Constructing and Validating Entity-Relationship Data Models in the PVS Specification Language: A case study using a text-book example

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

Arijit Sengupta
Raj Soin School of Business
Wright State University, USA

Edward L. Robertson
School of Informatics, Indiana University, USA

Steven D. Johnson
Computer Science Department, Indiana University, USA

April 11, 2006

Abstract

Data Modeling frameworks like the Entity-Relationship (ER) approach are usually specified using graphical and natural language representations. This limits the ability to formally express and verify the consistency of constraints on data models. The use of mathematical notation makes the specification precise, but also complex and tedious to write, and, in the absence of automated support for validation, error prone.

We use the PVS specification language and its theorem proving environment to formally construct, reason with, and mechanically validate an example data model at various levels of abstraction. The methodology proposed here makes modeling resemble programming in a strongly typed language. Models are implemented as PVS theories consisting of type declarations, function definitions, axioms and theorems. Entities and relationships are expressed as types. Constraints on the data model are expressed as axioms relating entity and relationship sets. Additional correctness conditions are generated by PVS's type checker. Using the theory interpretation mechanism of PVS, we prove the correctness of the example's logical model with respect to its ER model.

The example model we consider has about fifteen attributes, entities and relationships, and twelve constraints. The complete hand-coded specification of the model is about 600 lines of PVS (including libraries). Verification of the correctness of the model reduces to interactively proving about thirty correctness conditions. The proofs of almost all of these are quite small (4 steps or less). With modest additional effort, it should be possible to automatically generate the specification and proofs, paving the way for automatic verification of data models. We see our work as the initial step towards this goal.

Contents

1 Introduction

The modeling of data at a conceptual level and its subsequent modeling at the logical level are essential prerequisites to the design of any database, irrespective of the technology used to implement it. In the design of relational database systems, conceptual models of data typically employ a graphical notation based on ER diagrams. While diagrams are easier to understand, their vocabulary and expressive power are limited by the difficulty in extending the graphical notation in a reasonably standard manner. Designers therefore employ natural language to express nontrivial constraints, like referential integrity, for example. Natural language complements diagrammatic notation, but it is often the source of inaccuracies and ambiguities in specifications. This makes it difficult to use natural language to *formally reason* about the model and explore the design space within the boundaries of correctness.

This paper presents a design methodology which uses a formal specification language to support conceptual and logical data modeling. The result is a more powerful notation that allows arbitrary constraints to be expressed, a high degree of abstraction in the specification, and the advantage accrued by using typechecking to guarantee correctness of construction of the model. Using this approach, we show how the correctness of the logical model with respect to the conceptual model may be *formally* verified. To the best of our knowledge, this is the first successful attempt at representing a data model formally in a specification language *and* using typechecking to mechanically verify the correctness of its logical part with respect to its conceptual part.

1.1 Problem: constructing, validating and reasoning with database models

A *data model* is a conceptual representation of a real world enterprise built from analyzing the user's requirements. Data modeling is a fundamental prerequisite for the physical design and implementation of a database. It has three phases: conceptual, logical and physical. The conceptual modeling phase constructs a high level abstract representation of the structure of data. The logical phase builds a specification consisting of tables in a relational database, or classes and object structures in an object-oriented database suitable for implementation. The physical phase is concerned with the design of the storage structures and access methods needed for efficiently accessing and storing the data elements in the database. The first two phases – conceptual and logical – are more abstract; they can be done independent of the underlying software environments or hardware platforms.

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 in the form of schema, must conform. The most commonly used conceptual model is the *entity-relationship*

(ER) model [12] developed by Chen and discussed extensively in [4, 30]. Conceptual design in the ER model consists of a collection of *entities*, *attributes* of and *relationships* among those entities. The other part of the conceptual model specifies *constraints* between *instances* of entities and relationships.

A conceptual model lends itself to use in basic understanding and presentation of the underlying structure of the data. The importance of a conceptual model, however, goes beyond just the ability of presentation and use by managers and inexperienced developers. It is also a crucial means for assistance in the subsequent phases of development. Models, for example affect user behavior in writing queries in databases [14]. Several research studies show that user-database interaction at the conceptual level is less error-prone than at the logical or physical levels [10, 11, 27].

A good data modeling methodology should address the following three important design concerns: First, the methodology should allow the designer to *express* models at various levels of abstraction *and* support a framework to show how each level correctly implements the previous level. In particular, the modeling methodology should ensure the correctness of the logical model with respect to the conceptual model. Second, the modeling methodology should enable the designer to *reason* about the correctness of the model. Third, the methodology to allow the designer to *explore* design alternatives within the boundaries of correctness. The work of this paper shows how a specification language supported by a powerful typechecker can be used to address the first two of the above mentioned challenges. The problem of exploring design alternatives will be discussed in a future paper.

1.2 Motivation: The need for formal data modeling

How a data model is represented determines its usefulness. Traditional methodologies have employed diagrams to represent data models, which are built using a fixed set of graphical components connected by edges of different kinds. Simplicity in construction and understanding is the primary advantage of diagrams. This simplicity, however, comes at the cost of expressivity and integrity. The graphical notation allows only a simple set of constraints to be expressed between the model's various components. Examples include participation and cardinality constraints. Since the graphical notation is not extensible, more complex constraints need to expressed in natural language. For example, properties like referential integrity of a relation can not be expressed in the standard vocabulary of traditional ER diagrams. It should be noted, however, that the lack of precision is due to the chosen representation (graphical components) and not the modeling methodolgy itself.

Conceptual models can be represented in a precise manner using a more formal approach based on types, expressions, relations and constraints. There is an abundance of literature recognizing the need for such an approach to data modeling [5, 7, 29, 30] and modeling in related areas like object oriented software engineering and UML[6, 33] as well. There has also been work on the importance of conceptual models in the context of development [20], in the context of business processes and business intelligence [22] and in the context of decision support [19]. Formal modeling methodologies, however, are characterized by reasonably moderate mathematical notation. In the absence of automated support for validation and maintenance, the notation is prone to errors. This in turn limits the use of mathematical notation for conceptual designs in practice.

Like program development, the process of data modeling and design is incremental and iterative. At each stage, the design is extended, reasoned with, validated, and then further extended. Program developers typically use type checkers to reason with and validate their programs. Data modelers, however, have little support to validate their designs. Thus an environment that allows the construction, reasoning and validation of data models is needed.

1.3 Approach: Data modeling using a Specification Language

We use the specification language PVS and its theorem proving environment to formally construct, reason with, and interactively validate an example data model at various levels of abstraction. PVS, or Prototype Verification System is a general purpose specification language combined with a type checker closely integrated with a theorem proving environment [1, 24]. PVS's specification language is based on set theory and higher-order logic. Its type system is based on parametric and dependent types. Types, predicates and sets are treated uniformly in PVS. As a result, theorem-proving and typechecking are synonymous in PVS. Type checking in PVS is undecidable. As a consequence, the type correctness of a PVS specification is proved interactively by the user and the system. The flexible type system of PVS makes it convenient to state and prove constraints. PVS is widely used for specifying, reasoning with, and verifying a variety of systems: hardware and computer architectures, safety-critical computer systems, and requirements analysis.

Modeling with a specification language is like programming in a strongly typed environment. 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 a set of declarations that define types, functions, axioms and theorems. Data constraints are either explicitly expressed as axioms or implicitly using dependent types. The correctness of the specification often depends on the generation and validation of *type correctness conditions* (tcc's) that are automatically generated and interactively verified as theorems. The majority of the tcc's for the model discussed in this paper have proofs that are quite small and elementary, but in general they can be hard or even impossible to prove. In the latter case, this is usually taken to mean that there is a type error in the specification and the specification must be repaired. Reasoning also proceeds by the user declaring, and then interactively proving properties about the specification. Finally, the theory interpretation feature of PVS [25] allows one to formally state and prove that a data model at one level of abstraction *implements* another, more abstract model.

Although PVS was chosen for the purpose of specification, it should be noted that PVS is really a vehicle for formally specifying the model in what is essentially a "standard" notation of logic. The logical notation could be expressed in syntax other than PVS's, like the Object Constraint Language[33], or even plain mathematical notation, for example. The PVS theory implementing the model may therefore be considered as one of the several possible formal representations of the specification of the model. On the other hand, formal specifications written in PVS are *machine checkable* for correctness. Thus the PVS notation affords the designer the ability to automatically or semi-automatically verify the correctness of the model and its implementation. Other applications of such formal specifications include better presentation of the model semantics, as well as proving properties related to the data model.

1.4 Summary of work and outline of paper

The methodology proposed here should be seen as an initial step towards addressing the problem of automatically constructing, formally reasoning with, and verifying the correctness of data models. Our approach emphasizes the use of a specification language backed by powerful theorem proving technology to achieve this goal.

Using the specification language approach, we model an example data base enterprise at three distinct levels of abstraction, each specified as a PVS theory:

1. The first theory is a parametric "abstract" data model that captures the types and constraints specified in a traditional ER model without commiting to the concrete types of the attributes, entities or relationships. From a modeling point of view, these are abstractly specified parameters to the PVS theory.
2. The second level theory corresponds to a traditional ER model and is obtained by instantiating the abstract theory with concrete entity and relationship types.
3. The third theory corresponds to the logical model of the enterprise, consisting of tables.

Using the specification approach allows us to formally establish the soundness of the logical model with respect to the ER model using typechecking. Soundness is established by proving a set of correctness conditions generated in the process of typechecking the specification.

The main contributions of this work may therefore be summarised as follows:

1. **Specification of data models as parametric theories**: Our implementation shows how a data model may be formally specified at various levels of abstraction. The theories specifying the different models are parameterized on the types of the entitities, relationships and attributes, and constraints are stated as axioms.

2. **Correctness of implementation via typechecking**: We define a particular table structure implementing a conceptual data model and prove the correctness of the mapping from the conceptual to the table schema.

3. **Better insight into existing diagrammatic representations**. Implementing data model in a specification language provides a better understanding of existing diagram-based representations, which has important pedagogical value. For example, the formalism we use yields the following observations about an ER model and its diagram:

   (a) Each of the edges in an ER model diagram are more useful when interpreted as arrows (directed edges) denoting projection functions from the relation or entity to the entity or attribute, respectively.

   (b) The distinction between entity types and entity sets, not clear from the diagrammatic representation of the ER model, is made explicit in the model formalized in the specification language.

   (c) A key constraint may be thought of as a consequence of the injectivity of the projection function restricted to a particular entity set.

We illustrate our methodology using an example data model adapted from [31]. The model is implemented as a set of PVS theories. Our presentation of the example is very "code oriented." We believe that understanding the model requires careful scrutiny of the source code of the theories that make up the model. To make the paper self-contained, the entire source code (minus some comments) from the implementation is listed and discussed in the paper. Familiarity with PVS syntax is not assumed. The original source code (with comments) for the example model is available online [2].

