Verification of Object Relational Maps

Krishna K. Mehra*  Sriram K. Rajamani*
A. Prasad Sistla† Sumit K. Jha‡

*Microsoft Research India  †University of Illinois at Chicago
‡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.

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 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 [8, 5]. 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 mappings 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 [18]. 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, they have difficulty in generating counter-models 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 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 sub-typing 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 scale and generate counter models for invalid formulas. Thus, we were forced to write our own model generation tool using boolean SAT solvers. 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. In our model generator, we also bound the maximum number
of rows in our relations \( k \), in addition to bounding the number of constraints by \( c \). This leads to formulas with (approx) \( a \times k \times log(c) \) 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 most examples that we have encountered.

We have implemented a verification tool \textsc{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 [9], and query and update views provided in Microsoft’s Extended SQL (ESQL) [11] language. \textsc{roundtrip} is able to prove the correctness, or find counter-models for most examples we have tried in a few minutes. \textsc{era} is very close to ESQL. Although, we give the translation from \textsc{era} to FOL in this paper, \textsc{roundtrip} does a direct translation from ESQL to FOL.

The paper is organized as follows. Section 2 motivates the problem with examples. Section 3 defines \textsc{era} and presents its semantics. Section 5 states the round trip problem formally. Section 6 gives the translation from \textsc{era} to FOL. Section 7 presents experimental results.

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 \( \text{Orders}(id, \text{amount}) \), but two relations (1) \( \text{SmallOrders}(id, \text{amount}) \) for orders with amount less than \$100, and (2) \( \text{BigOrders}(id, \text{amount}) \) for orders with amount greater than or equal to \$100. All the views/queries are specified using a syntax similar to ESQL.

Suppose we are given the following update view for \( \text{SmallOrders} \):
\[
\text{SELECT \( (id, \text{amount}) \) FROM \text{Orders} WHERE amount < 100}
\]
Further, suppose that we are given the following update view for \( \text{BigOrders} \):
\[
\text{SELECT \( (id, \text{amount}) \) FROM \text{Orders} WHERE amount \geq 100}
\]

This can translate to the FOL formulas:
\[
A_1. \forall x, y. \text{SmallOrders}(x, y) \iff (\text{Orders}(x, y) \land (y < 100))
\]
\[
A_2. \forall x, y. \text{BigOrders}(x, y) \iff (\text{Orders}(x, y) \land (y \geq 100))
\]

Finally, suppose we are given the query view for \( \text{Orders} \):
\[
\text{SELECT \( (id, \text{amount}) \) FROM \text{SmallOrders} \land \text{BigOrders}}
\]
\[
\text{UNION ALL}
\]
\[
\text{SELECT \( (id, \text{amount}) \) FROM \text{SmallOrders}}
\]
\[
\text{UNION ALL}
\]
\[
\text{SELECT \( (id, \text{amount}) \) FROM \text{BigOrders}}
\]
This can translate to the FOL formula:
\[
A_3. \forall x, y. \text{Orders}_{\text{new}}(x, y) \iff (\text{SmallOrders}(x, y) \lor \text{BigOrders}(x, y))
\]

We can now formulate the roundtrip verification condition as proving the conjecture:
\[
C. \forall x, y. \text{Orders}_{\text{new}}(x, y) \iff \text{Orders}(x, y)
\]
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 \( \text{BigOrders} \) as:
SELECT (id, amount) FROM Orders WHERE amount > 100

This translates to the FOL formula:

\[ A'_2. \forall x, y. \text{BigOrders}(x, y) \Leftrightarrow (\text{Orders}(x, y) \land (y > 100)) \]

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 prove the conjecture. By using a model generator, we can generate a counter model \( \text{Orders} = \{(1, 100)\}, \text{SmallOrders} = \{\}, \text{BigOrders} = \{\}, \text{Orders}_{\text{new}} = \{\} \).

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) \( \text{CPerson}(id, \text{name}) \), and (2) \( \text{CCustomer}(\text{addr} : \text{AddressType}(\text{state}, \text{zip})) : \text{CPerson} \). Here \( \text{CCustomer} \) inherits from \( \text{CPerson} \) and adds an extra property \( \text{addr} \), which is of a complex type \( \text{AddressType} \) with two fields \( \text{state} \) and \( \text{zip} \). Suppose the database schema has only one type \( \text{SPerson}(id, \text{name}, \text{state}, \text{zip}) \). Further, suppose we have one entity set \( \text{CPersons} \) in the object model of type \( \text{CPerson} \), and one entity set \( \text{SPersons} \) in the database of type \( \text{SPerson} \).

