Preliminary Explorations in Specifying and Validating Entity-Relationship Models in PVS

Venkatesh Choppella
Indian Institute of Information Technology and Management – Kerala
Thiruvananthapuram, India

Arijit Sengupta
Wright State University
Dayton, OH, USA

Edward L. Robertson
Indiana University
Bloomington, IN, USA

Steven D. Johnson
Indiana University
Bloomington, IN, USA

ABSTRACT

Entity-Relationship (ER) diagrams are an established way of doing data modeling. In this paper, we report our experience with exploring the use of PVS to formally specify and reason with ER data models. Working with a text-book example, we rely on PVS's theory interpretation mechanism to verify the correctness of the mapping across various levels of abstraction. Entities and relationships are specified as user defined types, while constraints are expressed as axioms. We demonstrate how the correctness of the mapping from the abstract to a conceptual ER model and from the conceptual ER model to a schema model is formally established by using typechecking. The verification involves proving the type correctness conditions automatically generated by the PVS type checker. The proofs of most of the type correctness conditions are fairly small (four steps or less). This holds out promise for complete automatic formal verification of data models.

Keywords

Data modeling, entity-relationship diagrams, mapping, formal methods, PVS, type checking, data refinement.

1. INTRODUCTION

Data modeling is a fundamental prerequisite for the physical design and implementation of a database system. Data modelers analyze the user's requirements and build *data models*, which are conceptual representations of real world enterprises.

A data model consists of a set of type, function, relation and constraint definitions. This model is validated for *consistency* and then used as a reference for further design refinements and *implementation*. The model serves as a *specification* to which the database design, usually specified in the form of a set of schemata, must conform. The most popular conceptual modeling framework is the *entity-relationship (ER) model* [11]. An ER model consists of a collection of *entities*, and *attributes* of and *relationships* among those entities. In addition, the model specifies *constraints* between its various entity and relationship sets.

A good modeling framework should allow a designer to (*a*) *express* and *reason* about data models at a high level of abstraction in a semantically precise manner, (*b*) *validate* the correctness of models across various levels of abstraction, and (*c*) *explore* design alternatives within correctness boundaries. In this paper, we explore the first and second problems: how to specify models at varying levels of abstraction using a specification language and then validate mappings between abstraction levels. We plan to address the third issue in a future paper.

1.1 Expressivity and correctness of mapping in ER modeling

ER models have an underlying formal semantics based on the elementary theory of sets and relations. Data modelers, however, prefer to employ ER diagrams, which are annotated, undirected graphs. The vertices in these graphs are the objects of the model: attributes, entities, and relationships. These objects are connected by edges which related these objects. In addition, an ER model consists of constraints. A limited set of decorations on vertices and edges encode key attributes, and participation and cardinality constraints. The diagrammatic approach allows for easy construction and intuitive understanding of models. But such notation does not easily extend to encoding arbitrary constraints arising from complex business logic. As a consequence, it becomes difficult for designers to express, much less prove, the correctness of their conceptual design and its mapping to a relational implementation. Designers therefore employ natural language to informally express nontrivial constraints. Natural language complements diagrammatic notation, but it is often the source of inaccuracies and ambiguities in specifications.

1.2 Specifying data models in PVS

How well can we apply the principles and techniques from formal specification languages to the ER data modeling prob-

AFM'07, November 6, Atlanta, GA, USA.

lem? We address this question by exploring the use of the Prototype Verification and Specification (PVS) language [2] for doing ER modeling. We look at two specific subproblems: (*a*) how to specify and reason with a data model at the abstract, ER and schema level, and, (*b*) how to prove that the mapping from abstract to ER and ER to schema is correct. Our preliminary experience indicates that the generality of PVS's specification language, its rich type system and libraries allow us to *reason* with our models in ways not easily possible with ER diagrams. Furthermore, this reasoning forms the basis of verifying the correctness of mapping between an ER model and the relational schemata that represent its implementation.

The mapping across different layers of abstraction of the ER model is an instance of the more general problem of *data refinement*: abstract, uninterpreted types and objects over those types at higher levels of abstractions are provided interpretations at the lower level of abstractions. To be sure, the problem of specification and the step-wise refinement to implementation has a long history in formal methods, software engineering and data models as well. This includes formally specifying the "Norman's database" example in VDM, RAISE and COLD-K [16, 18, 20, 37, 39]. Additionally, languages like Z, B and Alloy also come with various levels of automated support for data refinement and formal validation [1, 4, 22].

PVS is a modern, general purpose specification language with support for higher-order logic, dependent types, and interactive theorem proving environment for verification and typechecking. It is therefore natural to explore how well ER modeling can be done in PVS.

All specifications, including requirements, conceptual data models and logical models are expressed as *theories* in PVS. A theory is a basic PVS module and consists of declarations that define types, functions, axioms and theorems. Model-specific data constraints are encoded as axioms. Hard-to-anticipate constraints governing interactions between the various elements of the model are generated automatically *type correctness conditions* (TCCs). The modeler interactively verifies these TCCs to ensure that the model is consistent. The specification's correctness depends on verifying the type correctness conditions. For the example discussed in this paper, the majority of TCCs have proofs that are quite small and elementary. In general, however, the TCCs can be hard or even impossible to prove. In the latter case, this means that the specification is erroneous. PVS typechecking is undecidable, and the modeler needs to interactively prove theorems to typecheck a specification. Reasoning also proceeds by the modeler declaring, and then interactively proving lemmas about the specification. The result is a more powerful notation that allows arbitrary constraints to be expressed precisely and unambiguously using a high degree of abstraction.

