\text{dlt}(x_2,\xi)\approx\xi]\}$, which entails $(\forall x_2\text{:E})\ \{[(\exists x_1\text{:E})\ \text{rltd}^*_2(\text{'tks'},x_1,x_2,\xi)]\rightarrow\text{dlt}(x_2,\xi)\approx\xi\}$. So, we see that the translation $q[\chi]$ of χ is a consequence of the dynamic constraints supported by the data model, hence, the application structure induced by a properly closed data model realization will satisfy axiom χ.

Let us now examine the dynamic constraint Act, which we have postponed.

A data model realization D gives rise to state-named structure $D^\dagger$. Now, Act will hold in such encapsulated structure $D^\dagger$ provided it holds for each single-update transition, e. g.
$(\forall x,x_1\text{:Std})(\forall x_2\text{:Crs})(\forall\xi,\zeta\text{:}*)\ \{[\text{actv}^*(x,\xi)\wedge\text{enrll}(x_1,x_2,\xi)\approx\zeta]\rightarrow\text{actv}^*(x,\zeta)\}$ and
$(\forall x,x_1\text{:Std})(\forall x_2\text{:Crs})(\forall\xi,\zeta\text{:}*)\ \{[\text{actv}^*(x,\xi)\wedge\text{drp}(x_1,x_2,\xi)\approx\zeta]\rightarrow\text{actv}^*(x,\zeta)\}$.

Consider the latter case and call it δ. We know that $D^\dagger \rhd \delta$ iff $(D^\dagger)^u \rhd u[\delta]$. Since we cannot guarantee the latter, we are led to refining the translation of drp to:
if $(\exists x_1\text{:E})\ [\text{e\_st}(\text{'Std'},x_1)\wedge\text{rltd}^*_2(\text{'tks'},x_1,x_2,\xi)]$ **then** ξ **else** $\text{unlk}_2(\text{'tks'},x_1,x_2,\xi)$.

By refining the translations and adding appropriate axioms, we obtain a specification $\text{Dt\_Mdl}^{\text{Appl}}$ for the instantiated data model extending specification Dt_Mdl for the data model. Then our translation $u^*\text{:Mnpl}^*\rightarrow\text{Encpsl}^{\text{Mnpl}^*}$ will become an interpretation of specifications and, as a result, every data model realization will give rise to a realization for the application.

6.3 Application Representation and Users' Interfaces

We may wish to extend our application even further to cover both teaching and research activities. For this purpose, we extend our application signatures by adding a new sort Fclt (for faculty members); query predicates tchs (with tchs(f,c) intended to mean that faculty member f:Fclt teaches course c:Crs) and advs (with advs(f,s) intended to mean that faculty member f:Fclt advises research of student s:Std); as well as updates assgn (with assgn(f,c) assigning faculty member f:Fclt to teach course c:Crs), sprvs (by sprvs(f,s) faculty member f:Fclt begins to advise research of student s:Std), and rlss (rlss(f,s) mirroring that faculty member f:Fclt ceases to advise research of student s:Std). Let us call Univ this extended application. Notice that, if one wishes the course assignment to remain stationary, this can be expressed, as before, by $(\forall x_1{:}Fclt)(\forall x_2{:}Crs)(\forall \xi,\zeta{:}*)\ [tchs^*(x_1,x_2,\xi) \leftrightarrow tchs^*(x_1,x_2,\zeta)]$.

A natural extension of our representation to cover Univ is by translating:

(x:Fclt) to (x:E) with relativization e_st('Fclt',x), for sort Fclt;

$tchs(x_1,x_2)$, with x_1:Fclt and x_2:Crs, to $rltd_2('tchs',x_1,x_2)$ for query predicate tchs,

$advs(x_1,x_2)$, with x_1:Fclt and x_2:Std, to $rltd_2('advs',x_1,x_2)$ for query predicate advs;

$assgn^*(x_1,x_2,\eta)$, with x_1:Fclt and x_2:Std, to $lnk_2('tchs',x_1,x_2,\eta)$, for update assgn,

$sprvs^*(x_1,x_2,\eta)$, with x_1:Fclt and x_2:Std, to $lnk_2('advs',x_1,x_2,\eta)$, for update sprvs,

$rlss^*(x_1,x_2,\eta)$, with x_1:Fclt and x_2:Std, to $unlk_2('advs',x_1,x_2,\eta)$, for update rlss.

