Verification of Object Relational Maps

Krishna K. Mehra Sriram K. Rajamani
Microsoft Research India

A. Prasad Sistla
University of Illinois at Chicago

Sumit K. Jha
Carnegie Melon University

Abstract

Enterprise software systems need to deal with two dominant data models. While object oriented languages (such as Java, C#, C++) are the dominant ways to write business logic, relational databases are the dominant ways to store data. Object-Relational (OR) maps are widely used to mediate between these two data models. We present a system to verify correctness of OR maps. We formulate simple correctness conditions for OR maps, and convert these conditions to validity of formulas in first order logic. We have built a verification tool called ROUNDTRIP *that is able to both validate and find errors in OR maps defined in the ESQL language of the Microsoft EDM data model.*

1 Introduction

Automated methods for verifying the correctness of computer systems have gained prominence in the last decade due to their successful application to certain classes of practical problems. Such classes include verification of hardware systems such as processors, pipelines and cache coherency protocols, and low-level software such as device drivers. The area of database software has been largely untouched by these developments partly due to the complexity of these systems. Mainstream application software, that interfaces with databases, needs to deal with both object oriented and relational data models. While object oriented languages such as Java and C# are in widespread use for writing business logic, relational databases continue to dominate persistent storage of data. Mediating between the object oriented and relational data models is an important problem [10, 7]. Object-Relational maps (OR maps) are the most common ways to do this mediation. In this paper, we describe methods based on theorem provers for verifying OR maps and describe a tool that is based on these methods. To the best of our knowledge, ours is the first effort in this area.

There are several ways to write or semi-automatically generate OR maps, and several tools and techniques have been developed [3]. Regardless of the technique used to generate the maps, OR maps can be specified using two queries (or views):

1. a query view, Q, that maps relations in the database to objects in the program, and
2. an update view, U, that maps objects in the program to relations in the database.

If the user writes an object into a database and reads it back, it is reasonable to expect to obtain the same value that was written. More formally, the compositions $Q \circ U$ and $U \circ Q$ need to be identity maps (with appropriate integrity constraints on input domains). This condition is called the round tripping condition.

In practice, round tripping is validated using testing. Thus, several test instances of data models are generated, either manually, or automatically, and the round tripping condition is checked on these instances. In this paper, we propose to use formal verification to validate round tripping of OR maps. We formulate the round tripping condition as validity of a first order logic formula. Then, we use techniques from first order logic theorem proving to check this formula. The approach has the same benefits as other successful uses of formal verification. In several cases, we can prove that an OR map works for all test inputs. In several cases, we can also automatically generate counterexamples which are test cases that do not satisfy the round tripping condition.

Any effort to formally verify OR maps encounters two major technical difficulties:

- In order to model object oriented features present in OR maps, the relational schema needs to be extended to include features such as complex types and inheritance, and the relational algebra needs to be extended with operators such as field reference, type casting, address and dereference. While translating standard relational algebra to first order logic is well known, translating such extensions is non trivial.

- While existing first order logic theorem provers are able to prove validity of formulas generated from correct OR maps, these resolution based theorem provers have difficulty in generating counter-models for formulas. Existing approaches for model generation for first-order logic can not scale for formulas arising from incorrect OR maps due to high arity of relations involved.

In this paper, we present techniques to overcome the above difficulties.

We present a small and simplified language called Extended Relational Algebra (ERA) to model object oriented extensions to relational algebra. We present a formal inductive translation of ERA to first order logic (FOL) and argue that our translation is sound. The translation uses some non-standard choices of encoding ERA relations to FOL relations. Every type in ERA becomes a relation in FOL. Such an encoding allows us to model inheritance and subtyping cleanly using implications between FOL relations in the type hierarchy. It also enables us to handle complex types, taking addresses of entities, and dereferencing addresses uniformly.

We tried to use off-the-shelf FOL theorem provers to validate the generated FOL formulas. While the provers work well for valid formulas, we found that they do not generate counter models for invalid formulas. Existing approaches to first order logic model generation, such as ALLOY [14] attempt to search for a small model by bounding the number of distinct constants in the model. In the presence of relations with large arities, this approach does not scale. With c constants, and a relation of arity a, they use c^a boolean variables, which does not scale for large values of a. Using the constant bounding approach, we find that we cannot even generate the SAT formulas for several of our examples. Thus, we were forced to write our own model generation tool using boolean SAT solvers. In our model generator, we also bound the maximum number of rows in our relations by k, in addition to bounding the number of constraints by c. This leads to formulas with $a \times k \times log(c)$ boolean variables. We find that this approach scales to handle our examples. Our experimental results indicate that we can indeed find counter-models with small number of rows (say 2 or 3) for all examples that we have encountered.

We have implemented a verification tool ROUNDTRIP, to check correctness of OR maps. Our system is able to handle object and relational schema's provided in Microsoft's Entity Data Model (EDM) language [2], and query and update views provided in Microsoft's Extended SQL (ESQL) [2] language. ROUNDTRIP is able to prove the correctness, or find counter-models for most examples we have tried in a few minutes. ERA is close to ESQL but has been simplified for readability. Although, we give the translation from ERA to FOL in this paper, ROUNDTRIP does a direct translation from ESQL to FOL.

There has been prior work on translating object oriented query languages to relational query languages and relational calculus [1, 24, 19, 23]. However, a practical query language such as ESQL has several object oriented features such as dereferencing, type casting, dynamic type checking, attribute renaming, and complex case splitting. Furthermore, these features can be nested within each other in very complex ways. In order to build a tool for verifying such a language, it is essential that use an *inductive* translation, i.e., a translation where we can walk up the Abstract Syntax Tree (AST) for the query and construct the formula of a node in the AST in terms of the formulas of the children. We believe that such a construction is quite non-trivial, and does not exist in the literature, to the best of our knowledge. For instance, we track bindings and existential quantifiers across several nodes in the AST, and handle them quite intricately. The inductive translation of ESQL to first order logic is a main contribution of our work.