Paper Roadmap

The rest of the paper consists of the following sections: Section 2 introduces the example model using an ER diagram and points to specific limitations of this approach. Section 3 defines a high-level specification of the data model in PVS. The next seven sections cover the implementation of the model in PVS in detail and form the bulk of the paper. The first two of these sections specify the abstract data model. Section 4 defines the attributes, entities and relationships in the abstract data model as uninterpreted types. Section 5 shows how constraints over abstract entity sets in the abstract model are specified as axioms. Section 6 shows how the abstract model is instantiated to obtain an ER model whose attribute, relation and entity types are concrete record types. The next four sections are devoted to developing the specification of the logical (relational) data model. Section 7 specifies the types used in the logical model. Section 8 defines the tables used in the logical model. Section 9 specifies the constraints in the logical model and also functions for reconstructing entity elements from table entries. Section 10 completes the specification by showing the correctness of the logical model with respect to the conceptual (ER) model defined in Section 6. Section 11 discusses the results of the implementation: the sizes of the theories, and effort involved in proving type correctness conditions and user defined lemmas. Section 12 compares our work with existing approaches in the literature. Section 13 discusses future work and Section 14 concludes the paper.

2 An Example Movie Data Model

The implementation discussed in this paper is based on the "movies" example used in [31, Chapter 2]. The diagram representing the ER model for the movies enterprise is shown in Figure 1.

2.1 Attribute, Entity and Relationship types

The ER model of the movie enterprise consists of four entity types: *stars*, *studios*, *movies*, and *crews*. Each of these types contains attributes. Each of *stars* and *studios* have a *name* and an *address*. A *movie* has a *title* and the *year* in which movie was made. A *crew* has a number. The movie enterprise also has four relationships: The *stars_in* relationship relates stars with movies. The *owns* relationship relates movies with studios. The *unit_of* relationship relates crews with studios. A crew is identified by its number and the studio of which it is a unit. The *contracts* relationship relates stars, movies, and studios. Attributes are drawn as circles, entities as boxes, and relationships as diamonds in ER diagrams.

2.2 Entity and Relationship sets and Constraints

The description in Section 2.1 refers to the *types* of the different attributes, entities and relations. Distinct from these are specific *sets* of entities and relations over these types. This distinction between entity types and entity sets is not explicitly made in traditional ER diagrams.

The entities and relationships of the movies enterprise are governed by a total of twelve constraints on the participating entity and relationship sets: four key constraints, two cardinality constraints, and six referential integrity constraints. The ER diagram fails to convey this information completely and precisely. Therefore it is complemented with an informal, but precise natural language specification of the constraints, which are given in the rest of this section.

2.3 Key Constraints

1. The attribute *name* is a key for the *stars* entity set.
2. The attribute *name* is a key for the *studios* entity set.
3. The combined attribute *title, year* is a key for the *movies* entity set.

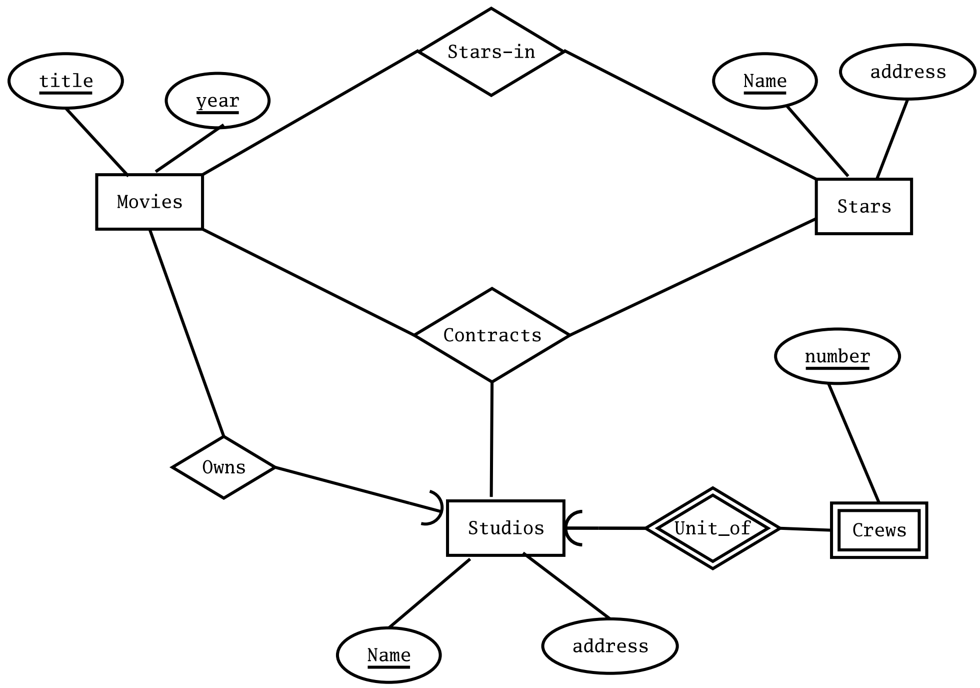

Figure 1: ER model diagram of the movie enterprise

4. The compound attribute obtained by combining the *number* attribute of *crew* with the *studio* element reached via the *unit_of* relation from an element of the *crews* entity set is a key for the *crews* entity. (See cardinality constraint 2 in Section 2.4.)

In ER diagram notation, key constraints are expressed by underlining key attribute names. *crews* is referred to as a *weak* entity because its key is defined in terms of attributes from supporting relationships in which *crews* participates. The weak entity and its supporting relationship are marked by double borders.

2.4 Cardinality constraints

The relationship and entity sets are governed by the following cardinality constraints:

1. Every element in the *movies* entity set is related to exactly one element in the *studios* set via the *owns* relationship, and that movie-studio pair is in the *owns* relationship set. In other words, *owns* set captures a many-to-one relationship from *movies* set to *studios* set.

2. Every element in the *crews* entity set is related to exactly one element in the *studios* set via the *unit_of* relationship, and that crews-studio pair is in the *unit_of* relationship set. In other words, *unit_of* set captures a many-to-one relationship from *crews* set to *studios* set.

In ER diagram notation, many-to-one relationships are indicated by edges with round arrows at the “one” end of the relation.

2.5 Referential Integrity Constraints

The entity components for every relationship set in the enterprise are drawn from their respective entity sets. Thus

1. For every element in the *stars_in* relationship set, the constituent components are drawn from *stars* and *movies* entity sets.

2. For every element in the *owns* relationship set, the constituent components are drawn from *movies* and *studio* entity sets.

3. For every element in the *unit_of* relationship set, the constituent components are drawn from *studios* and *crews* entity sets.

4. For every element in the *contracts* relationship set, the constituent components are drawn from *stars*, *movies* and *studios* entity sets.

5. The relation obtained by projecting the *star* and *movie* components from the *contracts* relationship set is a subset of the *stars_in* relationship set.

6. The relation obtained by projecting the *movie* and *studio* components from the *contracts* relationship set is a subset of the *owns* relationship set.

Note that referential integrity constraints are not explicitly indicated in the ER diagram notation.

3 High-level architecture of example data model

Specifications in PVS are written as a set of parameterized *theories*. The specification of the movie enterprise in PVS consists of four main theories and three auxiliary theories:

- `movie_param_abstract` : an abstract specification of the elements of the model. The abstract specification consists of two parts: the first is a type specification containing type and function declarations. The second is a constraint specification that defines constraints over types declared in the type specification.
- `movie_rec` : A record-based definition of *entities* over a set of uninterpreted "primitive" types.
- `movie_er` : The realization of the abstract specification as an ER model obtained by instantiating the parameters of `movie_param_abstract` with the types defined in `movie_rec` .
- `movie_schema` : A schema-based implementation shown to be correct with respect to the ER model in `movie_er` .
- `keys` : A parametric specification of key constraints.
- `props` : A helper theory with logical miscellany. (See Appendix A.)
- `function_results` : A helper theory dealing with propositions about functions. (See Appendix A.)

3.1 The PVS theory importing mechanism

In PVS, a theory consists of a set of related type and constant definitions, axioms, and theorems. Our implementation of the data model in PVS relies extensively on PVS's *theory import* mechanism, which comes in two forms, serving different, but related purposes. The two forms are distinguished by the different syntax for imports ("[...]" vs. "{{...}}") in PVS.

theory parameterization and instantiation : Theories in PVS may be *parameterized*, where the parameters may be types, or constants. A parameterized theory A may be imported by a theory B by an import statement in B. The import may be thought of as instantiating A with appropriate arguments matching the parameters of A. The parameterized theory A may include *assumptions* which need to be discharged as proof obligations by the importing theory B. An example of this in our implementation is the theory `key` (see Section 5.1 on page 16) which contains assumptions that are discharged when it is imported at various occasions in theories `movie_param_abstract` and `movie_er` (see for example, Sections 5.2 on page 19 and 9.1 on page 33).

theory interpretation : The theory interpretation mechanism of PVS allows the specification of an "abstract" theory by definining a set of uninterpreted types, constants and axioms. An abstract theory A is *implemented* by a concrete theory B when B imports A. The uninterpreted types and constants of the imported (abstract) theory A are provided definitions in the importing (concrete) theory B. The interpretation obligates the concrete theory to prove that it satisfies the axioms of the abstract theory.

Our implementation relies on both import mechanisms. The parametric theory `movie_param_abstract` (Section 4) defines an abstract ER model parameterized on attribute, entity and relationship types, entity sets and projection functions. The abstract theory contains constraints encoded as axioms (Section 5 on page 16). These axioms are defined using the types, constants and functions. The theory `movie_er` (Section 6 on page 26) imports `movie_param_abstract` by instantiating the parameters of `movie_param_abstract` with the types and functions defined in the theory `movie_rec` (Section 6.1). The theory `movie_er` still contains the axioms defined by `movie_er,` but these are now instantiated versions. Thus `movie_er` is a concrete instance of the more abstract data model specified in `movie_er.` `movie_er,` however, still contains constants which, while denoting entity and relationship sets, are left uninterpreted. In the second part of the implementation, the theory `movie_schema` imports the theory `movie_er` specifying the record-based ER model using the theory interpretation mechanism (Section 10 on page 41). During this import, the interpretation for the entity and relationship sets left uninterpreted in `movie_er` is supplied in `movie_schema.`

4 Abstract model: type specifications

We begin the specification of the abstract model by defining a theory `movie_param_abstract` parameterized by attributes, entities and relationships types. The code showing this parameterization is given in Listing 4.0.1 on the next page. All the above parameters are declared as *uninterpreted*, but non-empty types (indicated by the keyword `TYPE+`). There is no distinction made between attribute, entity, and relationship types. We collectively refer to these as *abstract entity types* or *abstract entities*. The eventual concrete realization of these abstract entities is irrelevant to this level of modeling. Thus we do not wish to commit to the actual representation (as datatypes like records, or lists, or tables) of objects at this level of modeling. We also defer the decision of what additional attributes an entity or relation may have in its eventual concrete form.