For specifying ER models, we rely on the use of modeling constructs like functions and their various kinds (injections, partial functions, etc.). For example, it seems more natural to model certain weak entities using functions rather than relationships. In going from abstract to ER to schema, we apply data refinement to map uninterpreted types first to nested record types and then to flat record types. PVS implements data refinement via the principle of *theory interpretations*, an idea from universal algebra and logic in which the axioms of one theory are interpreted as theorems by another [31]. We specify the example model as three separate theories, each capturing on level of abstraction: the first is an abstract model in which entity, attribute and relation types are parameters. The second is an ER model defined by instantiating the abstract theory with concrete record types for entities and relations. The third is a schema model in which entities and relations are implemented as flat records. We then verify the correctness for the two mappings: from abstract to ER, and ER to schema. This allows one to formally state and prove that a data model at one level of abstraction implements another, more abstract model.

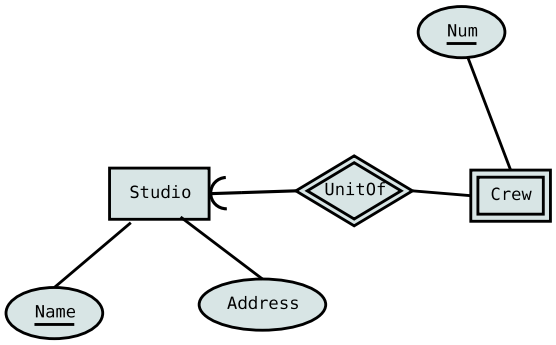

Figure 1: ER diagram slice of Ullman and Widom example [35, Chapter 2].

Outline of rest of the paper:

The rest of the paper consists of the following sections: Section 2 introduces an example model using an ER diagram and points to specific limitations of the diagram approach. Section 3 defines an abstract data model for the mini-example. Section 4 defines a record-based ER model for the mini-example. Section 5 presents a schema-based model of the mini-example. The next section, Section 6 addresses the issue of correctness of the mapping from the ER to the schema level. Section 7 discusses the results of the implementation: the sizes of the theories, and effort involved in proving type correctness conditions and user defined lemmas. Section 8 compares our work with existing approaches to data modeling in the literature. Section 9 discusses future work and Section 10 concludes the paper.

2. EXAMPLE ER MODEL

We pick the movies example data model from Ullman and Widom's introductory college text on databases to explore the approach of formally specifying and verifying a data model [35, Chapter 2]. Due to limited space, we focus on a self-contained slice of the example. The ER diagram of the slice is shown in Figure 1. The PVS specification of the complete example is discussed in an earlier technical report [12]. PVS source code for the complete example is available online [3].

The model consists of studio and crew entities. A crew is a unit of a studio. A studio has a name and address, whereas a crew has a number.

An ER model defines the *types* of attributes, entities and relationships. It also defines specific *sets* of entities and relationships *over* these types. The diagram does not distin-

guish between the types and the sets. When formalizing the model, however, we need to make the distinction explicit. Therefore, we use `Studio`, `Crew`, and `Unit_Of` for entity and relationship types, and `studios_set`, `crews_set`, and `unit_of_set` for entity and relationship sets, respectively.

2.1 Constraints

In addition to the entity and relationship types and sets, in the model shown in Figure 1, the following four constraints (of a total of twelve for the entire example) govern the model's semantics: key constraints (1) and (2), cardinality constraint (3), and referential integrity constraint (4).

1. The Name attribute is a key for `studios_set`.
2. The Num (number) attribute and `studios_set`, via the `unit_of_set`, together form a key for `crews_set`. (See cardinality constraint 3.)
3. `unit_of_set` is many-to-one from `crews_set` to `studios_set`.
4. For every element in `unit_of_set`, the constituent components belong to `studios_set` and `crews_set`.

In ER diagrams, key constraints are expressed by underlining key attribute names. `crews_set` is *weak*; its key is defined in terms of attributes from supporting relationships in which it participates. In ER diagrams, double borders identify weak entities and their supporting relationships. A many-to-one relationship has a round arrow on the edge at the "one" end the relation. Referential integrity means that the entity components of each element of a relationship set belong to their respective entity sets.

In the following sections, we build PVS theories for the example model, one for each level of abstraction: a parameterized abstract level, ER level, and the schema level. The type structure, entity set structure and the axioms used to specify the constraints vary for each level of abstraction.

3. ABSTRACT DATA MODEL

An abstract specification of a data model consists of the following kinds of objects:

- Attributes, entities, and relationship types, which are all uninterpreted. We call these *abstract entity types*.
- *Maps*, which connect abstract entity types. Some maps – those in correspondence with edges in the ER diagram, but oriented – are called *projectors*.
- *Abstract entity sets*, which are sets over the corresponding entity types.
- *Constraints*, which are specified as axioms over abstract entity sets, abstract entity types, and maps, and other functions.

The abstract specification is parameterized on the abstract entity types, maps, and the abstract entity sets.

Continuing the example, `movie_param_abstract` (Listings 3.1-3.7) is a specification of the abstract model of the example parameterized by abstract entity types (Listings 3.1), projectors (Listing 3.2), and abstract entity sets (Listing 3.3). The keyword TYPE+ posits that the types are non-trivial (i.e., nonempty). The abstract entity types and projectors together match the structure of the ER diagram in Figure 1.