We also believe that our techniques used in the translation to SAT formulas, for generating counter models, are new and have lower complexity in terms of the number of boolean variables as given earlier. Moreover, we provide formal semantics (for ERA) and the proof of correctness for our translation. As far as we know, this is the only formal proof of correctness for such complex translations. However, due to lack of space, we have not presented them in this paper. The formal semantics and proof of correctness are given in our technical report [18].

The paper is organized as follows. Section 2 motivates the problem with examples. Section 3 defines ERA and presents it semantics. Section 4 states the round trip problem formally. Section 5 gives the translation from ERA to FOL. Section 6 presents experimental results. Section 7 contains discussion of related work and future extensions.

2 Overview

We first illustrate issues in checking correctness of OR maps in a simpler setting with only relations, and without object oriented features. Suppose we have an entity set $Orders(id, amount)$, but two relations:

1. $SmallOrders(id, amount)$ for orders with amount less than \$100, and

2. $BigOrders(id, amount)$ for orders with amount greater than or equal to \$100.

All the views/queries are specified using a syntax similar to ESQL. Update views are used to map objects to relations in

$$\begin{array}{llllll}
A_1 & \forall x,y & \mathit{SmallOrders}(x,y) & \Leftrightarrow & (\mathit{Orders}(x,y) \land (y < 100)) \\
A_2 & \forall x,y & \mathit{BigOrders}(x,y) & \Leftrightarrow & (\mathit{Orders}(x,y) \land (y \geq 100)) \\
A_3 & \forall x,y & \mathit{Orders}_{new}(x,y) & \Leftrightarrow & (\mathit{SmallOrders}(x,y) \lor \mathit{BigOrders}(x,y)) \\
A'_2 & \forall x,y & \mathit{BigOrders}(x,y) & \Leftrightarrow & (\mathit{Orders}(x,y) \land (y > 100)) \\
C & \forall x,y & \mathit{Orders}_{new}(x,y) & \Leftrightarrow & \mathit{Orders}(x,y)
\end{array}$$

Figure 1. Axioms and Conjectures for Example 1

$$\begin{array}{lllll}
A_1 & \forall\ V,x,y,z & CCustomerType(V,x,y,z) & \Rightarrow & CPersonType(V,x,y) \\
A_2 & \forall\ x,y,z,w & CPersonType(CPersons,x,y) \land & & \\
 & & ((\exists\ a\ CCustomerType(CPersons,x,y,a) \land & & \\
 & & \quad AddressType(a,z,w)) \lor & & \\
 & & (\neg\exists\ a\ CCustomerType(Cpersons,x,y,a) \land & & \\
 & & \quad z = null \land w = null)) & \Leftrightarrow & SpersonType(SPersons,x,y,z,w) \\
A_3 & \forall\ x,y,a & \exists z,w(SPersonType(SPersons,x,y,z,w) \land & & \\
 & & \qquad w \neq null \land AddressType(a,z,w)) & \Leftrightarrow & CCustomerType(CPersons_{new},x,y,a) \\
A_4 & \forall x,y & \exists z,w\ \ SPersonType(SPersons,x,y,z,w) & \Leftrightarrow & CPersonType(CPersons_{new},x,y) \\
C_1 & \forall\ x,y & CPersonType(CPersons,x,y) & \Leftrightarrow & CPersonType(CPersons_{new},x,y) \quad \land \\
 & \forall\ x,y,a & CCustomerType(CPersons,x,y,a) & \Leftrightarrow & CCustomerType(CPersons_{new},x,y,a)
\end{array}$$

Figure 2. Axioms and Conjectures for Example 2

the database, while query views are used to map database relations to objects.
Suppose we are given the following update view for $\mathit{SmallOrders}$:

```
SELECT (id,amount) FROM Orders
       WHERE amount < 100
```

Further, suppose that we are given the following update view for $\mathit{BigOrders}$:

```
SELECT (id,amount) FROM Orders
       WHERE amount >= 100
```

This can be translated to the FOL formulas given in A_1 and A_2 in figure 1. [1]
Suppose we are given the query view for Orders:

```
SELECT (id,amount) FROM SmallOrders
  UNION ALL
SELECT (id,amount) FROM BigOrders
```

This can translate to the FOL formula given in A_3 (figure 1). Note Orders_{new} is a new relation. We can now formulate the roundtrip verification condition as proving the conjecture C using the axioms A_1, A_2 and A_3, which can be readily proved by using an off-the-shelf FOL theorem prover.

Now, suppose we made a mistake and wrote the update view for `BigOrders` as:

```
SELECT (id,amount) FROM Orders
       WHERE amount > 100
```

[1]Please note that our current implementation uses first-order logic with equality. We have included this example which uses relational operators like less-than for ease of explanation.

This translates to the FOL formula in A'_2.

Now, if we try to prove conjecture C using the axioms A_1, A'_2, and A_3, we find that we are unable to do so. By using a model generator, we can generate a counter model $\mathit{Orders} = \{(1,100)\}$, $\mathit{SmallOrders} = \{\}$, $\mathit{BigOrders} = \{\}$, and $\mathit{Orders}_{new} = \{\}$.

While the above examples illustrate both correct, and incorrect OR maps, they are too simplistic. Realistic OR maps include object oriented features which complicate translation to first order logic. To illustrate some of these complications, consider the following entity schema types: (1) $CPersonType(id, name)$, and (2) $CCustomerType(addr : AddressType(state, zip)) : CPersonType$. Here $CCustomerType$ inherits from $CPersonType$ and adds an extra property $addr$, which is of a complex type $AddressType$ with two fields $state$ and zip. Suppose the database schema has only one type $SPersonType(id, name, state, zip)$. Further, suppose we have one entity set $CPersons$ in the object model of type $CPersonType$, and one entity set $SPersons$ in the database of type $SPersonType$.