Listing 4.0.1 (Movie Types)

```
movie_param_abstract[
    Name, Address, Title, Year, Num: TYPE+,
    Star, Studio, Movie, Crew: TYPE+,
    StarsIn, Owns, UnitOf, Contracts: TYPE+,
```

The second set of parameters to the theory are functions between the various abstract types. These roughly correspond to the edges in the ER diagram model. We first consider projection functions on abstract entities, which project attributes from entities. These are declared in Listing 4.0.2.

Listing 4.0.2 (Functions on Entities)

```
    star_name:      [Star -> Name],
    star_address:   [Star -> Address],
    studio_name:    [Studio -> Name],
    studio_address: [Studio -> Address],
    movie_title:    [Movie -> Title],
    movie_year:     [Movie -> Year],
    crew_num:       [Crew -> Num],
```

Next, we declare projection functions on the abstract relationship entities which that we would like to eventually model as relationships. These functions are shown in Listing 4.0.3. The distinction between projection functions on entities and those on relationships is, strictly speaking, arbitrary at this stage of the modeling, but we will continue to make the distinction for the sake of convenience.

Listing 4.0.3 (Functions on Relationships)

```
    stars_in_star:  [StarsIn -> Star],
    stars_in_movie: [StarsIn -> Movie],

    owns_movie:  [Owns -> Movie],
    owns_studio: [Owns -> Studio],

    unit_of_crew:   [UnitOf -> Crew],
    unit_of_studio: [UnitOf -> Studio],

    contracts_star:   [Contracts -> Star],
    contracts_movie:  [Contracts -> Movie],
    contracts_studio: [Contracts -> Studio],
```

4.1 Abstract Entity Sets

Listing 4.1.1 on the following page identifies a set of *abstract entity sets* in the model. These are just sets over the abstract types defined in Listing 4.0.1. Like the abstract types and abstract projection functions, each abstract entity set is declared as a parameter to the theory.

Listing 4.1.1 (Abstract Entity Sets)

```
    stars_set:      set[Star],
    studios_set:    set[Studio],
    movies_set:     set[Movie],
    crews_set:      set[Crew],

    stars_in_set:   set[StarsIn],
    owns_set:       set[Owns],
    contracts_set:  set[Contracts],
    unit_of_set:    set[UnitOf]
 ]: THEORY
```

5 Abstract model: constraints

ER diagrams come equipped with a limited set of notational conventions to express a certain fixed class of constraints. Thus key attributes are underlined, n-to-m relations are indicated by annotating edges, and weak entities are indicated by double boxes. Arbitrary constraints, including specialized constraints on the types of attributes or entities, cardinality constraints etc. are traditionally expressed in natural language.

PVS allows the expression of arbitrary constraints as higher-order predicates over abstract entity sets. In the example we consider, however, we restrict ourselves to modeling key constraints, cardinality constraints and integrity constraints.

5.1 A parametric theory for keys

Key constraints are specified by instantiaing a theory for keys. The theory for keys, shown in Listing 5.1.1, defines the condition under which an abstract "attribute" entity of type R can be a key for uniquely identifying entities in a set S of elements of type D. The goal of this theory is to identify a *key function* that can be used to map a key to a value in the entity set, if it exists. The function $f : D \to R$ is usually a projection function, projecting an attribute in R from an entity in D. The elements of R can be used as keys provided the restriction of f to S is injective. This assumption is specified as an axiom in the theory. 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 to be identical to f over S is a bijection. Thus the function g from R to the lifted domain $S_\perp$ can be used as a key function, where g is obtained by extending 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.

Listing 5.1.1 (A theory for Keys)

```
25 key[D:TYPE, S:set[D],
26     R:TYPE, f:[D -> R]]: THEORY
27   BEGIN
28     ASSUMING
29       restriction_is_injective: AXIOM
30         injective?[(S), R]
31           (restrict[D,(S),R](f))
32     ENDASSUMING
33
34     image_f_S: set[R] = image[D, R](f,S)
35
36     I: TYPE = (image_f_S)
37     h(s:(S)): I = f(s)
38
39     h_is_bijective: LEMMA bijective?(h)
40
41     getForKey: [I -> (S)] = inverse_alt(h)
42
43     forKey(r: R):  lift[(S)] =
44       IF (member(r,image_f_S))
45         THEN up(getForKey(r))
46         ELSE bottom ENDIF
47   END key
```

The PVS theory `key` is parameterized on domain and range types `D` and `R`, a set `S` consisting of elements of type `D`, and a function `f` from `D` to `R`. The theory is further parameterized by the assumption (lines 28–32) that the function `restrict[D,(S),R](f)`, which denotes the restriction of `f` to `S`, is injective. In PVS, types have a set-theoretic semantics and a set may be converted to a type. The type expression `(S)` indicates the type obtained from the set `S`. `image[D,R](f,S)` denotes the image of `f` on `S` and is abbreviated `image_f_S`. This is turned into the type `(image_f_S)` and abbreviated `I`. The function `h` going from `(S)` to `I` is defined to coincide with `f`. The lemma `h_is_bijective` follows from the `injection_is_restrictive` axiom. The PVS proof consists of just two steps: a reference to the axiom followed by `(grind)`, which is an all-purpose simplification command. The function `g` is defined to be the inverse of `h` using the in-built higher-order function `inverse_alt`. The image of `f` restricted to `S` given by is defines a function `forKey`, which takes an element `r` from `R`, and returns an element from `S` uniquely determined by the key, if it exists, and `bottom` otherwise.

The typechecking of the theory generates three type correctness conditions, shown in Listing 5.1.2.

Listing 5.1.2 (TCC's for the *key* theory)

```
% Subtype TCC generated (at line 40, column 18) for  f(s)
```

```
    % expected type  I
  % proved - complete
h_TCC1: OBLIGATION FORALL (s: (S)): image_f_S(f(s));

% Assuming TCC generated (at line 44, column 28)
    % for inverse_alt
    % generated from assumption
    % function_inverse_alt.inverse_types
  % proved - complete
getForKey_TCC1: OBLIGATION
   (EXISTS (d: (S)): TRUE) OR (FORALL (r: I): FALSE);

% Subtype TCC generated (at line 48, column 26) for  r
    % expected type  I
  % proved - complete
forKey_TCC1: OBLIGATION
  FORALL (r: R): (member(r, image_f_S)) IMPLIES image_f_S(r);
```

The tcc's are obligations that the type checking process generates. The first is a verification condition that checks if the result $f(s)$ belongs to the set `image_f_S`. Its proof directly follows from the definition of `image_f_S`. The second is a technical condition generated as a consequence of the definition of the library theory `inverse_alt`. Its proof is straightforward from a case analysis of the emptyness and non-emptyness of the set `S`. The proof of the third condition follows directly from the definition of the predefined set membership predicate `member`. These proofs are easy enough for PVS to complete them automatically. The status `proved-complete` indicates that the proofs (and dependencies, if any, of the proofs) have been proved by PVS.

The PVS proof of the Lemma `h_is_injective` is stored as a lisp structure consisting of just two proof steps:

Listing 5.1.3 (PVS proof of *h_is_injective*)

```
((use "restriction_is_injective")
 (grind))
```

The first proof step commands invokes the `restriction_is_injective` Axiom, declared as part of the theory's assumptions (Listing 5.1.1, line 32). The next command `(grind)` instructs the theorem prover to apply its standard simplification rules. Figure 2 shows the PVS-generated human readable form of the sequent-style proof.

Specific uses of the `key` theory are obtained by by suitably instantiating the key constraints with different entity types and sets. These are discussed in the following subsections.

Verbose proof for `h_is_bijective`.
`h_is_bijective:`

{1}	bijective?(h)

`h_is_bijective:`

{1}	bijective?(h)

Using lemma restriction_is_injective,
`h_is_bijective:`

{-1}	injective?$[(S),\ R]$(restrict$[D,\ (S),\ R]$(f))
{1}	bijective?(h)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `h_is_bijective`.
Q.E.D.

Figure 2: Proof of Lemma h_is_bijective. The step `(grind)` of the stored lisp structure in Listing 5.1.3 corresponds to the last step of skolemization, instantiation and if-lifting.

5.2 Key Constraints for *stars_set*

The key constraint on *stars_set* states that two elements of *stars_set* are identical if they agree on their *star_name* value. This is easily captured as the constraint below:

Constraint 5.2.1 (Key constraint on *stars_set*) $\forall s_1, s_2 \in \mathit{stars\_set} : \mathit{star\_name}(s_1) = \mathit{star\_name}(s_2) \implies s_1 = s_2$

We are, however, interested in stating the constraint as a property of the function `star_name` :

Constraint 5.2.2 (Equivalent constraint on *stars_set*) *star_name restricted to stars_set is injective.*

Returning to the theory `movie_param_abstract,` in Listing 5.2.1, the key constraint `star_name_injective_on_stars_set` is defined as an axiom. This axiom states that the projection `star_name` is injective when restricted to the `stars_set` abstract entity. From the theory of keys defined in Section 5.1, it follows that the `Name` abstract attribute entity is a key for elements of the `Star` abstract entity in the `star_set` abstract entity set.

We define `star_key` as an instantiation of the theory `key` with the parameter list `[Star, (stars_set), Name, star_name].` Finally we define the star-specific key function `star_for_name` to be the `forKey` function of the instantiated theory `star_key.` The axiom `star_name_injective_on_stars_set` ensures that the importing theory `movie_param_abstract` satisfies the assumption `restriction_is_injective` of Listing 5.1.1, lines 28–32.

Listing 5.2.1 (Key constraint on *stars_set*)

```
star_name_injective_on_stars_set: AXIOM
  injective?[(stars_set), Name](
     restrict[Star,(stars_set), Name](star_name))

IMPORTING key[Star, (stars_set), Name, star_name]
  AS star_key

star_for_name: [Name -> lift[(stars_set)]] =
    star_key.forKey
```

The importing of theory `key` proceeds without the generation of additional tcc's.

5.3 Key Constraints for *studios_set*

The key constraint for `studios_set` is modeled in a way that is similar to `stars_set.` The constraint for `studios_set` is shown in Constraint 5.3.1, and its PVS specification is shown in Listing 5.3.1.