How do we represent the entity set \( \text{CPersons} \) in FOL? One complication is that \( \text{CPersons} \) can have objects of type both \( \text{CPerson} \) and \( \text{CCustomer} \). In certain queries, an object of type \( \text{CCustomer} \) might be cast into its supertype \( \text{CPerson} \), and later down-cast into an object of type \( \text{CCustomer} \). 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 \( \text{CPerson} \) and \( \text{CCustomer} \), we have two FOL relations \( \text{CPerson}(r, id, \text{name}) \) and \( \text{CCustomer}(r, id, \text{name}, a) \), where \( a \) is of type \( \text{AddressType} \). Conceptually we can think about \( a \) as being a foreign key that indexes in \( \text{AddrType} \). Since \( \text{AddrType} \) is a complex type, we have a FOL relation \( \text{AddressType}(a, \text{state}, \text{zip}) \), where \( a \) is the key of the relation.

Suppose update view \( \text{U} \) for \( \text{SPersons} \) is given as:

\[
\begin{align*}
\text{SELECT} & \quad \text{id, name, TREAT(\text{CPersons as CCustomer}).addr.state,} \\
& \quad \text{TREAT(\text{CPersons as CCustomer}).addr.zip FROM CPersons}
\end{align*}
\]

Translating this to FOL is far more complicated than in the pure relational case:

\[
\begin{align*}
A_1. & \quad \forall V, x, y, z. \text{CCustomer}(V, x, y, z) \Rightarrow \text{CPerson}(V, x, y) \\
A_2. & \quad \forall x, y, z, w. (\text{CPerson}(\text{CPersons}, x, y) \land \\
& \quad \exists a. (\text{CCustomer}(\text{CPersons}, x, y, a) \lor \\
& \quad \exists b \text{CCustomer}(\text{CPersons}, x, y, b) \land a = \text{null}) \land \\
& \quad (\text{AddressType}(a, z, w) \lor \\
& \quad (\neg \text{AddressType}(a, z, w) \land (z = \text{null}) \land (w = \text{null})))) \\
& \quad \Leftrightarrow \text{SPerson}(\text{SPersons}, x, y, z, w)
\end{align*}
\]

Axiom \( A_1 \) states that every \( \text{CCustomer} \) object is also a \( \text{CPerson} \) object. Axiom \( A_2 \) relates \( \text{SPersons} \) to \( \text{CPersons} \), taking into account that objects in \( \text{CPersons} \)
could be either of type \textit{CPerson}, or of type \textit{CCustomer}. In the latter case, the last attribute of \textit{CCustomer} is a foreign key \(a\) for the relation \textit{AddressType}.

Let the query view \(Q\) for \textit{CPersons} be given by the following ESQL statement:

\[
\text{SELECT VALUE CCustomer(id, name, addr) FROM}
\]
\[
\begin{array}{l}
\text{SELECT id, name,AddressType(state, zip) AS addr}
\text{FROM SPersons}
\text{WHERE !IsNull(Spersons.zip)}
\end{array}
\]
\[
\text{UNION ALL}
\]
\[
\begin{array}{l}
\text{SELECT VALUE CPerson(id, name) FROM}
\text{SELECT id, name FROM SPersons}
\text{WHERE IsNull(Spersons.zip)}
\end{array}
\]

Then, we can generate the FOL formula for \(Q\) as:

\[A_3. \forall(x,y,z,w) \ (SPerson(SPersons,x,y,z,w) \land \neg(w = null) \land \exists a \text{ AddressType}(a,z,w)) \iff CCustomers(CPersons\_new,x,y,a)\]

\[A_4. \forall(x,y,z,w) \ SPerson(SPersons,x,y,z,w) \land (w = null) \iff CPerson(CPersons\_new,x,y)\]

Now, to prove the round trip condition, we need to prove the following conjecture which states that \textit{CPersons} and \textit{CPersons\_new} are equivalent using axioms \(A_1, A_2, A_3, \text{ and } A_4\):

\[C_1. \forall(x,y) \ CPerson(CPersons,x,y) \iff CPerson(CPersons\_new,x,y) \land \forall(x,y,a) \ CCustomer(CPersons,x,y,a) \iff CCustomer(CPersons\_new,x,y,a)\]

In general, a view is defined by an ESQL command \(e\) of the form \(\text{SELECT p FROM r WHERE c}\), 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 semantics of Extended Relational Algebra (ERA). We use ERA to study object oriented extensions to traditional relational algebra.