We would thus represent our extended application Univ on the ERA data model, much as before by guaranteeing the required behavior.

Now, let us consider external schemas, say the external schema of student John Doe. Imagine that it consists of two views - Study and Research - both providing restricted access to the application Univ. In the former John Doe can inspect the courses he is taking and alter his course situation, whereas in the latter he can see who advises whom and approach or abandon faculty members for research purposes. Their signatures are as follows.

John Doe's Stdy view has

sort Crs;

queries lstd and my_crss, both over sort Crs;

updates try and quit, both with argument from sort Crs.

and his Rsrch view has

sorts Fclt and Std;

queries who?, over sorts Fclt and Std;
updates apprch and abndn, both with argument from sort Fclt.
John Doe's interface with the application Univ can be described by translations of his view signatures into the language of Univ, as follows.
John Doe's Stdy view could be translated into Univ by mapping the new symbols:
my_crss(x), with x:Crs, to tks(John_Doe,x) for query my_crss;
try(x), with x:Crs, to enrll(John_Doe,x), for update try,
quit(x), with x:Crs, to drp(John_Doe,x), for update quit.
Likewise, his Rsrch view could be translated into Univ by mapping the new symbols:
who?(x_1,x_2), with x_1:Fclt and x_2:Std, to advs(x_1,x_2) for query who?;
apprch(x), with x:Fclt, to assgn(x,John_Doe), for update apprch,
abndn(x), with x:Fclt, to rlss(x,John_Doe), for update abndn.

So, much as in the case of representing an application on a data model, a user's interface with application Univ involves translations of his views into Univ, and these translations are required to be interpretations of the corresponding specifications for adequate behavior.

7. Conclusion

We have analyzed the process of representing an application concept on a data model, aiming at clarifying the roles played by the diverse language dichotomies: application vs. data model, static vs. dynamic, etc.

This analysis is suggested by the similarity with encapsulation and implementation of data types in Software Engineering, which in turn leads to interpretation of specifications based on translation of their languages.

The process of representing an application concept on a data model is explicable as an interpretation of the application specification into an appropriate extension of the data model specification. Both extension and interpretation involve information and manipulation levels, which can be dealt with subsequently. The information level involves instantiating the query language of the latter with symbols (mainly constants and unary relativization predicates) corresponding to those of the former, much as the familiar reduction of many-sorted reasoning to unsorted one in classical first-order logic. The manipulation level then extends the query interpretation by association of updates.

Let us now comment on some methodological aspects of this process. We have illustrated how an existing representation can be extended so as to cope with extensions of the application. We have also indicated, albeit briefly, how the process of trying to verify correctness can suggest refinements of a preliminary representation.

We have kept our example deliberately simple so as to concentrate on the main ideas. To see their generality notice, for instance, that they would carry over to the case of applications with declared keys. The data model could also be extended to add, for example, the notion of objects [Atkinson et al. '89]; if an entity is created in more than one entity-set, this entity should be regarded as the same object instance in all sets. Also, the logical formalizations for our simple version can be extended to cover other features of the usual ERA data models. This would be analogous to Reiter's logical reconstruction of the relational model [Reiter '84]. But we have also logically reconstructed other aspects, such as representation in terms of interpretation. This effort might suggest other data models as well [Kuper & Vardi '93].

By emphasizing their specifications the formal distinction between application concepts and data models practically vanish. It is then not difficult to visualize how these ideas carry over to the other levels of the ANSI-X3-SPARC architecture. Namely, by considering specifications for external schemas, application concepts, data models, and underlying machine-supported system, we regard them as formal objects of the same nature: abstract data types. In the same vein, see that user interfaces, data model representation, and physical implementation can be regarded as instances of the same general process: data type implementation (interpretation into a conservative extension) [Veloso & Maibaum '95].

REFERENCES

M. Atkinson et al. - The object-oriented database system manifesto. Proc. Intern. Conf. on Deductive and Object-Oriented Databases (40-57), 1989.