Constraint 5.3.1 (Key constraint on *studios_set*) *studio_name restricted to studios_set is injective.*

Listing 5.3.1 (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
```

The theory `key` is instantiated as `studio_key`. The key function `studio_for_name` is defined as the exported function `studio_key.forKey`.

5.4 Key Constraints for *movies_set*

The entity set *movies_set* has the pair (*title*, *year*) as a key. The abstract entity set *movies_set* uses the combination of the projection functions `movie_title` and `movie_year` as a key:

Constraint 5.4.1 (Key constraint on *movies_set***)** *The function*

$$\lambda m : Movie.\ (movie\_title(m), movie\_year(m))$$

is injective on movies_set.

The specification of this constraint is shown in Listing 5.4.1. The compound type `TitleYear` and projection function `movie_title_year` from the abstract entity `Movie` to `TitleYear` are used to define a key function. The rest of the specification is similar to the specification of key constraints for *stars_set* and *studios_set*. The `key` theory is instantiated as `movie_key`. The key function `movie_for_title_year` is defined as the exported function `movie_key.forKey`.

Listing 5.4.1 (Key constraint on *movies_set***)**

```
TitleYear: TYPE = [Title, Year]

movie_title_year(mv: Movie): TitleYear =
  (movie_title(mv), movie_year(mv))

movie_title_year_injective_on_movies_set: AXIOM
injective?[(movies_set), TitleYear](
    restrict[Movie, (movies_set),
             TitleYear](movie_title_year))

IMPORTING key[Movie, (movies_set),
              TitleYear, movie_title_year]
  AS movie_key

movie_for_title_year: [TitleYear -> lift[(movies_set)]] =
  movie_key.forKey
```

5.5 Referential Integrity Constraints for *stars_in_set*

Referential integrity and participation constraints typically apply to relationship sets. Referential integrity requires that for every element in a relationship set, every constituent element that is of an entity type must be present in the data model's entity set of that type.

We start with the referential integrity constraint for *stars_in_set*, stated below:

Constraint 5.5.1 (Referential Integrity for *stars_in_set***)** $\forall s \in stars\_in\_set : stars\_in\_star(s) \in stars\_set \ \wedge \ stars\_in\_movie(s) \in movies\_set.$

The PVS rendition of Constraint 5.5.1 is shown in the `stars_in_ref_integrity` axiom in Listing 5.5.1. The function `stars_in_stars_movie` projects pairs of type `[Star, Movie]` from elements of type `[StarsIn]`. The image of this function on `stars_in_set` is defined to be the binary relation `stars_in`, which in traditional conceptual ER modeling is treated as a basic relationship set. Here, however, `stars_in` is derived from the more abstract (and hence not necessarily binary) entity `stars_in_set`.

Listing 5.5.1 (Referential Integrity for *stars_in_set***)**

```
stars_in_ref_integrity: AXIOM
  FORALL (sm: (stars_in_set)):
    member(stars_in_star(sm),stars_set) AND
    member(stars_in_movie(sm),movies_set)

stars_in_star_movie(si:StarsIn): [Star, Movie]=
  (stars_in_star(si), stars_in_movie(si))

stars_in: set[[Star, Movie]] =
  image(stars_in_star_movie, stars_in_set)
```

5.6 Referential Integrity Constraints for *owns_set*

The referential integrity constraint for `owns_set` is similar to the constraint for `stars_in_set`:

Constraint 5.6.1 (Referential Integrity for *owns_set***)** $\forall o \in owns\_set : owns\_studio(o) \in studio\_set \wedge owns\_movie(o) \in movies\_set$

The PVS rendition of Constraint 5.6.1 is shown in the `owns_ref_integrity` axiom in Listing 5.6.1. The function `owns_movie_studio` projects pairs of type `[Movie, Studio]` from an element of type `[Owns]`. The image of this function on `owns_set` is defined to be the binary relation `owns`.

Listing 5.6.1 (Referential Integrity for *owns_set***)**

```
owns_ref_integrity: AXIOM
  FORALL (own: (owns_set)):
    member(owns_studio(own),studios_set) AND
    member(owns_movie(own),movies_set)

 owns_movie_studio(o:Owns): [Movie, Studio] =
   (owns_movie(o), owns_studio(o))

 owns: set[[Movie,Studio]] =
   image(owns_movie_studio, owns_set)
```

5.7 Cardinality Constraint for *owns_set*

There is an additional constraint on *owns_set*: every movie in *movies_set* is owned by exactly one studio which also occurs in *studios_set*. This constraint is captured in Listing 5.7.1 by the `function_owns` axiom. A consequence of this axiom is the function `studio_for_movie,` which maps an element `m` of `movies_set` to `the` (unique) element related to it by the `owns` relation.

Listing 5.7.1 (Many-to-one cardinality constraint on *owns_set*)

```
function_owns:  AXIOM
  FORALL (m: (movies_set)):
    exists1(LAMBDA(s: (studios_set)): owns(m,s))

studio_for_movie(m: (movies_set)): (studios_set) =
  the(s:(studios_set) | owns(m,s))
```

The definition of `studio_for_movie` spawns a tcc which is easily proved in four steps using the `exists1_singleton?_equivalence` lemma from the library theory `props` listed in Appendix A.

5.8 Referential Integrity Constraints for *contracts_set*

The referential integrity constraint for *contracts_set*, is a collection of three independent constraints. They are listed as follows:

Constraint 5.8.1 (Referential Integrity for *constraints_set*) $\forall c \in$ *constraints_set:*

1. $contracts\_star(c) \in stars\_set \wedge contracts\_studio(c) \in studios\_set \wedge contracts\_movie(c) \in movies\_set$
2. $(contracts\_star(c), contracts\_movie(c)) \in stars\_in$
3. $contracts\_studio(c) = studio\_for\_movie(contracts\_movie(c))$

(1) states that for every element c of the abstract entity set *contracts_set*, the projections of c are present in the respective abstract entity sets. (2) states that the pair (`contracts_star` (c), `contracts_movie` (c)) is contained in the binary relation `stars_in` , defined earlier in Listing 5.5.1. (3) states that `studio` and `movie` components of `contracts_set` are related via the `studio_for_movie` function defined in Listing 5.7.1.

These constraints are captured by three axioms in Listing 5.8.1.

Listing 5.8.1 (Referential Integrity for *contracts_set*)

```
contracts_ref_integrity: AXIOM
  FORALL (c: (contracts_set)):
    member(contracts_star(c),stars_set) AND
    member(contracts_studio(c),studios_set) AND
    member(contracts_movie(c),movies_set)

contracts_star_movie(c: Contracts):
   [Star, Movie] =
   (contracts_star(c), contracts_movie(c))
```

```
  contracts_star_movie: set[[Star, Movie]] =
     image(contracts_star_movie, contracts_set)

  contracts_stars_in_ref_integrity: AXIOM
    subset?(contracts_star_movie, stars_in)

  contracts_owns_ref_integrity: AXIOM
    FORALL (c: (contracts_set)):
      contracts_studio(c) =
        studio_for_movie(contracts_movie(c))
```

5.9 *contracts_set* induces a function: An example of reasoning in the abstract model

From the `contracts_owns_ref_integrity` axiom, the *studio* component is uniquely determined by the *movie* component. This means that the pair consisting of the *star*, *movie* components projected from `contracts_set` uniquely determine the *studio* component. Equivalently, the (graph of the) binary relation `contracts` obtained by pairing the pair consisting of the *star* and *movie* components of `contracts_set` with the *studio* component of `contracts_set` is a function. This result is captured by the lemma `function_contracts` in Listing 5.9.1.

Listing 5.9.1 (*contracts* is a function)

```
  contracts_star_movie_studio(c: Contracts):
     [[Star,Movie], Studio] =
     (contracts_star_movie(c), contracts_studio(c))

   contracts: set[[[Star,Movie],Studio]] =
     image(contracts_star_movie_studio, contracts_set)

  studio_for_stars_in_relation(sm:[Star,Movie], std:Studio)
    :bool =
      stars_in(sm) AND std = studio_for_movie(sm`2)

   function_studio_for_stars_in_relation:  LEMMA
     function?(studio_for_stars_in_relation)

   subset_contracts_studio_for_stars_in_relation: LEMMA
    subset?(contracts, studio_for_stars_in_relation)

   function_contracts:  LEMMA
     function?(contracts)
```

The proofs of the Lemmas `subset_contracts_studio_for_stars_in_relation` and `function_studio_for_stars_in_relation` are one and three steps respectively and are not shown.

Lemma `function_contracts` follows from the two preceding lemmas `subset_contracts_studio_for_stars_in_relation` and `function_studio_for_stars_in_relation` and the lemma `subset_function` of the library theory `function_results` (listed in Appendix A). The PVS proof script is shown in Listing 5.9.2.

Listing 5.9.2 (Proof of *contracts* is a function)

```
((use "subset_contracts_studio_for_stars_in_relation")
 (use "function_studio_for_stars_in_relation")
 (use "subset_function[[Star,Movie],Studio]")
 (grind))
```

5.10 Referential Integrity and cardinality constraints for *unit_of_set*

Next, we consider the constraints on *unit_of* and its implication on the weak entity *crew.* The referential integrity constraint of `unit_of_set` states that both participating studio and crew elements must be drawn from the respective entity sets `studios_set` and `crews_set.` This is shown in Listing 5.10.1.

Listing 5.10.1 (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)

     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)
```

The cardinality constraint for *unit_of* requires that this relation be many-to-one: for every element drawn from `crew_set,` there is a unique element from `studios_set` related to it in the binary relation `unit_of.` This element is obtained by projecting the crew and studio components from `unit_of_set.` The constraint is specified by the axiom `function_unit_of.` This axiom allows the definition of a function `crew_studio` that maps `crews_set` to `studios_set.`

Listing 5.10.2 (Many-to-one from crew to studio in *unit_of_set*)

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

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

```

5.11 Keys for weak entities

The entity `crews_set` is weak. The weakness is witnessed by the key for `crews_set,` which involves the (unique) studio in `studios_set` obtained via the `crew_studio` function and the (unique) number obtained using the `crew_number` function. The PVS specification to build the key function for the entity set `crews_set` is given in Listing 5.11.1.

Listing 5.11.1 (Key constraint for weak entity set *crews_set*)

```
  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)

   IMPORTING key[(crews_set), (crews_set),
                 [Studio, Num], crew_studio_num]
     AS crew_key

   crew_for_studio_num: [[Studio, Num] -> lift[(crews_set)]]
     = crew_key.forKey
END movie_param_abstract
```

Along with Section 3, this section completes the specification of the type structure and constraints on the movie model listed in the theory `movie_param_abstract.` Typechecking of theory results in four easily provable tcc's. In addition, the reasoning part consists of three lemmas (see Table 4 on Page 46).

6 Record-based ER model

The data model introduced in Sections 3 and 5 was abstract because it was parameterized on the types of objects and functions between them. In this section, the abstract model is instantiated into a concrete ER model using a record representation for entities and the corresponding record projection functions. In the ER model, however, the types of the attributes are left uninterpreted, i.e., unspecified. This is because the concrete types of these have no role to play in the modeling at the ER level for our example.

6.1 Record types for Entities and Relationships

The first step in constructing the concrete ER model is to define record types corresponding to the entities and the relationships. This is done in the theory `movie_rec`. Note that the attribute types in this theory are left uninterpreted. Thus the ER model based on importing `movie_rec` is kept generic. The theory `movie_rec` shown in Listing 6.1.1.

Listing 6.1.1 (Uninterpreted types for Attributes)