How do we represent the entity set $CPersons$ in FOL? One complication is that $CPersons$ can have objects of type both $CPersonType$ and $CCustomerType$. In certain queries, an object of type $CCustomerType$ might be cast into its super-type $CPersonType$, and later down-cast into an object of type $CCustomerType$. To handle such cases uniformly, we create one relation in the FOL for each type in the entity schema. The actual entity set is present as a name in the first attribute of the relation. In our current example, since we have two types $CPersonType$ and $CCustomerType$, we have two FOL relations $CPersonType(r, id, name)$ and $CCustomerType(r, id, name, a)$, where a is of type $AddressType$. Conceptually we can think about a as being a foreign key that indexes in $AddrType$. Since $AddrType$ is a complex type, we have a FOL relation $AddressType(a, state, zip)$, where a is the key of the relation. Inheritance is modeled using the axiom A_1 (figure 2), states that every $CCustomerType$ object is also a $CPersonType$ object.

Suppose update view U for $SPerons$ is given as:

```
SELECT  id, name,
 TREAT(CPersons as CCustomerType).addr.state,
 TREAT(CPersons as CCustomerType).addr.zip
FROM CPersons
```

Translating this to FOL is far more complicated than in the pure relational case (see figure 2). Axiom A_2 relates $SPersons$ to $CPersons$, taking into account that objects in $CPersons$ could be either of type $CPersonType$, or of type $CCustomerType$. In the latter case, the last attribute of $CCustomerType$ is a foreign key a for the relation $AddressType$.

Let the query view Q for $CPersons$ be given by the following ESQL statement:

```
SELECT VALUE CCustomerType(id, name, addr)
FROM
  SELECT  id, name,
       AddressType(state, zip) AS addr
    FROM SPersons
    WHERE !IsNull(Spersons.zip)
UNION ALL
SELECT VALUE CPersonType(id, name) FROM
  SELECT   id, name FROM SPersons
    WHERE IsNull(Spersons.zip)
```

Then, we can generate the FOL formula for Q as in A_3 and A_4 (figure 2). Now, to prove the round trip condition, we need to prove the conjecture C_1 which states that $CPersons$ and $CPersons_{new}$ are equivalent using axioms A_1, A_2, A_3, and A_4.

In general, a view is defined by an ESQL command e of the form `SELECT p FROM r WHERE c`, in which p, r and c may all contain object oriented constructs. The translation of such expressions is quite non-trivial and is done inductively using a function $\mathcal{F}$ from query expressions to FOL formulas. Formula $\mathcal{F}(e)$ is defined in terms of $\mathcal{F}(p)$, $\mathcal{F}(r)$ and $\mathcal{F}(c)$. It turns out, that in order to do the translation inductively, at each expression node e, we need to maintain a list of variables $\mathcal{E}(e)$ that need to be existentially quantified and a formula $\mathcal{B}(e)$ that defines bindings to these variables. Section 5 gives the inductive translation formally.

3 Extended Relational Algebra

In this section, we describe the syntax and some aspects of the semantics of Extended Relational Algebra (ERA). The full formal semantics can be obtained from [18]. We use ERA to study object oriented extensions to traditional relational algebra.

We give the syntax of the ERA using the BNF notation given in figure 3. The basic operators in this algebra are σ (*Selection*), ρ (*Renaming*), Π(*Projection*), $\times$ (*cross product*) and the set union and difference. It also has type casting, type checking, pointer referencing and dereferencing, etc.

The grammar has six non-terminal symbols. Among these τ defines types. We assume that in the definition of a type the name id uniquely identifies the type. Each expression generated from r is an ERA expression. Semantically it denotes a set of entities or simply a relation. The non-terminals e, p, f define entity, property and field expressions, respectively. Finally c defines conditions. Each expression generated from the non-terminal e denotes a single entity or tuple. In the first rule for e, s is an entity set or an identifier/variable that renames an entity set (using the rename operation). In the entity expressions the expressions *treat* e *as* τ cast an entity of some base type to one of it's subtypes or vice versa. Also, $\&$ generates the address and $*$ dereferences an address. Note that . is used to access attributes of an entity, while the $\cdot$ notation is used to access sub-fields of a complex type.

3.1 ERA semantics

An EDM database D is a triple $(T, ESN, type)$ where T is a set of types and ESN is a set of entity sets, and $type$ is a function that associates a type with each entity set. We assume that there is a function key that given an entity type τ gives the fields in the entity that define the key for the entity sets of that type; We assume that all key fields are of base type.

An ERA expression is a string generated from the non-terminal r. We say that an ERA expression is a rename operation if it is of the form $\rho_s(r')$. We say that it *renames* an entity set eid with the variable s if either $r' = eid$ or r' itself renames eid with some variable s'. We say that a variable s refers to the entity set eid in ERA expression r' if there is a sub-expression r of r' that renames eid with s and r does not appear in the scope of a projection or another rename operation and does not appear in the scope of $\cup$ or $\setminus$. For an entity set eid, we can view it as a variable when used in an entity expression. In this case, we say that eid references eid in r, if eid appears as a sub-expression in r and does not appear in the scope of a projection, rename, $\cup$ or $\setminus$.

Example 1: Consider the entity type $CPersonType(id, name)$ and it's sub-type $CCustomerType(addr : AddressType(state, zip)) : CPersonType$ as given in section 2. Let

Types	τ	::=	int | bool | string	(base types)
			| id$(\tau_1, \tau_2, \ldots, \tau_n)$	(complex type)
			| id$(\tau_1, \tau_2, \ldots, \tau_n) :: \tau$	(sub type)
			| $ref_\tau(\tau_1, \tau_2, \ldots, \tau_r)$	(ref type)
Relations	r	::=	eid	(table name)
			| $\rho_s(r)$	(rename r to s)
			| $\sigma_c(r)$	(select)
			| $\Pi_{p_1,p_2,\ldots,p_k}(r)$	(project)
			| $r_1 \times r_2$	(cross product)
			| $r_1 \cup r_2$ | $r_1 \setminus r_2$	(set operations)
Entity expression	e	::=	s	(identifier, table name)
			| treat e as τ	(type cast)
			| $*p$	(dereference)
PropertyExpressions	p	::=	f	(field)
			| & e	(address)
Field	f	::=	$e.i$	(field access)
			| $f \cdot i$	(sub-field access)
Conditions	c	::=	isof(e, τ)	(type check)
			| isnull(p)	(null check)
			| $p_1 = p_2$	(value equality check)
			| $c_1 \wedge c_2$ | $c_1 \vee c_2$ | $\neg c$	(boolean combinations)

Figure 3. BNF for Extended Relational Algebra

$COrderType(Oid, Odesc)$ be another entity type. Let $CPersons$ and $COrders$ be entity sets of types $CPersonType$ and $COrderType$, respectively. In the ERA expression $\rho_s(CPersons) \times COrders$, s refers to the entity set $CPersons$. On the other hand in the ERA expression $\rho_t(\rho_s(CPersons)) \times COrders$ s does not refer to the entity set $CPersons$.