Casanova, M. A.; Veloso, P. A. S.; Furtado, A. L. - Database specification formalisms: an eclectic perspective. Proc. 3rd ACM Symposium on Principles of Database Systems (110-118). Waterloo, Canadá, 1984.

Chen, P. - The entity-relationship model: toward a unified view of data. ACM Trans. on Database Systems **1**(1), 1976.

Dosch, W. ; Mascari, G. ; Wirsing, M. - On the algebraic specification of databases. Proc. 8th International Conf. on Very Large Data Bases (370-385), 1982.

Enderton, H. B. - A Mathematical Introduction to Logic. Academic Press; New York, 1972.

Ehrig, H.; Mahr, B. - Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics. Springer-Verlag, Berlin, 1985.

Furtado, A. L. ; Casanova, M. A. ; Veloso, P. A. S. - Application-oriented approaches. In Furtado, A. L. ; Neuhold, E. J. (eds.) Formal Techniques for Data Base Design (5- 44). Springer-Verlag, Berlin, 1986.

Furtado, A. L. ; Maibaum, T. S. E. ; Veloso, P. A. S. - Especificações abstratas (de bancos de dados) via níveis de traço. Rev. Bras. de Computação **1**(3) 179-193, 1981.

Furtado, A. L. ; Veloso, P. A. S. ; Castilho, J. M. V. de - Verification and testing of S-ER representations. In Chen, P. P. (ed.) Entity-Relationship Approach to Information Modeling and Analysis (123-147). North-Holland, Amsterdam, 1983.

Gehani, N. and McGettrick, A. D. - Software Specifications Techniques. Addison-Wesley, Reading, 1986.

Guttag, J. V. - Abstract data types and the development of data structures. Comm. Assoc. Comput. Mach. **20**(6) 396-404, 1977.

Harel, D. - First-order Dynamic Logic. Springer-Verlag, Berlin, 1979.

Kowalski, R. - Logic for data description. In Mylopulos, J.; Brodie, M. L. (eds.) Artificial Inteligence and Databases (259-271). Morgan Kaufmann, 1989,

Kuper, G. M.; Vardi, M. Y. - The logical database model. ACM Trans. Database Systems **18**(3) 379-413, 1993.

Liskov, B.; Zilles, S. - An introduction to formal specifications of data abstractions. In Yeh, R. T. (ed.). Curent Trends in Programming Methodology, vol. I (1-32). Prentice-Hall, 1977.

Reiter, R. - Towards a logical reconstruction of relational database theory. Brodie, M. L.; Mylopulos, J.; Schmidt, J. W. (eds.) On Conceptual Modeling (191-238). Springer-Verlag, 1984.

Rescher, N.; Urquhart, A. - Temporal Logic. Springer-Verlag, Berlin, 1971.

Shoenfield, J. R. - Mathematical Logic. Addison-Wesley, Reading, 1967

Turski, W. M.; Maibaum, T. S. E. - The Specification of Computer Programs. Addison-Wesley, Wokingham, 1987.

Veloso, P. A. S. - Verificação e Estruturação de Programas com Tipos de Dados. Edgard Blücher, São Paulo, SP;1987.

Veloso, P. A. S.; Casanova, M. A.; Furtado, A. L.- Formal data base specification: an eclectic perspective. PUC - RJ, Dept. Informática, MCC 1/84, 1984.

Veloso, P. A. S.; Furtado, A. L. -Stepwise construction of algebraic specifications. In Gallaire, H.; Minker, J.; Nicholas, J. M. (eds.). Advances in Data Base Theory, vol. 2 (321-352). Plenum, New York, 1984.

Veloso, P. A. S. ; Furtado, A. L. - Towards simpler and yet complete formal specifications. In Langefors, B.; Verrijn-Stuart, A. A.; Bracchi, G. (eds.) Trends in Information Systems (257-271). North-Holland, Amsterdam, 1986.

Veloso, P. A. S. ; Maibaum, T. S. E. - On the modularization theorem for logical specifications. Inform. Proc. Letters 53 (287-293), 1995