```
movie_rec: THEORY

  BEGIN

% Attribute Types
% ---------------
   NameEntity:     TYPE+
   AddressEntity:  TYPE+
   TitleEntity:    TYPE+
   YearEntity:     TYPE+
   NumEntity:      TYPE+
   TitleYearEntity: TYPE = [TitleEntity,YearEntity]
```

Entities are defined as record types with projection functions corresponding to record selections (indicated by the infix back quote operator). The definition of these entities is shown in Listing 6.1.2.

Listing 6.1.2 (Entity Type Definitions)

```
%  Entity Types
%  ------------
   StarEntity: TYPE = [# name: NameEntity, address: AddressEntity #]
   star_entity_name(s:StarEntity): NameEntity = s`name
   star_entity_address(s:StarEntity): AddressEntity = s`address

   StudioEntity: TYPE = [# name: NameEntity, address: AddressEntity #]
   studio_entity_name(s:StudioEntity): NameEntity = s`name
   studio_entity_address(s:StudioEntity): AddressEntity = s`address

   MovieEntity: TYPE = [# title: TitleEntity, year: YearEntity #]
   movie_entity_title(s:MovieEntity): TitleEntity = s`title
   movie_entity_year(s:MovieEntity): YearEntity = s`year

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

Next, in Listing 6.1.3, we define relationships based on the entities. These relationships are implemented as nested record structures.

Listing 6.1.3 (Relationship Type Definitions)

```
   StarsInEntity: TYPE =
     [# star: StarEntity, movie: MovieEntity #]
   stars_in_entity_star(stars_in: StarsInEntity): StarEntity
    = stars_in‘star
   stars_in_entity_movie(stars_in: StarsInEntity): MovieEntity
    = stars_in‘movie

   OwnsEntity: TYPE = [# studio: StudioEntity, movie: MovieEntity #]
   owns_entity_studio(owns: OwnsEntity): StudioEntity = owns‘studio
   owns_entity_movie(owns: OwnsEntity): MovieEntity = owns‘movie

   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

   ContractsEntity: TYPE =
     [# star: StarEntity, movie: MovieEntity, studio: StudioEntity #]
   contracts_entity_star(contracts: ContractsEntity): StarEntity
     = contracts‘star
   contracts_entity_movie(contracts: ContractsEntity): MovieEntity
     = contracts‘movie
   contracts_entity_studio(contracts: ContractsEntity): StudioEntity
      = contracts‘studio
  END movie_rec
```

6.2 Instantiating the abstract model to obtain an ER model

The theory `movie_er` constructs an ER model in three stages. The first involves importing the theory `movie_rec`, which contains type definitions of the entities and relationships. In addition, a helper theory `props` used for proving type conditions is also imported. The code corresponding to this is shown in Listing 6.2.1.

Listing 6.2.1 (Relationship Type Definitions)

```
movie_er: THEORY

  BEGIN

   IMPORTING props
   IMPORTING movie_rec
```

In the next stage, shown in Listing 6.2.2, entity and relationship sets are defined. Note that these are defined as uninterpreted constants.

Listing 6.2.2 (Entity and Relationship sets)

```
    stars_entity_set:     set[StarEntity]
    studios_entity_set:   set[StudioEntity]
    movies_entity_set:    set[MovieEntity]
    crews_entity_set:     set[CrewEntity]

    stars_in_entity_set:  set[StarsInEntity]
    owns_entity_set:      set[OwnsEntity]
    contracts_entity_set: set[ContractsEntity]
    unit_of_entity_set:   set[UnitOfEntity]
```

In the final stage, shown in Listing 6.2.3, the theory `movie_param_abstract` is imported and instantiated with types and the entity and relationship sets defined earlier in the theory.

Listing 6.2.3 (Importing *movie_param_abstract*)

```
   IMPORTING movie_param_abstract[
  NameEntity, AddressEntity, TitleEntity,
  YearEntity, NumEntity, StarEntity,
  StudioEntity, MovieEntity, CrewEntity,
  StarsInEntity, OwnsEntity,
  UnitOfEntity, ContractsEntity,

    star_entity_name,  star_entity_address,
    studio_entity_name,  studio_entity_address,
    movie_entity_title, movie_entity_year,
    crew_entity_num,
    stars_in_entity_star, stars_in_entity_movie,
    owns_entity_movie, owns_entity_studio,
    unit_of_entity_crew, unit_of_entity_studio,

    contracts_entity_star, contracts_entity_movie,
    contracts_entity_studio,

    stars_entity_set, studios_entity_set,
    movies_entity_set, crews_entity_set,

    stars_in_entity_set,  owns_entity_set,
    contracts_entity_set, unit_of_entity_set]
  END movie_er
```

7 Relational Model: Types

The ER model specified in Section 6 on page 26 is implemented as a relational schema-based model. The implementation is defined as the theory `movie_schema` which defines a set of *schemas* and integrity constraints on *tables*, which are sets of instances of schemas. The implementation is divided into four parts. The first part (this Section) consists of the type definitions. The second part consists of table definitions (Section 8 on page 32). The third part defines axioms that capture the integrity constraints at the table level (Section 9 on page 33). Finally, the relational schema model is defined as an instantiation of the parametric ER model (Section 10 on page 41).

The type definitions for the primitive attribute types are shown in Listing 7.0.4. We choose to make concrete the attribute types which were left abstract in the abstract model. We could, however, have postponed this decision further, since the rest of the development of the theory `movie_schema` is agnostic to the actual choice of the type of the attributes. Traditionally, however, the choice of primitive types for attributes is made at the relational model level.

In the second part of Listing 7.0.4, the record-based implementation `movie_rec` is imported with the primitives defined earlier.

Listing 7.0.4 (Schema type definitions)

```
movie_schema: THEORY
  BEGIN
   IMPORTING props
   IMPORTING function_results

% Types
% -----

   NameP:     TYPE = string
   AddressP:  TYPE = string
   TitleP:    TYPE = string
   YearP:     TYPE = posnat
   NumP:      TYPE = nat

   IMPORTING movie_rec{{
    NameEntity:= NameP,
    AddressEntity:= AddressP,
    TitleEntity:= TitleP,
    YearEntity:= YearP,
    NumEntity:= NumP
}}

TitleYearP: TYPE = [TitleP,YearP]
```

The schema are defined in terms of the entities specified in `movie_rec.` The schema definitions are given in Listing 7.0.5.

Listing 7.0.5 (Entity Schema type definitions)

```
% Schemas
% -------

  StarSchema: TYPE = StarEntity
  star_schema_name(s:StarSchema): NameP = s‘name
  star_schema_address(s:StarSchema): AddressP = s‘address

  StudioSchema: TYPE = StudioEntity
  studio_schema_name(s:StudioSchema): NameP = s‘name
  studio_schema_address(s:StudioSchema): AddressP = s‘address

  MovieSchema: TYPE = MovieEntity

  movie_schema_title(s:MovieSchema): TitleP = s‘title
  movie_schema_year(s:MovieSchema): YearP = s‘year

  CrewSchema: TYPE = [# num: NumP,
                        studio_name: NameP #]
  crew_schema_num(c: CrewSchema): NumP = c‘num
  crew_schema_studio_name(c: CrewSchema): NameP = c‘studio_name
  crew_schema_studio_name_num(c: CrewSchema): [NameP,NumP] =
     (crew_schema_studio_name(c),
      crew_schema_num(c))
```

The schema definitions for relationships are presented next (in Listing 7.0.6).

Listing 7.0.6 (Relationship Schema type definitions)

```
  OwnsSchema: TYPE = [# studio_name: NameP, movie_title: TitleP,
                       movie_year: YearP #]

  owns_schema_studio_name(owns: OwnsSchema): NameP =
     owns‘studio_name
  owns_schema_movie_title(owns: OwnsSchema): TitleP =
     owns‘movie_title
  owns_schema_movie_year(owns: OwnsSchema): YearP =
     owns‘movie_year
  owns_schema_movie_title_year(owns: OwnsSchema):
    TitleYearP = (owns‘movie_title, owns‘movie_year)

 UnitOfSchema: TYPE = CrewSchema
 unit_of_schema_crew_num(unit_of: UnitOfSchema): NumP =
   unit_of‘num
 unit_of_schema_studio_name(unit_of: UnitOfSchema): NameP =
   unit_of‘studio_name
 unit_of_schema_studio_name_crew_num(unit_of: UnitOfSchema):
```

```
    [NameP,NumP] = (unit_of_schema_studio_name(unit_of),
            unit_of_schema_crew_num(unit_of))

  ContractsSchema: TYPE =  [# star_name: NameP,
                              movie_title: TitleP,
                              movie_year: YearP,
                              studio_name: NameP #]

   contracts_schema_star_name(contracts: ContractsSchema):
     NameP = contracts‘star_name
   contracts_schema_movie_title(contracts: ContractsSchema):
     TitleP = contracts‘movie_title
   contracts_schema_movie_year(contracts: ContractsSchema):
     YearP = contracts‘movie_year
   contracts_schema_movie_title_year(c: ContractsSchema):
     TitleYearP = (contracts_schema_movie_title(c),
                  contracts_schema_movie_year(c))

   contracts_schema_studio_name(contracts: ContractsSchema):
     NameP = contracts‘studio_name
```

8 Relational Model: Tables

The next part of the theory defines the tables as sets of elements of schema types. It is useful to also define a set of derived tables, or *views*, which are projections of the original tables. The PVS code for the table definitions is given in Listing 8.0.7.

Listing 8.0.7 (Table Definitions)

```
% Tables
% ------
  stars_table: set[StarSchema]
  studios_table: set[StudioSchema]
  movies_table: set[MovieSchema]
  crews_table: set[CrewSchema]
  stars_in_table: set[StarsInSchema]
  owns_table: set[OwnsSchema]
  contracts_table: set[ContractsSchema]
  unit_of_table: set[UnitOfSchema] = crews_table

% Derived Tables
% --------------
  star_names_table: set[NameP] =
       image(star_schema_name,(stars_table))

```

```
  studio_names_table: set[NameP] =
       image(studio_schema_name,(studios_table))

  movie_titles_table: set[TitleP] =
       image(movie_schema_title,(movies_table))

  movie_years_table: set[YearP] =
       image(movie_schema_year,(movies_table))

  movie_schema_title_year(mv: MovieSchema): TitleYearP =
      (movie_schema_title(mv), movie_schema_year(mv))

  movie_title_years_table: set[TitleYearP] =
       image(movie_schema_title_year,(movies_table))
  studio_name_crew_nums_table: set[[NameP,NumP]] =
       image(crew_schema_studio_name_num, (crews_table))
```

9 Relational Model: Constraints and Instance Reconstruction

While the constraints on the conceptual model were predicates over entity sets, in the relation model, constraints are predicates over tables. We consider the constraints on each of the tables corresponding to entity and relationship.