Types of entity and property expressions.

Now, for every w which is either an entity, or a property or a field expression, and for every ERA expression r, we define $expr\_type(w, r)$ which is the type of w in r and we also define $Var(w, r)$ which is the variable that w denotes in r. For an entity expression w that does not contain *, $Var(w)$ is the entity set variable specified in w; further, if w does not contain $treat$ then $expr\_type(w, r)$ is the entity type referenced by $Var(w)$; if w contains $treat$ then $expr\_type(w, r)$ may be the type mentioned in the $treat$ clause. If w contains * then $Var(w) = Null$ and $expr\_type(w, r)$ is the type of entity the pointer points to. For an entity expression $w = e$, $Var(e, r)$ and $expr\_type(e, r)$ are formally defined as follows.

- If $e = s$ then $Var(e, r) = s$. Further more, if s references entity set eid in r then $expr\_type(e, r) = type(eid)$.

- If $e = treat\ e'\ as\ \tau$ then $Var(e, r) = Var(e', r)$. In this case, if τ is a sub-type of $expr\_type(e', r)$ then $expr\_type(e, r) = \tau$; otherwise, $expr\_type(e, r) = expr\_type(e', r)$.

- If $e = *p$ and $expr\_type(p, r) = ref_\tau$ for some entity type τ, then $Var(e, r) = Null$ and $expr\_type(e, r) = \tau$.

For a property or a field expression w, $Var(w, r)$ and $expr\_type(w, r)$ are defined as follows.

- If w is the property expression $\&e$ then $Var(w, r) = Null$. In this case, if $expr\_type(e, r) = \tau$ then $expr\_type(w, r) = ref_\tau$.

- For a property expression w, if $w = f$ where f is a field expression, $expr\_type(w, r) = expr\_type(f, r)$ and $Var(w, r) = Var(f, r)$.

- If w is a field expression then we do as follows. It should be easy to see that $w = e.i_1 \cdot i_2 \ldots \cdot i_k$ for some entity expression e. If $expr\_type(e, r) = id(\tau_1, \ldots, \tau_n)$ then we define $expr\_type(w, r)$ to be the type of the sub-attribute $(id.i_1 \cdot \ldots \cdot i_k)$. If e does not contain $*$ then $Var(w, r) = s.i_1 \cdot \ldots \cdot i_k$ where $s = Var(e, r)$. If e contains $*$ then $Var(e, r) = Null$.

Example 2: Let r be the ERA expression $\rho_s(CPersons) \times COrders$. It should be easy to see that $expr\_type(s, r)$ is $CPersonType$ and $expr\_type(treat \ s \ as \ CCustomerType, \ r)$ is $CCustomerType$.

It should be easy to see that for any entity or property expression w, $expr\_type(w, r)$ and $Var(w, r)$ are easily computed using the above definitions.

4 Formulation

Let $\mathcal{P}$ be the set of all physical database states, and $\mathcal{E}$ be a set of all entity database states. We consider two ERA relational expressions: (1) an update view U, and (2) a query view Q. Semantically, $U : \mathcal{E} \rightarrow \mathcal{P}$ and $Q : \mathcal{P} \rightarrow \mathcal{E}$.

The round-trip condition from the entity side requires that $Q \cdot U$ be the identity map. However this may not hold as it does not take into consideration the integrity constraints the entities and databases are supposed to satisfy. Let $\mathcal{E}'$ be the set of all elements of $\mathcal{E}$ that satisfy the integrity constraints of the EDM data declaration. Let $\mathcal{P}'$ be the image of $\mathcal{E}'$ under the mapping U, i.e., $\mathcal{P}' = \{U(s) : s \in E'\}$. We say that the pair of maps (U, Q) satisfy the round-trip condition from the entity side, if $\forall s \in \mathcal{E}' \, Q(U(s)) = s$. Let U' be the mapping which is a restriction of U to $\mathcal{E}'$. Then the above condition is equivalent to requiring that $Q \cdot U'$ be the identity function on domain $\mathcal{E}'$. Similarly, we say that the pair (U, Q) satisfies the round-trip condition from the database side if $\forall t \in \mathcal{P}' \, U(Q(t)) = t$. Now we have the following lemma which is easily proved.

Lemma 1: $\forall t \in \mathcal{P}' \, U(Q(t)) = t$ iff $\forall t \in \mathcal{E}' \, Q(U(t)) = t$

The central question in the paper is to check if a pair (U, Q) satisfies the round-trip condition:$\forall s \in \mathcal{E}' \;\; Q(U(s)) = s$. Our approach is to translate the ERA relational expressions U and Q into first order logic, and use first order logic theorem proving to check the round-trip condition.

5 Translation of ERA to FOL

Corresponding to the EDM database $D = (T, ESN, type)$, we define a canonical relational database E. As indicated earlier, for each type $\tau = id(\tau_1, \ldots, \tau_n)$, we have a relation $id(name, a_1, \ldots, a_m)$ where $name$ is an entity set (i.e., its name) if τ is an entity type, otherwise $name$ is of type integer that is a key for the relation. If τ_i is of complex type $id'(\tau'_1, \ldots, \tau'_k)$ then a_i is of type integer and is a foreign key referencing the table $id'(\tau'_1, \ldots, \tau'_k)$.