4 Syntax

We give the syntax of the ERA using the BNF notation given in figure 1. The basic operators in this algebra are \(\sigma\) (Selection), \(\rho\) (Renaming), \(\Pi\) (Projection), \(\times\) (cross product) and the set union and difference.
The grammar has six non-terminal symbols. Among these $\tau$ 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 $\textit{treat } e \textit{ as } \tau$ cast an entity of some base type to one of its 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.

Types

$\tau ::= \text{int} \mid \text{bool} \mid \text{string} \quad \text{(base types)}$
$\mid \text{id}(\tau_1,\ldots,\tau_n) \quad \text{(complex type)}$
$\mid \text{id}(\tau_1,\ldots,\tau_n) :: \tau \quad \text{(sub type)}$
$\mid \text{ref}(\tau_1,\ldots,\tau_r) \quad \text{(ref type)}$

Relations

$r ::= \text{eid}(\text{table name}) \quad \text{(table name)}$
$\mid \rho_s(r) \quad \text{(rename r to s)}$
$\mid \sigma_c(r) \quad \text{(select)}$
$\mid \Pi_{p_1,p_2,\ldots,p_k}(r) \quad \text{(project)}$
$\mid r_1 \times r_2 \quad \text{(cross product)}$
$\mid r_1 \cup r_2 \mid r_1 \setminus r_2 \quad \text{(set operations)}$

Entity expression

$e ::= s \quad \text{(identifier, table name)}$
$\mid \text{treat } e \textit{ as } \tau \quad \text{(type cast)}$
$\mid \star p \quad \text{(dereference)}$

PropertyExpressions

$p ::= f \quad \text{(field)}$
$\mid \& e \quad \text{(address)}$

Field

$f ::= e.i \quad \text{(field access)}$
$\mid f.i \quad \text{(sub-field access)}$

Conditions

$c ::= \text{isof}(e, \tau) \quad \text{(type check)}$
$\mid \text{isnull}(p) \quad \text{(null check)}$
$\mid p_1 = p_2 \quad \text{(value equality check)}$
$\mid c_1 \land c_2 \mid c_1 \lor c_2 \mid \neg c \quad \text{(boolean combinations)}$

\textbf{Fig. 1.} BNF for Extended Relational Algebra
4.1 ERA semantics

An EDM database $D$ is a triple $(T, ES, type)$ 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 $\tau$ 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 $CPerson(id, name)$ and its sub-type $CCustomer(addr : AddressType(state, zip)) : CPerson$ as given in section 2. Let $COrder(OId, Odesc)$ be another entity type. Let $CPersons$ and $COrders$ be entity sets of types $CPerson$ and $COrder$, 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 $CPerson$.

**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 $e = \text{treat } e'$ as $\tau$ then $Var(e, r) = Var(e', r)$. In this case, if $\tau$ 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 $\tau$, 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, \( \text{expr_type}(w, r) = \text{expr_type}(f, r) \) and \( \text{Var}(w, r) = \text{Var}(f, r) \).

- If \( w \) is a field expression then we do as follows. It should be easy to see that \( w = e.1 \cdot i_2 \ldots i_k \) for some entity expression \( e \). If \( \text{expr_type}(e, r) = \text{id}(\tau_1, \ldots, \tau_n) \) then we define \( \text{expr_type}(w, r) \) to be the type of the sub-attribute \( \text{id}(i_1 \ldots i_k) \). If \( e \) does not contain \( * \) then \( \text{Var}(w, r) = s.1 \cdot i_2 \ldots i_k \) where \( s = \text{Var}(e, r) \). If \( e \) contains \( * \) then \( \text{Var}(e, r) = \text{Null} \).

**Example 2:** Let \( r \) be the ERA expression \( \rho_s(C\text{Persons}) \times C\text{Orders} \). It should be easy to see that \( \text{expr_type}(s, r) \) is \( C\text{Person} \) and \( \text{expr_type}(\text{treat s as CCustomer, r}) \) is \( C\text{Customer} \).

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

**Types and variables of ERA expressions**

For an ERA expression \( r' \) we define a sequence of variables \( \text{Var}(r') \) and a sequence of types, \( \text{era_type}(r') \), representing the types of these variables in that order. These definitions are given by induction.