Listing 3.1 (Abstract entity types).

```
movie_param_abstract[Name, Address, Num: TYPE+,
  Studio, Crew: TYPE+, UnitOf: TYPE+,
```

Listing 3.2 (Projectors).

```
studio_name:    [Studio -> Name],
studio_address: [Studio -> Address],
crew_num:       [Crew -> Num],
unit_of_crew:   [UnitOf -> Crew],
unit_of_studio: [UnitOf -> Studio],
```

Listing 3.3 (Abstract entity sets).

```
studios_set: set[Studio],
crews_set: set[Crew],
unit_of_set: set[UnitOf]]: THEORY
```

3.1 A theory for keys

In ER modeling, a *key* is an attribute or set of attributes that uniquely determine an element of an entity set. In our formalization of the abstract model, a key is identified not by an attribute name, but by a *key function*, which is often built using projectors that are injective. This is the natural way to model keys, since attributes of an entity are accessible using projector functions emanating from the entity.

Listing 3.4 (A theory for keys).

```
key[D:TYPE, S:set[D], R:TYPE, f:[D -> R]]: THEORY
  BEGIN
    ASSUMING
      restriction_is_injective: AXIOM
        injective?[(S), R]
          (restrict[D,(S),R](f))
    ENDASSUMING

    image_f_S: set[R] = image[D, R](f,S)
    I: TYPE = (image_f_S)
    h(s:(S)): I = f(s)

    h_is_bijective: LEMMA bijective?(h)
    getForKey: [I -> (S)] = inverse_alt(h)
    forKey(r: R):  lift[(S)] =
      IF (member(r,image_f_S))
        THEN up(getForKey(r)) ELSE bottom ENDIF
  END key
```

The theory for keys in Listing 3.4 defines the condition under which an abstract attribute entity type R is a key for uniquely identifying entities in a set S of elements of type D. The goal is to identify a key function that maps a key to a value in the entity set, if it exists. The function $f : D \to R$ is often a projector, retrieving an attribute in R from an entity in D. The elements of R qualify as keys provided the restriction of f to S is injective. The axiom in the theory captures this assumption. To see why this formulation implies the existence of a key function, let $I \subseteq R$ be the image of f on S. Since f restricted to S is injective, $h : S \to I$ defined as equal to f over S is a bijection. Therefore the function g from R to the lifted domain $S_\perp$ is a key function. g extends the bijective function $h^{-1} : I \to S$ to the domain R and range $S_\perp$. For an element $k \in R$, g maps k to $h^{-1}(k)$, if k is I, and to $\perp$ otherwise.

We now instantiate the `key` theory with different entity types and sets to obtain specific key constraints. The axiom in Listing 3.5, line 33 posits the injectivity of the restriction

Listing 3.5 (Key constraint on *studios_set*).

```
studio_name_injective_on_studios_set:
 AXIOM
injective?[(studios_set), Name](
 restrict[Studio, (studios_set),Name]
  (studio_name))

IMPORTING key[Studio, (studios_set),
  Name, studio_name]  AS studio_key

studio_for_name: [Name ->
 lift[(studios_set)]] =  studio_key.forKey
```

Listing 3.6 (Referential integrity of *unit_of_set*).

```
unit_of_ref_integrity: AXIOM
 FORALL (u: (unit_of_set)):
  member(unit_of_studio(u),studios_set)
  AND member(unit_of_crew(u),crews_set)
```

of the projection `studio_name` to `studios_set`. The axiom justifies the existence of a *key function* `studio_for_name` mapping names to the lifted domain of studios (line 42).

3.2 Referential integrity constraints

Referential integrity is specified in terms of projector functions and abstract entity sets. If $f : A \longrightarrow B$ is a projector from abstract entity type A to B, and a and b are, respectively, the entity sets of type A and B, then, referential integrity on the projector f emanating from the abstract entity type A of a is the property that $\forall x \in a.f(a) \in b$. Coming back to our example, the axiom on line 49 of Listing 3.6 is the referential integrity constraint on the projectors of `unit_of_set`.

3.3 Cardinality constraints

We consider the specification of the cardinality constraint on `unit_of_set` as an example.

In Listing 3.7, the image of the derived projector function `unit_of_crew_studio` on `unit_of_set` is used to define the binary relation `unit_of` (line 62). The cardinality constraint on `unit_of_set` boils down to declaring that the binary relation is a total function from `crews_set` to `studios_set`. This yields two projectors `crew_studio` and `crew_studio_num`.

3.4 Weak entities and foreign keys

Listing 3.8 shows how weak entities and foreign keys are specified at the abstract level. The injectivity of the derived projector `crew_studio_num` is used to axiomatize the key constraint on `crews_set` and yield the key function

Listing 3.7 (Cardinality constraint on *unit_of_set*).

```
unit_of_crew_studio(u:UnitOf):
 [Crew, Studio] =
   (unit_of_crew(u), unit_of_studio(u))

unit_of: set[[Crew,Studio]] =
 image(unit_of_crew_studio,unit_of_set)

function_unit_of:  AXIOM
 FORALL (cr: (crews_set)):
  exists1(LAMBDA(s: (studios_set)):
    unit_of(cr,s))
```

Listing 3.8 (Key constraint on *crews_set*).

```
crew_studio(cr: (crews_set)): (studios_set)
 = the(s:(studios_set) | unit_of(cr,s))

crew_studio_num(c:(crews_set)):
 [Studio, Num] =
  (crew_studio(c), crew_num(c))

crew_studio_num_injective_on_crews_set:
  AXIOM
   injective?[(crews_set), [Studio, Num]]
    (crew_studio_num)

% crew_for_studio_num  % key
```