Along with each constraint, we also define conversion functions that reconstruct entity elements from table entries. These functions are then used to provide interpretations to the entity and relationship set identifiers of the ER model (Listing 6.2.2 on page 29). This interpretation is the link that establishes the correctness of the schema model of this section with respect to the ER model of Section 6 on page 26.

9.1 Key Constraints on stars_table

Key constraints on `stars_table` are specified by instantiating the `key` theory: The projection function `star_schema_name` is injective on `stars_table.`

Listing 9.1.1 (Key Constraints on *stars_table*)

```
star_schema_name_injective_on_stars_table: AXIOM
  injective?[(stars_table), NameP]
    (restrict[StarSchema, (stars_table),
              NameP](star_schema_name))

    IMPORTING key[StarSchema, (stars_table),
                  NameP, star_schema_name]  AS star_schema_key
```

```
162
163     maybe_stars_table_entry_for_name:
164       [NameP -> lift[(stars_table)]] =  star_schema_key.forKey
165
166     stars_table_entry_for_name:
167       [(star_names_table) -> (stars_table)] =
168         star_schema_key.getForKey
169
170 %  Instances for Stars Table Entries
171 %  --------------------------------
172    star_instance_for_stars_table_entry(s:(stars_table)):
173      StarEntity = s
```

The reconstruction of *star entities* from *star table entries* (Lines 172–173 in Listing 9.1.1 on the previous page) is trivial since both are represented identically.

9.2 Key Constraints on studios_table

The specification of key constraints on `studios_table` is similar to the constraints of *stars table* and is given in Listing 9.2.1. The reconstruction of *studio* entities is also similar to the reconstruction of *star* entities in Section 9.1 on the previous page.

Listing 9.2.1 (Key Constraints on *studios_table*)

```
studio_schema_name_injective_on_studios_table: AXIOM
  injective?[(studios_table), NameP]
    (restrict[StudioSchema, (studios_table),
              NameP](studio_schema_name))

    IMPORTING key[StudioSchema, (studios_table),
                  NameP, studio_schema_name] AS studio_schema_key

    maybe_studios_table_entry_for_name:
      [NameP -> lift[(studios_table)]] =  studio_schema_key.forKey

    studios_table_entry_for_name:
      [(studio_names_table) -> (studios_table)] =
       studio_schema_key.getForKey

%  Instances for Studios Table Entries
%  --------------------------------
   studio_instance_for_studios_table_entry
     (s:(studios_table)):  StudioEntity = s
```

9.3 Key Constraints on movies_table

The key constraint on `movies_table` is specified by declaring that the projection function `movie_scheme_title_year` when restricted to `movies_table` is injective. Again, because of the identical representation of movie entity elements and movie table entries, the reconstruction of movie entities from movie table entries is just the identity function.

Listing 9.3.1 (Key Constraints on *movies_table*)

```
    movie_schema_title_year_injective_on_movies_table: AXIOM
      injective?[(movies_table), TitleYearP](
        restrict[MovieSchema, (movies_table),
                 TitleYearP](movie_schema_title_year))

    IMPORTING key[MovieSchema, (movies_table),
                  TitleYearP, movie_schema_title_year]
      AS movie_schema_key

    maybe_movies_table_entry_for_title_year:

      [TitleYearP -> lift[(movies_table)]] =
      movie_schema_key.forKey

    movies_table_entry_for_title_year:
      [(movie_title_years_table) -> (movies_table)] =
      movie_schema_key.getForKey

%  Instances for Movies Table Entries
%  ---------------------------------
   movie_instance_for_movies_table_entry
     (m:(movies_table)): MovieEntity = m
```

9.4 Referential Integrity Constraints of stars_in_table

Like the key constraints, the referential integrity constraints of relationships in the conceptual model are mapped to the tables in the logical model. The referential integrity constraint on the `stars_in_table` specifies that every entry in the `stars_in_table` has its components drawn from the derived tables `star_names_table` and `movie_title_years_table` (these tables are defined in Listing 8.0.7 on page 32).

The code for reconstructing a *stars_in* entity element from the corresponding table element (lines 245–262 in Listing 9.4.1) is verbose but self-explanatory.

Listing 9.4.1 (Referential Integrity Constraints on *stars_in_table*)

```
235     stars_in_table_ref_integrity: AXIOM
236       FORALL (si: (stars_in_table)):
```

```
        member(stars_in_schema_star_name(si), star_names_table)
        AND
        member(stars_in_schema_movie_title_year(si),
               movie_title_years_table)

%  Instances for StarsIn Table Entries
%  -----------------------------------

  star_instance_for_stars_in_table_entry(si: (stars_in_table)):
   StarEntity =
    LET n = stars_in_schema_star_name(si) IN
      LET s = stars_table_entry_for_name(n) IN
        star_instance_for_stars_table_entry(s)

  movie_instance_for_stars_in_table_entry(si: (stars_in_table)):
   MovieEntity =
   LET ty = stars_in_schema_movie_title_year(si) IN
      LET m = movies_table_entry_for_title_year(ty) IN
        movie_instance_for_movies_table_entry(m)

  stars_in_instance_for_stars_in_table_entry
    (si:(stars_in_table)): StarsInEntity =
      (#
       star:= star_instance_for_stars_in_table_entry(si),
       movie:= movie_instance_for_stars_in_table_entry(si)
      #)
```

9.5 Referential Integrity and Cardinality Constraints for owns_table

The referential integrity constraints on `owns_table` specifies that every entry in the `owns_table` has its components drawn from the derived tables `movie_title_years_table` and `studio_names_table.` The cardinality constraint on `owns_table` requires that for each movie table entry there is exactly one studio table entry such that the two are related via the `owns` relation. This is stated by declaring that the `owns` relation is a function.

The constraints are shown Listing 9.5.1. Note the correspondence between these axioms and the referential integrity and cardinality constraint axioms for abstract entity set `owns_set` shown in Listing 5.6 on page 22 and Listing 5.7 on page 22.

Reconstruction of an `owns_set` element from the corresponding `owns_table` entry is done by extracting the *movie* and *studio* components and then combining them together to form a *owns* record. The code is shown on lines 293-309 in Listing 9.5.1.

Listing 9.5.1 (Referential Integrity and Cardinality Constraints on ***owns_table*****)**

```
    owns_table_ref_integrity: AXIOM
      FORALL (own: (owns_table)):
        member(owns_schema_movie_title_year(own),
               movie_title_years_table) AND
        member(owns_schema_studio_name(own),
               studio_names_table)

    owns(s:StudioSchema, m: MovieSchema): bool =
      member((# studio_name := studio_schema_name(s),
                movie_title := movie_schema_title(m),
                movie_year  := movie_schema_year(m) #),
             owns_table)

    function_owner: AXIOM
      FORALL (m: (movies_table)):
        exists1(LAMBDA(s: (studios_table)): owns(s,m))

    owner(m: (movies_table)): (studios_table) =
      the({s: (studios_table) | owns(s,m)})

    owner_for_movie_entry:
      [(movies_table) -> (studios_table)] = owner

%  Instances for Owns Table Entries
%  -------------------------------
    studio_instance_for_owns_table_entry
      (x: (owns_table)): StudioEntity =
        LET n = owns_schema_studio_name(x) IN
          LET s =  studios_table_entry_for_name(n) IN
            studio_instance_for_studios_table_entry(s)

    movie_instance_for_owns_table_entry
      (x: (owns_table)): MovieEntity =
      LET ty = owns_schema_movie_title_year(x) IN
        LET m = movies_table_entry_for_title_year(ty) IN
          movie_instance_for_movies_table_entry(m)

    owns_instance_for_owns_table_entry(x: (owns_table)): OwnsEntity =
     (# studio := studio_instance_for_owns_table_entry(x),
        movie  := movie_instance_for_owns_table_entry(x) #)
```

9.6 Referential Integrity Constraints of contracts_table

The three referential integrity constraints of `contracts_table` mirror the constraints on `contracts_set` specified in Section 5.8 on page 23.

The specification of the `contracts_table` constraints is shown in Listing 9.6.1. The first constraint `contracts_table_ref_integrity` states that for every entry c of `contracts_table`, its star, studio and movie components are drawn respectively from `stars_table`, `studio_table` and `movies_table`. For the second constraint, first, the function `contracts_schema_stars_in` is used to construct the triplet consisting of *star_name*, *movie_title* and *movie_year* from each entry of `contracts_table` to yield the set `contracts_table_stars_in` (line 332-333). The constraint `contracts_table_stars_in_ref_integrity` defined next states that this set is a subset of `stars_in_table`. The third constraint `contract_table_owns_ref_integrity` states that the set `contracts_table_owns` obtained by extracting the *studio, movie_title* and *movie_year* components from `contracts_table` using the function `contracts_schema_owns` is a subset of `owns_table`.

The rest of the code (lines 351-378) in Listing 9.6.1 is devoted to reconstructing `ContractEntity` elements from `contracts_table` entries.

Listing 9.6.1 (Constraints on *contracts_table*)

```
317      contracts_table_ref_integrity: AXIOM
318        FORALL (c: (contracts_table)):
319          member(contracts_schema_star_name(c),
320                 star_names_table) AND
321          member(contracts_schema_studio_name(c),
322                 studio_names_table) AND
323          member(contracts_schema_movie_title_year(c),
324                 movie_title_years_table)
325
326      contracts_schema_stars_in(c: ContractsSchema):
327        StarsInSchema =
328          (# star_name:= contracts_schema_star_name(c),
329             movie_title:= contracts_schema_movie_title(c),
330             movie_year:= contracts_schema_movie_year(c) #)
331
332      contracts_table_stars_in: set[StarsInSchema] =
333         image(contracts_schema_stars_in, contracts_table)
334
335      contracts_table_stars_in_ref_integrity: AXIOM
336        subset?(contracts_table_stars_in, stars_in_table)
337
338      contracts_schema_owns(c: ContractsSchema):
339        OwnsSchema =
340         (# studio_name := contracts_schema_studio_name(c),
341            movie_title := contracts_schema_movie_title(c),
342            movie_year := contracts_schema_movie_year(c) #)
343
344      contracts_table_owns: set[OwnsSchema] =
345         image(contracts_schema_owns, contracts_table)
346
```

```
    contracts_table_owns_ref_integrity: AXIOM
      FORALL (c: (contracts_table)):
       subset?(contracts_table_owns, owns_table)

%  Instances for Contracts Table Entries
%  -------------------------------------

   star_instance_for_contracts_table_entry
     (x: (contracts_table)): StarEntity =
        LET n = contracts_schema_star_name(x) IN
          LET s = stars_table_entry_for_name(n) IN
            star_instance_for_stars_table_entry(s)

   studio_instance_for_contracts_table_entry
     (x: (contracts_table)): StudioEntity =
        LET n = contracts_schema_studio_name(x) IN
          LET s = studios_table_entry_for_name(n) IN
            studio_instance_for_studios_table_entry(s)


   movie_instance_for_contracts_table_entry
     (x: (contracts_table)): MovieEntity =
        LET ty = contracts_schema_movie_title_year(x) IN
          LET m = movies_table_entry_for_title_year(ty) IN
            movie_instance_for_movies_table_entry(m)

    contracts_instance_for_contracts_table_entry
      (c: (contracts_table)): ContractsEntity =
     (# star:= star_instance_for_contracts_table_entry(c),
        movie := movie_instance_for_contracts_table_entry(c),
        studio:= studio_instance_for_contracts_table_entry(c)
      #)
```

9.7 Referential Integrity constraints for *unit_of_table*

The referential integrity constraint for `unit_of_table` is shown in Listing 9.7.1. Note that because the `unit_of_table` and `crews_table` are synonymous (Listing 8.0.7 on page 32), the referential integrity for `unit_of_table` needs to specify the constraint only on the studio component of the `unit_of_table.` It might be instructive to compare the definition of this constraint at the table level with the constraint on `unit_of_set` (Listing 5.10.1 on page 25).