- \( r' = \text{eid}: \text{Var}(r') = (\text{eid}.1, \ldots, \text{eid}.n) \) and \( \text{era_type}(r') = (\tau_1, \ldots, \tau_n) \) where \( \text{type}(\text{eid}) = \text{id}(\tau_1, \ldots, \tau_n) \).
- \( r' = \rho_s(r): \text{Var}(r') = (s.1, \ldots, s.n) \) where \( n \) is the length of \( \text{Var}(r) \) and \( \text{era_type}(r') = \text{era_type}(r) \).
- \( r' = \sigma_i(r): \text{Var}(r') = \text{Var}(r) \) and \( \text{era_type}(r') = \text{era_type}(r) \).
- \( r' = \Pi_{p_1, \ldots, p_k}(r): \text{Var}(r') = (x_1, \ldots, x_k) \) and \( \text{era_type}(r') = (\tau_1, \ldots, \tau_k) \) where for \( i, 1 \leq i \leq k, \tau_i = \text{expr_type}(p_i, r) \) and \( x_i \) is as defined below. If \( \text{Var}(p_i, r) \neq \text{Null} \) then \( x_i = \text{Var}(p_i, r) \), otherwise \( x_i = z_j \) for a new variable \( z_j \).
- \( r' = r_1 \times r_2: \text{Var}(r') = (x_1, \ldots, x_m, y_1, \ldots, y_n) \) where \( (x_1, \ldots, x_m) = \text{Var}(r_1) \) and \( (y_1, \ldots, y_n) = \text{Var}(r_2) \). Similarly, \( \text{era_type}(r') = (\tau_1, \ldots, \tau_m, \tau'_1, \ldots, \tau'_n) \) where \( (\tau_1, \ldots, \tau_m) = \text{era_type}(r_1) \) and \( (\tau'_1, \ldots, \tau'_n) = \text{era_type}(r_2) \).
- \( r' = r_1 \cup r_2 \) or \( r' = r_1 \setminus r_2 \): Here we assume that \( \text{era_type}(r_1) = \text{era_type}(r_2) \).

If \( \text{Var}(r_1) = \text{Var}(r_2) \) then \( \text{Var}(r') = \text{Var}(r_1) \), otherwise \( \text{Var}(r') = (y_1, \ldots, y_n) \) where \( n \) is the length of \( \text{Var}(r_1) \) and \( y_1, \ldots, y_n \) are entirely a new set of variables.

**Legality of Expressions and Conditions**

Now, for an ERA expression \( r \), and a \( w \) which is an entity or a property or a field expression, we define whether \( w \) is legal in \( r \). For an entity expression \( w \) that has no \( * \) in it, \( w \) is legal in \( r \) if the entity variable \( \text{Var}(w, r) \) references an entity set in \( r \); for an entity expression \( w \) that has \( *p \) appearing in it, \( w \) is legal in \( r \) if the property expression \( p \) is legal in \( r \).

Consider a property expression \( w = p \). If \( p \) is of the form \( k & e \) then \( p \) is legal in \( r \) if the entity expression \( e \) is legal in \( r \). If \( p = f \) where \( f \) is a field expression then \( p \) is legal in \( r \) if \( f \) is legal in \( r \). If \( f \) is a field expression then it is of the form \( e.i_1 \ldots i_k \) where \( e \) is an entity expression. Let \( s = \text{Var}(e, r) \). If \( e \) does not contain \( \text{treat expression or } * \) then \( f \) is legal in \( r \) if \( s.i_1 \ldots i_k \) is in \( \text{Var}(r) \).
or is a sub-attribute of a variable in \( Var(r) \). If \( e \) contains \( * \) or \( treat \) then \( f \) is legal in \( r \), if \( e \) is legal in \( r \) and \( id.i_1 \ldots i_k \) is an attribute or a sub-attribute of the entity type \( expr.type(e,r) = id(\tau_1, \ldots, \tau_n) \).

A condition \( c \) is legal in \( r \) if every property mentioned in it is legal in \( r \) and every entity expression (which is not part of a property expression) is legal in \( r \) (e.g., as in \( isof(e, \tau) \) condition).

**Example 3:** The entity expressions \( s, COrders \) are legal in the ERA expression \( r = \rho_s(CPersons) \times COrders \). Also the property expression \( (treat \ s \ as \ CCustomer).address \cdot \ state \) is legal in \( r \). The property \( s.id \) is not legal in \( \Pi_{s, name}(r) \) where \( r \) is given above.