Listing 4.1 (Attr. and Entity Types).

```
movie_rec: THEORY
  BEGIN
% Attribute Types
% ---------------
   NameEntity: TYPE+
   AddressEntity:  TYPE+
   NumEntity:  TYPE+

%  Entity and Relationship Types
%  -----------------------------
   StudioEntity: TYPE = [# name: NameEntity,
    address: AddressEntity #]
   studio_entity_name(s:StudioEntity)
    : NameEntity = s`name
   studio_entity_address(s:StudioEntity)
    : AddressEntity = s`address

   CrewEntity: TYPE =
    [# num: NumEntity, studio: StudioEntity #]
   crew_entity_num(c: CrewEntity)
    : NumEntity = c`num
   crew_entity_studio(c: CrewEntity)
    : StudioEntity = c`studio

   UnitOfEntity: TYPE =
    [# crew: CrewEntity, studio: StudioEntity #]
   unit_of_entity_crew(unit_of: UnitOfEntity)
    : CrewEntity  = unit_of`crew
   unit_of_entity_studio(unit_of: UnitOfEntity)
    : StudioEntity  = unit_of`studio
  END movie_rec
```

`crew_for_studio_num`. The entity set `crews_set` is weak; the projection function `crew_studio_num` involves `unit_of_set`, which is an abstract entity "foreign" to `crews_set`.

4. RECORD-BASED ER MODEL

At the abstract level discussed in the previous section, we do not distinguish between attribute, entities and relationship types. Nor is the internal structure of these types revealed. At the ER level, entity and relationship types are records. Attribute types, are, however, left uninterpreted because their structure has no role to play at this level. The record types may be nested, as in the case of relationship types. The record types for the example are defined in the theory `movie_rec` (Listing 4.1). Projectors now correspond to record selectors. The infix back quote operator in the PVS code indicates record selection.

4.1 Instantiating abstract to ER

The ER model is obtained from the parameterized abstract model by instantiating the abstract theory with suitable types, both concrete and abstract (uninterpreted). The theory `movie_er` (Listings 4.2–4.3) specifies the ER model

Listing 4.2 (Entity and Relationship sets).

```
movie_er: THEORY                                    14
 BEGIN                                              15
   IMPORTING props                                  16
   IMPORTING movie_rec                              17
                                                    18
    studios_entity_set:   set[StudioEntity]         19
    crews_entity_set:     set[CrewEntity]           20
    unit_of_entity_set:   set[UnitOfEntity]         21
```

Listing 4.3 (Instantiating *movie_param_abstract*).

```
   IMPORTING movie_param_abstract[                   23
    NameEntity, AddressEntity, NumEntity,           24
    StudioEntity, CrewEntity,  UnitOfEntity,        25
                                                    26
    studio_entity_name, studio_entity_address,      27
    crew_entity_num,                                28
    unit_of_entity_crew, unit_of_entity_studio,     29
                                                    30
    studios_entity_set, crews_entity_set,           31
    unit_of_entity_set]                             32
  END movie_er                                      33
```

for the example. First, the theory `movie_rec` containing record type defintions and another helper theory is included (Listing 4.2, line 17). Next, (Listing 4.2, lines 19–21), constants for entity and relationship sets are defined but their value is left unspecified, that is, they remain uninterpreted.

Finally (Listing 4.3), the theory `movie_param_abstract` is instantiated. As a result, the abstract types, projectors, abstract entity sets and constraints between them are all instantiated to use the record types and projectors of of `movie_rec` This completes the definition of the ER model. Note that the only things left unspecified are the uninterpreted attribute types (Listing 4.1) and the uninterpreted entity set constants (Listing 4.2). The correctness of the mapping of the abstract model to the record-based ER model is discussed in Section 6.1.

5. SCHEMA-LEVEL IMPLEMENTATION

The schema level types are flat (non-nested) record types. Sets over schema types are called *tables* in database parlance. All types at this level are concrete primitive types. The choice of what concrete types to use (primitive types such as varchars, integers, etc.) is a design decision that is specific to each schema implementation. In our example, we choose to implement the name and address attribute types as strings. The schema level specification of our example is given in the `movie_schema` theory (Listing 5.1, lines 20–22), which starts by grounding the attribute types. These types define an *interpretation* (lines 24-27) for the unspecified types in `movie_rec`. The choice of primitive types, however, does not affect the specification at this level. Schema types are defined as flat records (Listing 5.2). Note that some of the schema types, like `StudioSchema`, rely on already flat record types. Type refinement across the three levels is summarized in Table 1.

As a design decision, we choose to identify the schema types `UnitOfSchema` and `CrewSchema`. This optimization effectively eliminates the need for a separate `unit_of_table` (Listing 5.3).

Theory	Abstract	ER	Schema
Attributes	-	-	Primitive
Entities	-	Nested	Flat
Relationships	-	Nested	Flat

Table 1: Data type refinement in theories across levels of abstraction. Types are either uninterpreted (denoted by '-'), primitive, flat records, or nested records.