Note that some of the complex types and subtypes are entity types. In our translation, for each such type $\tau = id(\tau_1, \ldots, \tau_n)$, we use a relation $pkey_\tau$ that captures the one-one relationship between pointers to entities in entity sets and keys of such entities. Let $key(\tau) = (\tau_{i_1}, \ldots, \tau_{i_k})$. The arity of $pkey_\tau$ is $k + 2$. Formally, if $pkey_\tau(u, p, d_1, \ldots, d_k)$ is true then it indicates that p is the pointer to the entity with key values $d_1, ..., d_k$ in the entity set u. As part of the axioms, we assert the following: (a) p is the key of this relation, i.e., no two tuples have the same p values; this asserts that pointers are unique. (b) the attributes $u, d_1 \ldots, d_k$ also form a key indicating that this is a one-one relationship; (c) for each entity set u of type τ and for each key value present in the relation u, there exists a tuple in $pkey_\tau$ with the same values for $d_1, \ldots, d_k$; (d) for each entity set eid and for every tuple t in $pkey_\tau$ such that $t.u = eid$ there exists a tuple in eid with the same key values as $t.d_1, t.d_2, \ldots, t.d_k$.

For each relational expression r' generated by r, we define a FOL formula $\mathcal{F}(r')$. Intuitively, this formula has the property that, in any database state s, the result of evaluating r' on s outputs a relation which is exactly the set of tuples that satisfy $\mathcal{F}(r')$ in the interpretation s. Similarly, for each expression f which is an entity expression or property or field or condition, we generate a formula $\mathcal{F}(f)$. For an expression f, which is an entity expression or property or field, we generate a set $\mathcal{E}(f)$ of variables that need to be existentially quantified and a formula $\mathcal{B}(f)$ that defines a binding on these variables. Actually, $\mathcal{F}(f), \mathcal{E}(f)$ and $\mathcal{B}(f)$ not only depend on f but they also depend on the ERA expression r in whose context they are being defined. In these definition r is clear from the context, otherwise we will explicitly mention r.

For any FOL formula F, let $free\_var(F)$ denote the sequence of free variables that appear in F; the ordering of these variables is appropriately defined. For a sequence of variables $\vec{X}$, of the same length as $free\_var(F)$, we let $F(\vec{X})$ denote the formula obtained by substituting the variables in $\vec{X}$ for the free variables in F in the given order. For each identifier g, we have a set of first order variables $\{g \cdot i : i \geq 1\}$. For each relational expression r', $\mathcal{F}(r')$ is defined inductively based on its outer most connective, as follows. We also define a vector $free\_var(\mathcal{F}(r'))$ of variables that appear free in $\mathcal{F}(r')$.

- $r' = eid$: Let $type(eid) = id(\tau_1, \ldots, \tau_n)$. Then, $\mathcal{F}(eid) = id(eid, eid \cdot 1, \ldots, eid \cdot n)$ and $free\_var(\mathcal{F}(r')) = (eid \cdot 1, \ldots, eid \cdot n)$.

- $r' = \rho_s(r)$: Let n be the number of free variables in

$\mathcal{F}(r)$ and $\vec{Y} = (s\cdot 1, \ldots, s\cdot n)$. This translation simply renames the free variables in $\mathcal{F}(r)$ to be those in $\vec{Y}$. Formally, $\mathcal{F}(r') = \mathcal{F}(r)(\vec{Y})$ and $free\_var(\mathcal{F}(r')) = (\vec{Y})$.

- $r' = \sigma_c(r)$: In this case, $\mathcal{F}(r') = \mathcal{F}(r) \wedge \mathcal{F}(c)$ and $free\_var(\mathcal{F}(r')) = free\_var(\mathcal{F}(r))$.

- $r' = \Pi_{p_1,\ldots,p_n}(r)$: In this case, for each $i = 1, ..., n$, $\mathcal{F}(p_i)$ is a single variable. Let C be the formula $\bigwedge_{i=1}^{n} \mathcal{B}(p_i)$ and $\vec{Y} = \bigcup_{i=1}^{n} \mathcal{E}(p_i)$. Let $\vec{X}$ be the set of variables y such that y is not in $\{\mathcal{F}(p_i) \ : 1 \leq i \leq n\}$, and such that y appears either in $\vec{Y}$ or in $\mathcal{F}(r)$ as a free variable. Then, $\mathcal{F}(r') = \exists \vec{X}(\mathcal{F}(r) \wedge C)$. We define $free\_var(\mathcal{F}(r')) = (\mathcal{F}(p_1), \ldots, \mathcal{F}(p_n))$.

- $r' = r_1 \times r_2$: In this case, we assume that the formulas $\mathcal{F}(r_1)$ and $\mathcal{F}(r_2)$ do not have any common free variables. If this is not satisfied, the variables are renamed to satisfy this property. We define $\mathcal{F}(r') = \mathcal{F}(r_1) \wedge \mathcal{F}(r_2)$ and $free\_var(\mathcal{F}(r'))$ is obtained by concatenating the vectors $free\_var(\mathcal{F}(r_1))$ and $free\_var(\mathcal{F}(r_2))$ in that order.

- $r' = r_1 \cup r_2$ or $r' = r_1 \setminus r_2$: If $r' = r_1 \cup r_2$ then $\mathcal{F}(r') = \mathcal{F}(r_1) \vee \mathcal{F}(r_2)$. If $r' = r_1 \setminus r_2$ then $\mathcal{F}(r') = \mathcal{F}(r_1) \wedge \neg\mathcal{F}(r_2)$. Here we assume that $free\_var(\mathcal{F}(r_1)) = free\_var(\mathcal{F}(r_2))$. We define $free\_var(\mathcal{F}(r')) = free\_var(\mathcal{F}(r_1))$.