We define **well formed** ERA expressions as follows. An entity set \( eid \) is a well formed ERA expression. \( \rho_s(r') \) is well formed if \( r' \) is well formed. The ERA expression \( \sigma_c(r') \), is well formed if \( r' \) is well formed and every property appearing in \( c \) is legal in \( r' \). The ERA expression \( r = \Pi_{p_1, \ldots, p_l}(r') \) is well formed if \( r' \) is well formed, each of \( p_1, \ldots, p_l \) is legal in \( r' \) and all variables in \( Var(r) \) are distinct. The ERA expression \( r = r_1 \times r_2 \) is well formed if \( r_1, r_2 \) are well formed and all variables in \( Var(r) \) are distinct. The ERA expressions \( r_1 \cup r_2, r_1 \backslash r_2 \) are well formed if \( era.type(r_1) = era.type(r_2) \). We always assume that the ERA expressions we consider are well formed.

### Actual Semantics.

We use the concept of nested tuples to define the semantics of ERA expressions. Let \( \tau = (\tau_1, \ldots, \tau_n) \) be a sequence of types. We say that sequence \( a = (a_1, \ldots, a_n) \) is a value tuple of type \( \tau \) if for every \( i, 1 \leq i \leq n \), the following holds: if \( \tau_i \) is a base type then \( a_i \) is a value of this type; if \( \tau_i \) is of complex type \( id(\epsilon_1, \ldots, \epsilon_k) \) then \( a_i \) is a value tuple of type \( (\epsilon_1, \ldots, \epsilon_k) \). We say that \( a \) is a flat/simple tuple, if each \( \tau_i \) is a base type.

Let \( u \) be an entity set and \( type(u) = id(\tau_1, \ldots, \tau_m) \). This entity set has \( m \) properties and we refer to the \( i^{th} \) property by the name \( u.i \). An entity \( e \) of type \( type(u) \) is a value tuple of type \( (\tau_1, \ldots, \tau_n) \); some times we may say that the type of \( e \) is the tuple \( (\tau_1, \ldots, \tau_m) \). If \( \tau_1 \) is a complex type, then we refer to the \( j^{th} \) component of the \( i^{th} \) property as \( u.i \cdot j \). An entity set can contain entities of its subtypes which are longer tuples.

A database state \( \delta \) is a mapping that associates with each entity set \( u \) a set of entities each of which is either of type \( type(u) \) or of any subtype of \( type(u) \). The semantics of ERA expressions is given by a function \( G \) that given a database state \( \delta \) and ERA expression \( r' \), gives a set of tuples \( G(\delta, r') \) of type \( era.type(r') \). We define this semantics inductively on the structure of \( r' \).

- If \( r' \) is \( eid \) and \( type(eid) = id(\tau_1, \ldots, \tau_n) \) then \( G(\delta, r') = \{ (d_1, \ldots, d_n) : \exists \) an entity \( (d_1, \ldots, d_m) \in \delta(eid) \) where \( m \geq n \} \); note that for entities whose type is a subtype of \( type(eid) \) their properties that are not in \( type(eid) \) are left out; such properties can be retrieved by using the key part of the entity which is present in the first \( n \) properties.
- If \( r' \) is \( \rho_s(r) \) then \( G(\delta, r') = G(\delta, r) \).
– If \( r' \) is \( \sigma_c(r) \) then \( G(\delta, r') \) is exactly the set of those tuples in \( G(\delta, r) \) that satisfy the condition \( c \) (later we define the satisfaction of such a condition by a tuple).
– If \( r' \) is \( r_1 \times r_2 \) then \( G(\delta, r') \) is the set of tuples obtained by concatenating a tuple in \( G(\delta, r_1) \) with a tuple in \( G(\delta, r_2) \) in that order.
– If \( r' \) is \( H_{p_1, \ldots, p_k}(r) \) then \( G(\delta, r') \) is the set of tuples \( (d_1, \ldots, d_k) \) such that for some tuple \( t \in G(\delta, r) \), \( d_i \) \((i = 1, \ldots, k)\) is the values of \( p_i \) in the tuple \( t \).
– If \( r' \) is \( r_1 \cup r_2 \) then \( G(\delta, r') = G(\delta, r_1) \cup G(\delta, r_2) \). If \( r' \) is \( r_1 \setminus r_2 \) then \( G(\delta, r') = G(\delta, r_1) \setminus G(\delta, r_2) \).