Listing 5.1 (Interpreting Attribute Types).

```
movie_schema: THEORY                                15
  BEGIN                                             16
    IMPORTING props                                 17
    IMPORTING function_results                      18
                                                    19
    NameP:      TYPE = string                       20
    AddressP:   TYPE = string                       21
    NumP:       TYPE = nat                          22
                                                    23
    IMPORTING movie_rec{{                           24
     NameEntity:= NameP,                            25
     AddressEntity:= AddressP,                      26
     NumEntity:= NumP}}                             27
```

Listing 5.2 (Schema Types).

```
StudioSchema: TYPE = StudioEntity                   29
                                                    30
studio_schema_name(                                 31
 s:StudioSchema): NameP = s`name                    32
                                                    33
CrewSchema: TYPE =                                  34
[# num: NumP, studio_name: NameP #]                 35
                                                    36
crew_schema_studio_name_num(                        37
 c: CrewSchema): [NameP,NumP] =                     38
  (c`studio_name, c`num)                            39
                                                    40
UnitOfSchema: TYPE = CrewSchema                     41
```

Listing 5.3 (Table Definitions).

```
studios_table: set[StudioSchema]                    45
crews_table: set[CrewSchema]                        46
unit_of_table: set[UnitOfSchema] = crews_table      47
                                                    48
% Derived Tables                                    49
% --------------                                    50
studio_names_table: set[NameP] =                    51
  image(studio_schema_name,(studios_table))         52
studio_name_crew_nums_table: set[[NameP,NumP]] =    53
```

Listing 5.4 (Key Constraint on *studios_table*).

```
studio_schema_name_injective: AXIOM                62
 injective?[(studios_table), NameP]                63
  (restrict[StudioSchema, (studios_table),         64
    NameP](studio_schema_name))                    65
                                                   66
%studios_entry_for_name: % key                     67
```

Listing 5.5 (Ref. Integrity of *unit_of_table*).

```
unit_of_table_ref_integrity: AXIOM                 87
 FORALL (u: (unit_of_table)):                      88
  member(u`studio_name, studio_names_table)        89
```

Listing 5.6 (Cardinality on *unit_of_table*).

```
studio_for_crew(cr: (crews_table))                 91
 : (studios_table) =                               92
 studios_entry_for_name(cr`studio_name)            93
                                                   94
unit_of: set[[(crews_table),                       95
 (studios_table)]] = graph(studio_for_crew)        96
                                                   97
          function_unit_of:  LEMMA                 98
            function?[(crews_table),               99
             (studios_table)](unit_of)             100
```

Listing 5.7 (Key Constraint on *crews_table*).

```
crew_schema_studio_name_num_injective:             106
 LEMMA                                             107
 injective?[(crews_table), [NameP, NumP]]          108
  (crew_schema_studio_name_num)                    109
                                                   110
% crew_entry_for_studio_num:  % key                111
```

5.1 Constraints

While the constraints on the conceptual ER model are predicates over entity sets, at the schema level, they are encoded as predicates over tables.

The key constraint on `studios_table` (Listing 5.4) axiomatizes the injectivity of `studio_schema_name` projector on `studios_table`. `studios_entry_for_name`, the resulting key function, is obtained like `studio_for_name` is in Listing 3.5.

Referential Integrity of `unit_of_table`: Because `crews_table` and `unit_of_table` are synonymous (Listing 5.3), the referential integrity for `unit_of_table` (Listing 5.5, lines 87–89) needs to specify the constraint only on the studio component of the `unit_of_table`. It is instructive to compare the definition of this constraint at the table level with the constraint `unit_of_ref_integrity` on `unit_of_set` (line 49 of Listing 3.6).

Cardinality constraint on `unit_of_table`: The function `studio_for_crew` (Listing 5.6, lines 91–93) is a composition of the key function `studios_entry_for_name` with the projector derived from the `studio_name` field. The cardinality constraint of `unit_of_table` is thus automatically satisfied (Listing 5.6, lines 98–100).

Key Constraints of `crews_table`: the projection function `crew_schema_studio_name_num` (Listing 5.2) is injective on `crew_table` because `CrewSchema` is defined in terms of a studio name and a number attribute.

Constraint	PVS Specification
1	`studio_name_injective_on_studios_set`: AXIOM (Listing 3.5) `studio_schema_name_injective`: AXIOM (Listing 5.4)
2	`crew_studio_num_injective_on_crews_set`: AXIOM (Listing 3.8) `crew_schema_studio_name_num_injective`: LEMMA (Listing 5.7)
3	`function_unit_of`: AXIOM (Listing 3.7) `function_unit_of`: LEMMA (Listing 5.5)
4	`unit_of_ref_integrity`: AXIOM (Listing 3.6) `unit_of_table_ref_integrity`: AXIOM (Listing 5.5)

Table 2: Specification of constraints across `movie` theories. For each row, the entry in the left cell refers to the constraint in English in Section 2.1. For the right cell, the upper entry is the constraint in the abstract model (Section 3) and the lower entry is the constraint in the schema model (Section 5).

.

Table 2 summarizes the different constraints of the mini-example. The constraints are specified at three levels: natural language (Section 2), and PVS specification in the abstract model and the schema-level model. Because of representation decisions made at the schema level (namely, identifying the implementation of `unit_of` table with that of the `crews_table`, some constraints expressed as axioms at the abstract level are lemmas at the schema level. In addition, the axiom `unit_of_table_ref_integrity`, combined with the equivalence of representation between `unit_of_table` and `crews_table` is strong enough to implement the axiom `unit_of_ref_integrity`, the integrity constraint for the `unit_of` abstract entity set.

6. THEORY INTERPRETATIONS AND THE CORRECTNESS OF MAPPING

We have seen how to specify a data model at three levels of abstraction. How are these models related, and in what sense is a data model valid with respect to another? We rely on PVS's notion of implementation between theories [31]. A data model A is *valid* with respect to a model B if the theory specifying model B provides an *implementation* of the theory specifying A. When A is valid with respect to B, we say there is a valid mapping from A to B. PVS has two separate, but related notions of implementation: instantiation and interpretation. Both of these are specified using the IMPORT keyword and used in the example specification.