Listing 9.7.1 (Referential Integrity and Cardinality Constraints on *unit_of_table*)

```
    unit_of_table_ref_integrity: AXIOM
      FORALL (u: (unit_of_table)):
```

```
        member(unit_of_schema_studio_name(u),
               studio_names_table)

    studio_for_crew(cr: (crews_table)): (studios_table) =
      studios_table_entry_for_name(crew_schema_studio_name(cr))

    unit_of: set[[(crews_table), (studios_table)]] =
      graph(studio_for_crew)

    function_unit_of:  LEMMA
      function?[(crews_table), (studios_table)](unit_of)
```

9.8 Key Constraints of crews_table

The key constraints on `crews_table` are specified by instantiating the `key` theory, as shown in Listing 9.8.1. The second part of the listing (lines 420-439) defines the functions for reconstructing *crew* and *unit_of* entities from their respective tables. `crew_instance_for_crews_table_entry` is used to reconstruct *crew* instances from *crews_table* entries. Note the use of the key function `studios_table_entry_for_name` and the studio instance extraction function `studio_instance_for_studios_table_entry` defined earlier in Listing 9.1.1 on page 33. The reconstruction of *unit_of* entries relies on the equivalence of representation of the `UnitofSchema` and `CrewSchema` types.

Listing 9.8.1 (Key Constraints on *crews_table*)

```
404      crew_schema_studio_name_num_injective_on_crews_table: LEMMA
405        injective?[(crews_table), [NameP, NumP]]
406         (crew_schema_studio_name_num)
407
408       IMPORTING key[(crews_table), (crews_table),
409                      [NameP, NumP], crew_schema_studio_name_num]
410         AS crew_key
411
412       maybe_crew_for_studio_num:
413        [[NameP, NumP] -> lift[(crews_table)]]
414           = crew_key.forKey
415
416       crew_entry_for_studio_num:
417          [(studio_name_crew_nums_table) -> (crews_table)]
418            = crew_key.getForKey
419
420   %  Instances for CrewsTable Entries
421   %  ---------------------------------
422
423       crew_instance_for_crews_table_entry
```

```
      (c: (crews_table)): CrewEntity =
       LET n = crew_schema_num(c),
           sn = crew_schema_studio_name(c) IN
         LET  se = studios_table_entry_for_name(sn) IN
           LET st = studio_instance_for_studios_table_entry(se) IN
             (# num:= n, studio:= st #)

%  Instances for UnitOf Table Entries
%  ---------------------------------

% Exploit the equivalence of UnitOfSchema and CrewSchema
  unit_of_instance_for_unit_of_table_entry
    (u: (unit_of_table)): UnitOfEntity =
      LET cr = crew_instance_for_crews_table_entry(u) IN
        LET st = crew_entity_studio(cr) IN
          (# crew:= cr, studio:= st #)
```

10 Relational Model: Correctness of Implementation

We rely on the theory interpretation mechanism of PVS to ensure the correctness of the theory `movie_schema` specifying the logical model with respect to the theory `movie_er` specifying the ER model. The automatic verification of the soundness of the implementation with respect to the ER model consists of the following steps, which we collectively refer to as the **implementation correctness roadmap**:

1. Mapping the types, entity sets and constraints of the abstract model to constraints in the relational model (Sections 7 to 9). A summary of these constraints is discussed in Section 10.1.

2. Providing an interpretation of the entity sets of the ER model in the relational model. This is discussed in Section 10.2 on the next page.

3. Importing the ER model from the logical model. This is discussed in Section 10.3 on page 44.

4. Proving the type correctness conditions generated during the typechecking of the specifications and by the import. This is discussed in Section 10.4 on page 45 and also in Section 11 on page 45.

10.1 Constraint specification at different levels of abstraction

Tables 1 to 3 summarize the different constraints of the movie enterprise. The constraints are specified at three levels: natural language (Section 2), axioms

Constraint	PVS Specification
Sec. 2.3 (1)	star_name_injective_on_stars_set : AXIOM (Listing 5.2.1 on page 20) star_schema_name_injective_on_stars_table : AXIOM (Listing 9.1.1 on page 33)
Sec. 2.3 (2)	studio_name_injective_on_studios_set : AXIOM (Listing 5.3.1 on page 20) studio_schema_name_injective_on_studios_table : AXIOM (Listing 9.2.1 on page 34)
Sec. 2.3 (3)	movie_title_year_injective_on_movies_set : AXIOM (Listing 5.4.1 on page 21) movie_schema_title_year_injective_on_movies_table : AXIOM (Listing 9.3.1 on page 35)
Sec. 2.3 (4)	crew_studio_num_injective_on_crews_set : AXIOM (Listing 5.11.1 on page 26) crew_schema_studio_name_num_injective_on_crews_table : LEMMA (Listing 9.8.1 on page 40)

Table 1: Specification of key constraints across movie theories. For each row, the left column entry refers to the definition of the constraint in English. The right column entry refers to the corresponding PVS constraints for the abstract conceptual model (Section 5) and the relational schema-based model (Section 9).

in the PVS specification of the abstract model (Section 5 on page 16), and axioms or lemmas in the PVS specification of the relational model (Section 9 on page 33). Note that because of representation decisions made at the relational level (namely, identifying the type UnitOfSchema with CrewSchema, some constraints expressed as axioms at the abstract level are lemmas at the relational level. (See row 4 of Table 1 and row 2 of Table 2 on the following page.) In addition, in row 3 of Table 3, axiom unit_of_table_ref_integrity, alongwith the equivalence of representation between unit_of_table and crews_table is strong enough to implement the axiom unit_of_ref_integrity specifying the referential integrity constraint of the unit_of abstract entity set.

10.2 Entity Sets from Tables

The next step of the implementation correctness roadmap is to define an implementation of the entity sets of the ER model in terms of the tables of the relational model. Recall that these entity sets were defined as uninterpreted constants in the ER model. The definitions of implementations of the entity sets is shown in Listing 10.2.1 on page 44. stars_entity_set, studios_entity_set and movies_entity_set are obtained directly as their table implementa-

Constraint	PVS Specification
Sec. 2.4 (1)	`function_owns` : AXIOM (Listing 5.7.1 on page 23) `function_owns` : AXIOM (Listing 9.5.1 on page 36)
Sec. 2.4 (2)	`function_unit_of` : AXIOM (Listing 5.10.2 on page 25) `function_unit_of` : LEMMA (Listing 9.7.1 on page 39)

Table 2: Specification of cardinality constraints across `movie` theories. For each row, the left column entry refers to the definition of the constraint in English. The right column entry refers to the corresponding PVS constraints for the abstract conceptual model (Section 5) and the relational schema-based model (Section 9).

Constraint	PVS Specification
Sec. 2.5 (1)	`stars_in_ref_integrity` : AXIOM (Listing 5.5.1 on page 22) `stars_in_table_ref_integrity` : AXIOM (Listing 9.4.1 on page 35)
Sec. 2.5 (2)	`owns_ref_integrity` : AXIOM (Listing 5.6.1 on page 22) `owns_table_ref_integrity` : AXIOM (Listing 9.5.1 on page 36)
Sec. 2.5 (3)	`unit_of_ref_integrity` : AXIOM (Listing 5.10.1 on page 25) `unit_of_table_ref_integrity` : AXIOM (Listing 9.7.1 on page 39)
Sec. 2.5 (4)	`contracts_ref_integrity` : AXIOM (Listing 5.8.1 on page 23) `contracts_table_ref_integrity` : AXIOM (Listing 9.6.1 on page 38)
Sec. 2.5 (5)	`contracts_stars_in_ref_integrity` : AXIOM (Listing 5.8.1 on page 23) `contracts_table_stars_in_ref_integrity` : AXIOM (Listing 9.6.1 on page 38)
Sec. 2.5 (6)	`contracts_owns_ref_integrity` : AXIOM (Listing 5.8.1 on page 23) `contracts_table_owns_ref_integrity` : AXIOM (Listing 9.6.1 on page 38)

Table 3: Specification of referential integrity constraints across `movie` theories. For each row, the left column entry refers to the definition of the constraint in English. The right column entry refers to the corresponding PVS constraints for the abstract conceptual model (Section 5) and the relational schema-based model (Section 9).

tions. The other entity sets are defined using the instance reconstruction functions.