Now we define the translation for entity expressions. For an entity expression e', $\mathcal{F}(e')$, $\mathcal{E}(e')$ and $\mathcal{B}(e')$ are defined inductively as follows. We assume that these definitions are given in the context of the ERA expression r. If e' is not a legal entity expression in r then $\mathcal{F}(e') = Null$, $\mathcal{B}(e') = true$ and $\mathcal{E}(e') = \emptyset$. If e' is a legal entity expression in r then the above value are defined as follows.

- $e' = s$: Let $expr\_type(s, r) = id(\tau_1, \ldots, \tau_n)$. (Recall that $expr\_type(s, r)$ is defined in section 3). Let eid be the entity set referenced by s in r. Then, $\mathcal{F}(e') = id(eid, s\cdot 1, \ldots, s\cdot n)$, $\mathcal{B}(e') = true$ and $\mathcal{E}(e') = \emptyset$.

- e' = treat e as τ: Let $\tau = id'(\tau_1', \ldots, \tau_n')$ and let $\mathcal{F}(e) = id(u, v\cdot 1, \ldots, v\cdot m)$ for some u, v, m. Here u can be a constant or a variable. If τ is a super type of $expr\_type(e, r)$ then $\mathcal{F}(e') = \mathcal{F}(e)$, $\mathcal{B}(e') = \mathcal{B}(e)$ and $\mathcal{E}(e') = \mathcal{E}(e)$. Now, assume that τ is subtype of $type(u)$ and hence $n > m$. In this case, if the entity belongs to the subtype τ then the additional attributes of the entity are retrieved, otherwise null values are retrieved. This is done by defining $\mathcal{F}(e')$, $\mathcal{B}(e')$ as follows.
  - $\mathcal{F}(e') = id'(u, v\cdot 1, \ldots, v\cdot n)$;
  - $\mathcal{B}(e') = \mathcal{B}(e) \wedge (id'(u, v\cdot 1, \cdot, v\cdot n) \vee g)$ where $g = \neg\exists y_{m+1}, ..., y_n \, id'(u, v\cdot 1, \ldots, v\cdot m, y_{m+1}, \ldots, y_n) \bigwedge_{m<i\leq n}(v\cdot i = cNull)$;
  - $\mathcal{E}(e') = \mathcal{E}(e) \cup \{v\cdot j \ : m < j \leq n\}$.

- $e' = *p$: Let $expr\_type(p, r) = ref_\tau$ where $\tau = id(\tau_1, \ldots \tau_n)$, $key(\tau) = (i_1, \ldots, i_k)$ and v and j be new names. Recall that $key(\tau)$ gives the attributes of the key. In this case $\mathcal{F}(p)$ is a variable. Using the relation $pkey_\tau$ that relates pointers to key values of entities, we retrieve the key fields, the entity name and the entity. If this is a dangling pointer then we retrieve $Null$ values. This is done by appropriately defining $\mathcal{B}(e')$.
  - $\mathcal{F}(e') = id(j, v\cdot 1, \ldots, v\cdot n)$;
  - $\mathcal{B}(e') ::= \mathcal{B}(p) \wedge (g \vee h)$ where $g = pkey_\tau(j, \mathcal{F}(p), v\cdot i_1, \ldots, v\cdot i_k) \wedge id(j, v\cdot 1, \ldots, v\cdot n)$ and $h = (\neg\exists(j', y_1, \ldots, y_k) pkey_\tau(j', \mathcal{F}(p), y_1, \ldots, y_k) \bigwedge_{1\leq i\leq n}(v\cdot i = cNull))$;
  - $\mathcal{E}(e') = \mathcal{E}(p) \cup \{j, v\cdot 1, \ldots, v\cdot n\}$.

Now we give the translation of property expressions. For a property expression p, $\mathcal{F}(p)$, $\mathcal{B}(p)$ and $\mathcal{E}(p)$ are defined as follows.

- $p = f$: In this case, $\mathcal{F}(p) = \mathcal{F}(f)$, $\mathcal{B}(p) = \mathcal{B}(f)$ and $\mathcal{E}(p) = \mathcal{E}(f)$.

- $p = \&e$: let $\mathcal{F}(e) = id(u, v\cdot 1, \ldots, v\cdot n)$ for some id, u, v and n. Here u can be a variable or an entity name. Let $expr\_type(e, r) = \tau = id(\tau_1, \ldots \tau_n)$, $key(\tau) = (i_1, \ldots, i_k)$ and x be a new first order variable. Using the key values of u and the relation $pkey_\tau$, we retrieve the pointer to e by appropriately defining $\mathcal{B}(e)$.
  - $\mathcal{F}(p) = x$;
  - $\mathcal{B}(p) = \mathcal{B}(e) \wedge pkey_\tau(u, x, v\cdot i_1, \ldots, v\cdot i_k)$;
  - $\mathcal{E}(p) = \mathcal{E}(e) \cup \{x\}$.

For any field expression f', $\mathcal{F}(f')$, $\mathcal{B}(f')$ and $\mathcal{E}(f')$ are defined as follows.

- $f' = e.k$: Here we have two cases. In the first case, $\mathcal{F}(e) = Null$; in this case e is not legal but $e.k$ or a sub-attribute of it is a variable in $Var(r)$; we define $\mathcal{F}(f') = e.k$, $\mathcal{B}(f') = true$ and $\mathcal{E}(f') = \emptyset$. In the second case, $\mathcal{F}(e) = id(u, v\cdot 1, \ldots, v\cdot n)$ for some u, v, n. We define $\mathcal{F}(f')$ to be $v\cdot k$, $\mathcal{B}(f') = \mathcal{B}(e)$ and $\mathcal{E}(f') = \mathcal{E}(e)$.