6.1 Theory instantiation and abstract to ER

In PVS, a parametric theory A may be *instantiated* by a theory B using an 'IMPORT A' statement in B. This supplies actual arguments to the parametric types and constants of A. All of A's parameterized definitions and theo-

rems are available as instances in B, with the actual arguments to the parameters supplied in IMPORT statement in B. For B to correctly implement A, however, all the type correctness conditions, if any, generated by the IMPORT must be proved.

When `movie_param_abstract` is instantiated in the theory `movie_er` (Listing 4.3), no TCCs are generated. This is not entirely unexpected, since the record types and entity sets at the ER model level are obtained by a direct instantiation of the corresponding parameters at the abstract level. This establishes the correctness of the mapping from `movie_param_abstract` to `movie_er`.

6.2 Theory interpretation and ER to schema

In PVS, a theory A containing types, constant definitions, axioms and theorems may be *interpreted* by theory B if B provides an interpretation for the uninterpreted types and constants of A in such a way that the axioms of A may be interpreted as theorems in B. B thus becomes an "implementation" of A, demonstrating A's consistency with respect to B, provided the TCCs generated by the IMPORT in B of theory A are all proved.

To show that the schema model correctly interprets the ER model, we need to construct an interpretation for the uninterpreted types, constants (entity sets) and also prove as theorems the axioms in the ER model. This requires some effort since the schema model and the ER model operate at different but non-abstract type levels: ER models with nested records, and schema models with flat records.

Finally, the only uninterpreted objects in `movie_er` are the attribute types by virtue of importing `movie_rec`, and the entity sets (Listing 4.1). The schema model provides an interpretation for the attribute types (IMPORT statement in Listing 5.1). The parameter list to the import is a mapping *uninterpreted-constant* := *interpreted-value*. Next, we see how the interpretation of entity sets is constructed.

6.3 Entity construction

To build an interpretation for entity sets, we start by constructing an interpretation of the *entity elements* of the ER model using *entries*, which are elements of the tables in the schema model. Entity construction is done by defining a set of functions that construct an entity from a table entry (Listing 6.1). These functions are then used to interpret the entity sets (Listing 6.2) of the ER model. Recall that these entity sets were defined as uninterpreted constants in the ER model. Finally, the `IMPORT` statement of PVS is used to create an interpretation of the ER model's entity sets in terms of the tables in the schema model (Listing 6.3). Figure 2 illustrates the different notions of implementation (importing) used amongst the PVS theories in our example models.

6.4 Verifying type correctness conditions

The typechecking of the specifications and import statements in PVS automatically generates type correctness conditions. The theories `movie_param_abstract` and `movie_schema` generate one and five TCCs respectively. The theories `movie_rec` and `movie_er` generate no TCCs. The library theories (not shown) together generate three TCCs. None of these proofs are difficult to do. The completion of the proofs of the TCCs implies that the mapping between the ER model and the schema level is sound. Proof statistics for the PVS specifi-

Listing 6.1 (Entity Construction).

```
studio_instance_for_entry
 (s:(studios_table)):  StudioEntity = s

crew_instance_for_entry
 (c: (crews_table)): CrewEntity =
  LET n = c`num, sn = c`studio_name IN
   LET  se =
     studios_entry_for_name(sn)
    IN LET st =
     studio_instance_for_entry(se)
       IN (# num:= n, studio:= st #)

unit_of_instance_for_entry
 (u: (unit_of_table)): UnitOfEntity =
  LET cr = crew_instance_for_entry(u)
    IN LET st = crew_entity_studio(cr)
      IN (# crew:= cr, studio:= st #)
```

Listing 6.2 (Entity Sets from Tables).

```
studio_instances_set: set[StudioEntity] =
   studios_table

crew_instances_set: set[CrewEntity] =
 image(crew_instance_for_entry,
       crews_table)

unit_of_instances_set: set[UnitOfEntity] =
 image(unit_of_instance_for_entry,
       unit_of_table)
```

Listing 6.3 (Interpreting *movie_er*).

```
IMPORTING movie_er{{
 studios_entity_set := studio_instances_set,
 crews_entity_set   := crew_instances_set,
 unit_of_entity_set :=
  unit_of_instances_set}}
END movie_schema
```

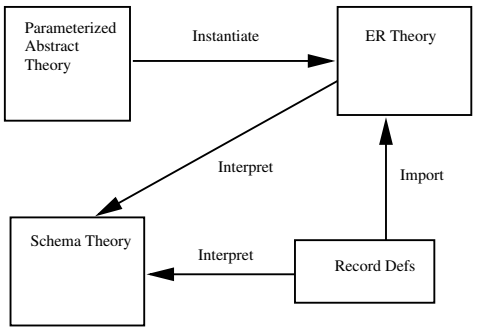

Figure 2: A high level view of the import relationships between different theories implementing the movie example. (Not all theories used are shown.)

Theory	Lines	TCCs	Lemmas
props	7	0	1
function_results	21	0	4
key	18	3	1
movie_rec	45	0	0
movie_er	32	0	0
movie_param_abstract	133	4	3
movie_schema	308	15	2
Total	**564**	**22**	**11**

Table 3: TCCs and user formulas in the different theories used to implement the complete movie example [3].