Listing 10.2.1 (Entity Sets from Tables)

```
    stars_entity_set: set[StarEntity] = stars_table
    studios_entity_set: set[StudioEntity] = studios_table
    movies_entity_set: set[MovieEntity] = movies_table
    crews_entity_set: set[CrewEntity] =
      image(crew_instance_for_crews_table_entry,
            crews_table)

    stars_in_entity_set: set[StarsInEntity] =
      image(stars_in_instance_for_stars_in_table_entry,
            stars_in_table)
    owns_entity_set: set[OwnsEntity] =
      image(owns_instance_for_owns_table_entry, owns_table)
    contracts_entity_set: set[ContractsEntity] =
      image(contracts_instance_for_contracts_table_entry,
            contracts_table)
    unit_of_entity_set: set[UnitOfEntity] =
      image(unit_of_instance_for_unit_of_table_entry,
            unit_of_table)
```

10.3 Interpreting the ER model theory in the Relational model theory by importing

The third step of the implementation correctness roadmap is to specify, using the PVS import statement, the interpretation of the ER model theory `movie_er` by the relational model theory `movie_schema.` The import is shown in Listing 10.3.1. The parameter list to the import is a mapping consisting of elements of the form

uninterpreted-constant := interpreted-value

Here, *uninterpreted-constant*s denote entity sets in the ER theory `movie_er` , while *interpreted-value*s are their implementations in `movie_schema.`

Listing 10.3.1 (Importing theory *movie_er*)

```
  IMPORTING movie_er{{
    stars_entity_set:= stars_entity_set,
    studios_entity_set:= studios_entity_set,
    movies_entity_set:= movies_entity_set,
    crews_entity_set:= crews_entity_set,

    stars_in_entity_set:= stars_in_entity_set,
```

```
    owns_entity_set:= owns_entity_set,
    contracts_entity_set:= contracts_entity_set,
    unit_of_entity_set:= unit_of_entity_set}}
  END movie_schema
```

10.4 Type correctness conditions

The final step of the implementation correctness roadmap is to ensure the overall type correctness of all the theories involved in the specification. A summary of the number of type correctness conditions for each of the theories is shown in Section 11, Table 4. Not surprisingly, the bulk of the tcc's generated are for the theory `movie_schema` (15 of 22) There are, however, *no* tcc's generated due to the import statement itself, and this *is* surprising. This indicates that the type checker was able to successfully resolve all the type conditions for this import before it got to the import statement.

11 Results

The number of lines of code, the number of tcc's generated, and the number of user formulas in each of the seven theories constituting the specification of the movie model are shown in Table 4. The abstract and schema specifications make up the bulk of the source code (441 lines out a total of 564). A total of 22 tcc's 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 tcc's, including `movie_er` , which is somewhat unexpected. There is, on the average, about one tcc generated for every 30 lines of code. The specification also consists of 11 user-defined lemmas: four in the library theory `function_results` , three in `movie_param_abstract,` two in `movie_schema` and on each in `key` and `props.` Together with the tcc's, the total number formulas is 33.

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. The proofs of the remain two lemmas, `subset_function` of theory `function_results` and `exists1_singleton?_equivalence` of theory `props` , are of length 47 and 25, respectively. Fortunately, these moderately sized proofs are both from the library theories: all the proofs in the main (abstract, er, and schema) theories are small.

The results of Figure 3 encourage us to believe that even as the size of the specifications and number of constraints increase, the number of tcc's will increase, but not the sizes of their proofs. The proofs in our implementation all use only elementary proof steps; PVS proof strategies have not been used. Their use could reduce the sizes of some of the longer proofs.

Theory	# lines	TCC's	User Lemmas	Total Formulas (TCC's + Lemmas)
props	7	0	1	1
function_results	21	0	4	4
key	18	3	1	4
movie_rec	45	0	0	0
movie_er	32	0	0	0
movie_param_abstract	133	4	3	7
movie_schema	308	15	2	17
Total	564	22	11	33

Table 4: TCC's and User Formulas in the `movie` theories.

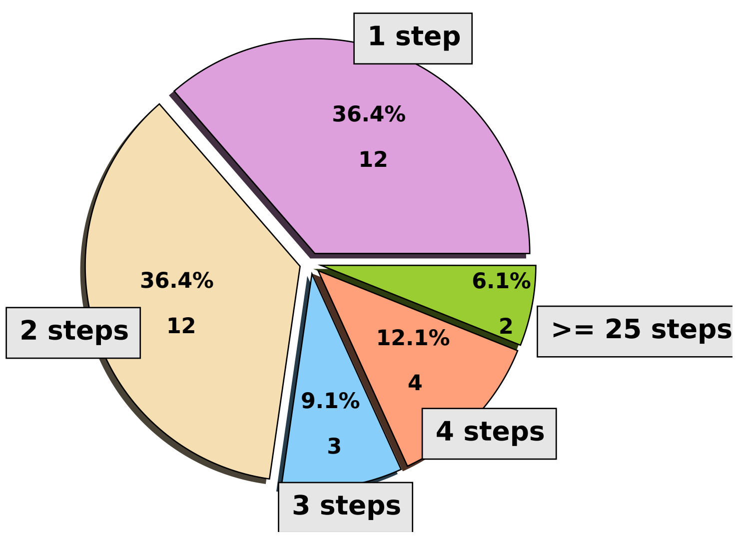

Figure 3: Distribution of the 33 proofs of the movie model according to size (in number of proof steps). All proofs are either of size four or less, or 25 or more.

12 Related Research

Research on conceptual models for database applications is not new. Since the early 70s, commonly accepted modeling approaches have been developed for databases, including the conceptual ER model [12] and the more physical relational model [13]. Since then, other models have been developed and used such as the object-oriented model [8] and the object-relational model [28]. One of the primary objectives for such models is to aid in the design of business data in applications for the purpose of automating business processes. These techniques were invented to "improve the quality of deliverables and to ensure that inexperienced system developers could follow repeatable SDLC processes.[22]" Typically, a conceptual model lends itself to easy migration to a logical and finally a physical model. The conceptual model can be used for presentation purposes, the logical model for formulating ad-hoc queries and the physical model for actual implementation in a database.

Deductive database languages like Datalog [9] have been used for reasoning with data models in the past by Neumann and others [23, 18]. Their 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. On the other hand, the methodology proposed in our work relies on the typechecking capabilities of a general-purpose specification language to prove correctness of the model and its implementation.

Although the importance of conceptual models in application design is well-accepted in literature, using a conceptual model for the purpose of specification is less common. There is some research in developing conceptual model through the process of specification [26]. Extensions to the ER model have been proposed with some amount of reasoning, semantics and constraint specification features [15]. Constraint specification is also researched in the context of object-oriented databases and UML [17]. A generic specification process of diagram languages such as the ER model has been researched by [21]. Specification languages are more common, however, in knowledge-based systems [16] and semantic databases [3], where the semantics of the data are more important than just mere structures. Finally, conceptual model-based verification and validation have also been researched, although more in the context of specific applications such as diagnosis [32].

Although different components of our work can be found in the literature on semantic databases and object-oriented databases, a coherent method for formal specification of conceptual models for the purpose of model verification still seems to be a necessity, something that we provide in this paper.

13 Future Work

Our work must be seen as an experiment in the evolution of a methodology that emphasizes correctness in data modeling and design. There are several directions in which this work can be extended.

13.1 Automation

While the size of our example was consisted of about fifteen or so elements (attributes, entities and relationships), the construction of its specification was relatively large (about 1000 lines of PVS code). Fortunately, it should be relatively straightforward to automatically generate the specification from the 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 and a judicious use of axioms of the model, it should not be difficult to automate most, if not all of the proofs. This is a positive indication for building future tools supporting this methodology.

13.2 Trigger Generation

Triggers are the the practical implication of constraints. It should be possible to automatically translate constriants 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 will involve computation proportional to size of the the update to the database, not the size of the database itself.

13.3 Modeling of more complex data

This work has applied the specification language approach to the more traditional Entity-Relationship modeling of data. The approach should be applicable to other data models like Object-Oriented or Object Relational, but we have no empirical evidence yet that this is indeed the case. Another interesting area is to formalize a data model of XML data. Again, we do not have a clear idea in what interesting ways the modeling of XML will impact the formalizaion.

14 Conclusions

The goal of this work was to show how data models may be constructed and validated using a formal specification language. We have shown, using a standard text-book example, how this is done. In the process, we have shown that design and modeling process can be carried out as a programming task in a strongly typed language with a reasonably sophisticated type checker at the backend.

While design verification plays in an important role in other disciplines (hardware and program verification), it has generally received less attention in data modeling. Data modeling is an important part of the requirements analysis phase of software engineering. We believe that this lack of emphasis is due to the absence of a design methodology that emphasize correctness and tools that support validation. The experiment presented in this paper is a small, but positive step towards creating such a methodology.

15 Acknowledgements

We thank Sam Owre of SRI for providing timely bug fixes to PVS and helping us understand the finer points of PVS's theory interpretation mechanism.

References

[1] PVS: Prototype Verification System. `http://www.csl.sri.com/pvs`.

[2] PVS Implementation of Ullman and Widom's Movie Data Model Example. `www.indiana.edu/db-group/movie-example`, 2005. see [31] for original example.

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

[4] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An entity-relationship approach*. Benjamin-Cummins, 1992. ISBN 0-8053-0244-1.

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

[6] 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 Springer LNCS, 1997.

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

[8] 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.

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

[10] H. C. Chan, K. K. Wei, and K. L. Siau. User-database interface: The effect of abstraction levels on query performance. *Management Information Systems Quarterly*, pages 441–464, December 1993.

[11] H. C. Chan, K. K. Wei, and K. L. Siau. An empirical study on end-users' update performance for different abstraction levels. *International Journal on Human-Computer Studies*, 41:309–328, 1994.

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

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

[14] P. De, A. Sinha, and I. Vessey. An empirical investigation of factors influencing object-oriented database querying. *Information Technology and Management*, 2(1):71–93, 2001.

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

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

[17] 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.

[18] 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.

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

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

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

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

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

[24] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

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

[26] K. L. Siau, H. C. Chan, and K. P. Tan. A CASE tool for conceptual database design. *Information and Software Technology*, 34(12), December 1992.

[27] K. L. Siau, H. C. Chan, and K. K. Wei. The effects of conceptual and logical interfaces on visual query performance of end users. In *Proc. International Conference on Information Systems (ICIS)*, pages 225–235, Amsterdam, The Netherlands, 1995.

[28] M. Stonebraker and D. Moore. *Object Relational DBMSs: The Next Wave*. Morgan Kaufmann, 1995. ISBN:1558603972.

[29] 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.

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

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

[32] 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.

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

A Additional Library Theories

Listing A.0.1 (*function_results*)

```
function_results[D,R: TYPE]: THEORY

  BEGIN
    x: VAR D
    y: VAR R
    f,h: VAR set[[D,R]]
    g: VAR [D -> R]


    dom(f): set[D] = image((LAMBDA(p:[D,R]): p‘1), f)
    cod(f): set[R] = image((LAMBDA(p:[D,R]): p‘2), f)

    function?(f): bool =
     FORALL(x: (dom(f))):
      exists1 (LAMBDA (y: (cod(f))): member((x,y),f))

    function?_function:  LEMMA
      function?(graph(g))

    subset_dom: LEMMA
      subset?(h,f) IMPLIES subset?(dom(h), dom(f))

    subset_cod: LEMMA
      subset?(h,f) IMPLIES subset?(cod(h), cod(f))

    subset_function: LEMMA
      subset?(h,f) AND function?(f) IMPLIES function?(h)

  END function_results
```

Listing A.0.2 (*props*)

```
props[T:TYPE]: THEORY

  BEGIN
    s: VAR T
    p: VAR pred[T]
    exists1_singleton?_equivalence:  LEMMA
     exists1(LAMBDA (s: T): p(s)) IFF  singleton?[T](s: T | p(s))
  END props
```