- $f' = f \cdot k$: If f is not legal in r then $\mathcal{F}(f') = f \cdot k$, $\mathcal{B}(f') = true$ and $\mathcal{E}(f') = \emptyset$. Otherwise, let $expr\_type(f, r) = id(\tau_1, \ldots \tau_n)$. In this case, f is an attribute of complex type. It's sub-fields are stored in the relation id. The attribute f is a key of this relation and the sub-field $f \cdot k$ is retrieved from this relation. This is achieved using $\mathcal{B}(f')$.
  - $\mathcal{F}(f') = f \cdot k$;
  - $\mathcal{B}(f') = \mathcal{B}(f) \wedge g$ where $g = id(f, f \cdot 1, \ldots, f \cdot n) \vee (\neg \exists y_1, \ldots, y_n\, id(f, y_1, \ldots, y_n) \wedge \bigwedge_{j=1}^{n} (f \cdot j = cNull))$. Note that the first disjunct in g retrieves the sub-fields from the relation id and the second retrieves null values if no such tuple exists in id;
  - $\mathcal{E}(f') = \mathcal{E}(f) \cup \{f \cdot 1, \ldots f \cdot n\}$.

Now we define $\mathcal{F}(c), \mathcal{B}(c)$ and $\mathcal{E}(c)$ for a condition c. In this case all the bindings of various fields referenced in c are merged with the main formula and all existential variables are existentially quantified. We give only two cases. Other cases are similar and are left out due to space consideration.

- $c = IsNull(p)$: In this case,
  - $\mathcal{F}(c) = \exists \mathcal{E}(p)(\mathcal{B}(p) \wedge (\mathcal{F}(p) = cNull))$.
- $c = IsOf(e, \tau)$: Let $\tau = id'(\tau'_1, \ldots \tau'_n)$. In this case, $\mathcal{F}(e) = id(u, v \cdot 1, \ldots, v \cdot m)$ for some u, v, m.
  - $\mathcal{F}(c) = \exists(\mathcal{E}(e) \cup \{v \cdot j\ : m < j \leq n\})(\mathcal{B}(e) \wedge id'(u, v \cdot 1, \ldots, v \cdot n))$.

6 Implementation

Our implementation ROUNDTRIP takes as input the object and the database schema expressed in the EDM language [2], and the query and update views expressed in the ESQL language [2] and produces the first-order logic formulas. ROUNDTRIP generates the axioms for the primary key constraints and the other domain constraints as described above, and then axioms for the queries. It then runs FOL theorem prover, E-PROVER [21], on the generated formula to verify correctness. Theorem provers such as E-PROVER try to prove correctness of a FOL formula $\mathcal{F}(e)$ by proving unsatisfiability of the formula $\neg\mathcal{F}(e)$. However, in most cases where the round-tripping condition is not satisfied (the FOL formula is satisfiable), they time out. If E-PROVER does not terminate within 120 seconds, then ROUNDTRIP runs our custom model generator on the formula $\neg\mathcal{F}(e)$.

Model Generation. Our custom model generator translates the first-order logic formula into a boolean satisfiability problem instance, and use a off-the-shelf SAT solver for searching for a correct model. The translation from FOL to SAT is parameterized by: (1) a maximum bound on the number of tuples in every relation k, and (2) the number of constants in the domain c.

Existing approaches [14, 8, 17] encode a predicate P of arity a into boolean variables of type $P_{v_1,\ldots,v_a}$, each of which indicate if the instance $P(v_1, \ldots, v_a)$ is true. Since each of the variables v_i can take any of the values from $(1..c)$, there are c^a variables. We have a new translation wherein the predicate is translated into a set of k rows, each of which contains a boolean variable P_i to indicate if that row is valid, and a set of a variables each of size $log_2(c)$. These a variables provide the valuation to the row when it is valid. Thus, the number of variables is $k*(1+a*log_2(c))$.

We first encode the FOL formula into a quantified boolean formula (QBF) using the above scheme. The QBF formula is now converted into a SAT instance. We use Binary Decision Diagrams [5] to perform these quantifications, since the nesting of quantifiers is quite deep, and directly eliminating quantifiers does not scale. We then convert the BDD for the formula after quantifier elimination into a set of equivalent CNF clauses and feed it to the SAT solver. Since we have bounded constants and rows in the translation process, if the SAT solver provides a model, then the model indeed is a model for the FOL formula and we can display it to the user in a suitable format. In case, the SAT solver says that the boolean formula is UNSAT, we need to increase k and c and repeat the process.

Empirical results. We ran experiments on a set of test cases from the ADO.NET Benchmark v3 suite obtained from a product group at Microsoft. We ran these experiments on a Pentium 4, 1800 Mhz processor with 2GB RAM. We ran these experiments with a timeout of 120s for E-PROVER. Our model generator bounded the number of rows as 2, and the number of constants in the domain as 32. We use Minisat [11] for SAT solving.

Figure 4 shows our empirical results. The first 16 examples are correct examples (i.e. the roundtrip condition is satisfied). The last 5 examples are incorrect examples. We manually introduced bugs in these examples to test our model generation.

The first column is the name of the test case. Columns 2 through 5 give some idea of the size of the example, including the number of EDM types, and the average and maximum arities of the relations. "Trans time" is the time taken by the ESQL-FOL translator, "TP time" is the time taken by E-PROVER, "MG Rslt" is the result of our custom model generator and "MG time" is the time taken by it. The model generator is run only if E-PROVER is not able to finish in 120s. "Num clauses" and "Num literals" give some mea-