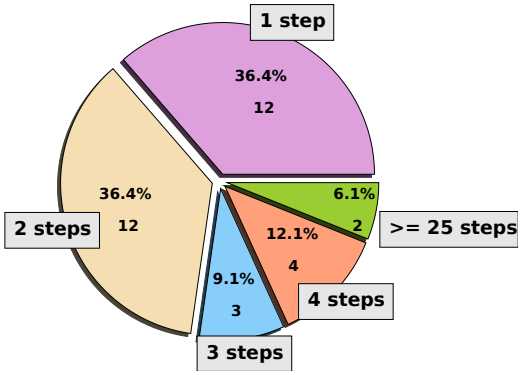

Figure 3: Distribution of the 33 proofs for the implementation of the full movie data model of Section 2 according to size (in number of user proof steps). All but two of the proofs are four steps or less.

cation of the complete Ullman and Widom example [3, 12] are shown in Section 7.

7. RESULTS

The number of lines of code, the number of TCCs generated, and the number of user formulas in each of the seven theories constituting the specification of the complete movie example [3, 12] are shown in Table 3. The abstract and schema specifications make up the bulk of the source code (441 lines out a total of 564). A total of 22 TCCs are generated. These are divided amongst the abstract and schema specifications, and the key library theory. The rest of the theories do not generate any TCCs, including `movie_er`. There is, on an average, about one tcc generated for every 30 lines of code. The specification also consists of 11 user-defined lemmas. Together with the TCCs, the total number formulas that need to be proved is 33. Not surprisingly, the bulk of the TCCs generated are for the theory `movie_schema` (15 of 22).

The distribution of the sizes of proofs of these 33 formulas is shown in Figure 3. All but two of them are of four or less steps in length and almost three-fourths are of length two or less. Fortunately, the two lemmas with much longer proofs (25 and 47 steps) are independent of the example model; they belong to library theories.

The results of Figure 3 encourage us to speculate that even as the number of model-specific constraints increase, the number of TCCs will increase, but not the sizes of their proofs. We expect that the number of generated TCCs to be proportional to the number of constraints in the model. We assume that the arity of relationships and the number of attributes on an entity is bounded. This implies that number of constraints varies linearly as the size of the ER diagram. This leads us to conjecture that the number of TCCs generated is at most linear in the size of the ER diagram of the model.

The proofs in our implementation all use only elementary proof steps and PVS's built-in strategies like GRIND. User-defined PVS proof strategies have not been used. Their use could further reduce the size of some of the longer proofs.

8. RELATED RESEARCH

Formalizing conceptual models for database applications was the original motivation for Codd's relational model and the conceptual ER model of Chen, which are both based on the theory of sets and relations [11, 13]. The relatively more recent object-oriented models [9] and object-relational models [32] also employ formal notations for their presentation.

Languages like Datalog are popular with the logic programming and deductive database community [10, 19]. Neumann and others use Datalog for building a framework for reasoning with data models [23, 30]. This approach relies on encoding instances, models and metamodels as Datalog programs. Integrity constraints are encoded as predicates and verification is done by querying these predicates for violations. However, Datalog is a highly restricted variant of Prolog and as such is only slightly more powerful than relational algebra and relational calculus, which form the core of the dominant databse query language, SQL. This restriction is because databases are often so large that even quadratic evaluation times are unreasonable. Datalog evaluations are thus explicitly decidable while PVS type-checking is not.

There are many papers recognizing the need for a formal approach to data modeling [6, 8, 33, 34]. This is also the case with modeling in related areas like object oriented software engineering and UML[7, 38]. There has also been work on the importance of conceptual models in the context of development [25], in business processes and business

intelligence [29], and in decision support systems [24].

Extensions to the ER model have been proposed with reasoning, semantics and constraint specification features [15]. Constraint specification has also been considered in the context of object-oriented databases and UML [21]. A generic specification process of diagram languages such as the ER model has also been reserached [28]. Specification languages are common in knowledge-based systems [17] and semantic databases [5]. Conceptual model-based verification and validation have also been researched in the context of specific applications such as diagnosis [36]. UML and OCL have been modeled formally using PVS [26]. In the ontology space, PVS has been used to formalize OWL and ORL specifications [14]. More recently, Mandelbaum et al. have proposed a data domain language PADS/ML for building a uniform automatic compilation framework of data models for data in ad hoc formats [27]. Their approach is promising, but is focused towards data analysis, and not modeling per se.

9. FUTURE WORK

This work is an initial step in the building of (semi) automated frameworks based on formal specification of data models. There are several directions for future work:

Automation: It should be relatively straightforward to automatically generate the PVS specification from an ER diagram. The second aspect of the automation involves generating automatic proofs of type correctness conditions and the correctness lemmas. Since most of the proofs involved a few steps, we expect that it should be possible to automate most, if not all of the proofs. This is a positive indication for building future tools based on this methodology.

Scaling: We have explored the approach with a small, text book example with about 12 constraints. Industry scale data modeling includes hundreds of constraints between dozens of entities and relationship types. We plan to use data models from industry case studies to investigate how our approach scales. The success of this scaling will be heavily dependent on the level of automation that can be achieved in generating the proofs of correctness and TCCs.

Trigger generation: Triggers are the practical implication of constraints. It should be possible to automatically translate constraints into triggers, which are tests that ensure the invariants are maintained at the end of every update to the database. However, while constraints are typically stated in terms of global properties, an efficient trigger should involve computation proportional to the size of the update to the database, not the size of the database itself.

Impact on design exploration: Model verification has an important role in allowing the designer to explore various design options during the modeling phase. in each case, the verification framework ensures that the design is explored within the boundaries of correctness. We plan to investigate how our framework supports such a correct-by-construction design methodology.

Working with other models: We have applied the specification language approach to traditional entity-relationship models of data. It should be interesting to consider formal specification of data models, like object models and their mapping to relations.