Problem Name	#-types	Avg Arity	Max Arity	Trans Time	TP Time	Num clauses	Num ltrls	MG Rslt	MG Time
Aruba437719_1	9	3.4	6	2.5	18.8				
Aruba437719_2	9	3.7	6	2.5	1.8				
Aruba437719_3	9	3.4	6	2.5	1.8				
Assn1-1_0	17	5.0	11	19.2	0.3				
Assn1-1_1	23	4.2	11	19.2	0.4				
Assn1-1_2	17	5.1	11	19.2	0.5				
dpmud-smpl_0	16	5.5	11	31.2	0.6				
dpmud-smpl_1	22	4.2	11	31.2	0.4				
FOJ1_0	11	2.9	6	4.3	timeout	682615	171262	UNSAT	26.3
Inheritance_0	6	5.7	7	1.5	48.3				
Inheritance_1	6	5.8	7	1.5	2.4				
InhVrtclPr_1	10	2.5	5	2.1	8.2				
SelfAssn_0	17	2.6	6	5.6	0.2				
SelfAssn_1	11	3.4	6	5.6	0.3				
SmplMapTwo_0	10	4.3	7	1.9	9.1				
Union1_2	11	5.0	8	3.6	timeout	1044966	262020	UNSAT	37.6
InhTPH2_0*	6	5.7	7	1.4	timeout	233485	58875	SAT	9.9
InhTPH2_1*	6	5.8	7	1.4	timeout	252723	63679	UNSAT	10.4
InhVrtclPr_0*	7	2.0	3	2.2	timeout	100922	25605	SAT	5.5
InhVrtclPr_1*	10	2.5	5	2.2	timeout	436059	109450	SAT	17.4
InvSmplMp_0*	5	2.4	3	1.2	timeout	27459	7188	SAT	3.8

Figure 4. Empirical results from the ROUNDTRIP tool

sure of the size of the generated SAT formula.

We find that E-PROVER is able to scale quite well for most correct examples. In most examples E-PROVER takes even less time than the ESQL-FOL translator. In 4 out of 6 case where E-PROVER times out our model generator is able to find a model. In 2 cases, the model generator says correct (UNSAT). We examined these cases, and found that our manually introduced bugs did not affect the correctness of these examples, as specified by the roundtrip condition.

7 Discussion

Formal methods have been successful in validating hardware [6, 15], low-level software [4], and protocol software [13, 12]. To the best of our knowledge, ours is the first such effort to validate data access in databases.

The relationship between Relational Algebra or SQL and first order logic is well known, and can be found in database text books [22, 1]. As indicated in the introduction, there have also been works [1, 24, 19, 23] on translating object oriented models and query languages in to relation models and query languages, and also into relational calculus. However, none of these handled the complexity of a practical query language such as ESQL (with OO extensions) used in a real product supporting many features. The pointer referencing and dereferencing, type casting, dynamic type checking, attribute renaming, complex case splitting, can all be nested within each other in ESQL in complex ways. Queries can be arbitrarily nested and a variety of joins, including outer joins, can be used. Most of the works [1, 24, 19, 23] handle path expressions (i.e., complex types) and some form of sub-typing. None of them handle pointer referencing and dereferencing, dynamic type checking, etc. Also many of these translations are informally given and no correctness proofs are presented. There have also been extensions of traditional Relational Algebra to nested models and object oriented models [1, 24, 19]. These extensions do not allow constructs for pointer referencing, dereferencing and dynamic type checking as ERA does.

Roundtrip verification is related to query equivalence in relational algebra. By restricting relational algebra we can obtain fragments for which query equivalence is decidable. Examples of such fragments are conjunctive queries, and queries where the project operator is not applied to subexpressions with the difference operator (i.e, no negation inside an existential quantification) [20]. Query equivalence has also been studied for such restricted fragments with extensions such as Datalog [16] or aggregate queries [9]. All of the above efforts have been theoretical investigations, and have not resulted in practical tools, since the queries that appear in practice can fall outside these decidable fragments. In contrast, we do not constrain the query language, and we have been able to build a practically useful tool using theorem proving.

We believe that our approach can also help with other verification problems in databases such as (1) verifying correctness of query optimizers, and (2) verifying query equivalence when queries need to be changed for the purpose of SQL migration. We also believe that our approach to first order logic model generation scales better than existing approaches, and plan to investigate it further.

Acknowledgment. We thank Atul Adya and Sergey Melnik for advice, discussions, and help in supplying the examples used for our empirical evaluation. We thank Sumit Gulwani for discussions in formulating the approach. We thank Supratik Chakraborty for comments on a draft of this paper.

References

[1] Abiteboul, R. Hull, and V. Vianu. *Foundations of Database Systems*. Addison-Wesley, 1997.

[2] A. Adya, J. Blakeley, S. Melnik, S. Muralidhar, and the ADO.NET Team. Anatomy of the ADO.NET entity framework. In *SIGMOD*, 2007.

[3] A. Adya, S. Melnik, and P. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD*, 2007.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01*, LNCS 2057. Springer-Verlag, 2001.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[6] J. Burch, D. Dill, E. Clarke, K. McMillan, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, 1990.

[7] M. J. Carey and D. J. DeWitt. Of objects and databases: A decade of turmoil. In *VLDB 96: Very Large Databases*, pages 3–14, 1996.

[8] K. Claessen and N. Sörensson. Paradox model finder – http://www.cs.chalmers.se/ koen/paradox/.

[9] S. Cohen, Y. Sagiv, and W. Nutt. Equivalences among aggregate queries with negation. *ACM Trans. Comput. Log.*, 6(2):328–360, 2005.

[10] W. Cook and A. Ibrahim. Integrating programming languages and databases: What is the problem? In *ODBMS.ORG, Expert Article*, 2006.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 03: Satisfiability Testing*, pages 502–518, 2003.

[12] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97*, pages 174–186, 1997.

[13] G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Tech Journal*, 5(2):72–87, - 2000.

[14] D. Jackson. Alloy: A new technology for software modelling. In *TACAS*, 2002.

[15] M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[16] A. Y. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability, and satisfiability in datalog extensions. In *PODS*, 1993.

[17] W. McCune. Mace4 reference manual and guide. *CoRR*, cs.SC/0310055, 2003.

[18] K. Mehra, S. K. Rajamani, A. P. Sistla, and S. Jha. Verification of Object Relational Maps. Technical Report MSR-TR-2007-71, Microsoft Research, 2007.

[19] J. Paredaens and D. V. Gucht. Converting nested algebra expressions to flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.

[20] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union difference operators. *J. ACM*, 27(4):633–655, 1980.

[21] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[22] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill Higher Education, 2001.

[23] L. Wong. Normal forms and conservative properties for query languages over collection types. In *PODS*, pages 26–36, 1993.

[24] C. T. Yu and N. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Caufman, 1997.