10. CONCLUSIONS

We have shown how data models may be specified and reasoned within PVS at different levels of abstraction. In particular, we have demonstrated how the support for higher-order functions, type checking, and interactive theorem proving in PVS allows the data modeler to reason about the interactions between the various data constraints. These are usually harder to do when using ER diagrams alone.

While design verification plays an important role in other disciplines (hardware and program verification), it has generally received less attention in ER data modeling. We believe this is due to the limited use of standard, formal notations and languages with verification support to express reasoning about data models. The work presented in this paper is a demonstration that general purpose specification languages like PVS with their powerful typechecking support can fill this gap.

Acknowledgements: We thank Sam Owre of SRI for patiently answering many of our queries on PVS.

11. REFERENCES

[1] The B-method. http://vl.fmnet.info/b/. Visited October 2007.

[2] PVS: Prototype Verification System. http://www.csl.sri.com/pvs. Visited October 2007.

[3] PVS source code accompanying [12] and this paper. http://www.iiitmk.ac.in/~choppell/research/code/movie-data-model/index.html Visited October 2007.

[4] The Z notation. http://vl.zuser.org/. Visited October 2007.

[5] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12:525–565, 1987.

[6] G. D. Battista and M. Lenzerini. Deductive entity-relationship modeling. *IEEE Trans. Knowl. Data Eng.*, 5(3):439–450, 1993.

[7] R. Breu, U. Hinkel, C. Hofmann, C. K. B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97 – Object Oriented Programming. 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

[8] D. Calvanese, M. Lenzirini, and D. Nardi. *Logics for Databases and Information Systems*, chapter Description Logics for Conceptual Data Modeling. Kluwer Academic, 1998.

[9] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[10] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases (Surveys in Computer Science)*. Springer, 1990.

[11] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–37, March 1976.

[12] V. Choppella, A. Sengupta, E. Robertson, and S. D. Johnson. Constructing and Validating Entity-Relationship models in the PVS Specification Language: A case study using a text-book example.

Technical Report 632, Indiana University Computer Science, April 2006.

[13] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 6(13):377–387, June 1970.

[14] J. S. Dong, Y. Feng, and Y. F. Li. Verifying OWL and ORL ontologies in PVS. In Z. Liu and K. Araki, editors, *1st International Colloquium on Theoretical Aspects of Computing (ICTAC) 2004*, volume 3407 of *LNCS*, pages 265–279. Springer, 2005.

[15] G. Engels, M. Gogolla, U. Hohenstein, K. Hulsmann, P. Lohr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modeling of database applications using extended ER model. *Data Knowledge Engineering*, 9:157–204, 1992.

[16] L. M. G. Feijs. Norman's database modularized in COLD-K. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II - theory, tools and applications*, volume 490 of *LNCS*, pages 205–229. Springer, 1991.

[17] D. Fensel. Formal specification languages in knowledge and software engineering. *The Knowledge Engineering Review*, 10(4), 1995.

[18] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. In C. H. D. Bjorner and H. Langmaack, editors, *VDM '90: VDM and Z - Formal Methods in Software Development*, volume 428 of *LNCS*, pages 189–210, 1990.

[19] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, 1984.

[20] C. George. The NDB database specified in the RAISE specification language. *Formal Aspects of Computing*, 4(1):48–75, 1992.

[21] M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[22] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[23] N. Kehrer and G. Neumann. An EER prototyping environment and its implemetation in a datalog language. In G. Pernul and A. M. Tjoa, editors, *Entity-Relationship Approach - ER'92, 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, October 7-9, 1992, Proceedings*, volume 645 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 1992.

[24] R. Kimball. Is ER modeling hazardous to DSS? *DBMS Magazine*, October 1995.

[25] C. H. Kung. Conceptual modeling in the context of development. *IEEE Transactions on Software Engineering*, 15(10):1176–1187, 1989.

[26] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. In *Proceedings of the Semantic Foundations of Engineering Design Languages (SFEDL '04)*, pages 39–47, 2005.

[27] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A Functional Data Description Language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 77–83. ACM Press, January 2007.

[28] M. Minas. Specifying diagram languages by means of hypergraph grammars. In *Proc. Thinking with Diagrams (TwD'98)*, pages 151–157, Aberystwyth, UK, 1998.

[29] L. Moss and S. Hoberman. The importance of data modeling as a foundation for business insight. Technical Report EB4331, NCR, November 2004.

[30] G. Neumann. Reasoning about ER models in a deductive environment. *Data and Knowledge Engineering*, 19:241–266, June 1996.

[31] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI International, April 2001.

[32] M. Stonebraker and D. Moore. *Object Relational DBMSs: The Next Wave*. Morgan Kaufmann, 1995.

[33] A. ter Hofstede and H. Proper. How to formalize it? formalization principles for information systems development methods. *Information and Software Technology*, 40(10):519–540, 1998.

[34] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer-Verlag, 2000.

[35] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 2 edition, 2002.

[36] F. van Harmelen and A. ten Teije. Validation and verification of conceptual models of diagnosis. In *Proceedings of the Fourth European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV97)*, pages 117–128, Leuven, Belgium, 1997.

[37] A. Walshe. *Case Studies in Systematic Software Development*, chapter NDB: The Formal Specification and Rigorous Design of a Single-User Database System. Prentice Hall, 1990.

[38] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise modeling with UML*. Object Technology Series. Addison Wesley, 1998.

[39] N. Winterbottom and G. Sharman. NDB: A non-programmer database facility. Technical Report TR.12.179, IBM Hursley Laboratory, England, September 1979.