The semantics of entity, property and field expressions are defined using two functions \( \psi \) and \( \psi' \). For a tuple \( t \in G(\delta, r) \) and an expression \( w \), where \( w \) is an entity or property or a field expression, \( \psi(t, w) \) gives the value of \( w \) in \( t \); the type of this value is \( expr_type(w, r) \). These value are defined only for the cases when \( w \) is legal in \( r \). The function \( \psi' \) is defined only when \( w \) is an entity expression. In this case, \( \psi'(t, w) \) gives the entity set to which the entity \( \psi(t, w) \) belongs. If the entity expression \( w \) has no \( * \) in it then \( \psi'(t, w) \) is the entity set referenced by \( Var(w, r) \) (\( i.e. \), it can be syntactically determined from \( r \) and \( w \)); however, when \( w \) has \( * \) in it (\( i.e. \), dereferencing) then \( \psi'(t, w) \) can not be syntactically determined. For the case when \( w \) is a property or a field expression, \( \psi(t, w) \) is a single element (or, equivalently a nested tuple having one element).

For an entity expression \( e' \), \( \psi(t, e') \) and \( \psi'(t, e') \) are defined as follows.

– If \( e' = s \) then \( \psi'(t, e') \) is the entity set represented by \( s \) in \( r \) and \( \psi(t, e') \) is the tuple consisting of the values of the variables \( s_1, \ldots, s_n \) in the tuple \( t \) (here the entity set \( s \), or the entity set represented by \( s \), has \( n \) attributes);
– If \( e' = \text{treat } e \) as \( s \) then \( \psi'(t, e') = \psi'(t, e) \) and \( \psi(t, e') \) is defined as follows. If \( s \) is a super type of \( expr_type(e, r) \) then \( \psi(t, e') = \psi(t, e) \). Consider the case when \( s \) is a sub-type of \( expr_type(e, r) \). Let \( \psi'(t, e) = \text{eid } \), \( i.e. \), \( \psi(t, e) \) is an entity in \( \delta(eid) \). If there exists an entity in \( \delta(eid) \) of type \( s \) or of a sub-type of \( s \) having the same key values that are present in \( \psi(t, e) \) then additional attribute values not present present in \( \psi(t, e) \) are retrieved and added to it to obtain a tuple \( t' \); if no such entity exists Null values are added to obtain \( t' \). Now \( \psi(e', t) \) is defined to be \( t' \).
– If \( e' = *p \) then \( \psi(t, e') \) is the entity, \( i.e. \), tuple pointed to by the address \( \psi(t, p) \), and \( \psi'(t, e') \) is the entity set to which this entity belongs.

The definition of \( \psi \) in the case of property and field expressions are fairly straightforward. We only give the non-trivial cases.

– For a property expression \( p = &e \), \( \psi(t, p) \) is the address of entity \( \psi(t, e) \) in the entity set \( \psi'(t, e) \).
– If \( p = f \) then \( \psi(t, p) = \psi(t, f) \).
– For a field expression \( f \), we define \( \psi(t, f) \) as follows. Every field expression \( f \) is of the form \( e_i_1 \ldots i_k \). If the entity expression \( e \) is legal in \( r \) (this has to be the case when \( * \) or \( \text{treat} \) construct is in it) then \( \psi(t, f) \) is the value of the \( i^{th} \) entry or its sub.attribute in \( \psi(t, e) \). Otherwise, for some \( j \leq k \) and
$s = \text{Var}(e, r), s.i_1 \ldots i_j$ is a variable in $\text{Var}(r)$; $\psi(t, f)$ is the value of this variable or a sub-attribute of this variable in the tuple $t$.

The satisfaction of a condition $c$ in a tuple $t \in G(\delta, r)$ is defined as follows. Here we assume that $c$ is legal in $r$.

- Consider the case when $c$ is $\text{isof}(e, \tau)$. In this case, we require that $e$ does not contain the $\text{treat}$ expression; $t$ satisfies $c$ if there is an entity in the entity set $\delta(\psi'(t, e))$ that is of type $\tau$ or a sub-type of $\tau$ having the same key values as the entity $\psi(t, e)$.
- The condition $\text{isnull}(p)$ is satisfied in $t$ if $\psi(t, p)$ is null.
- The condition $p_1 = p_2$ is satisfied in $t$ if $\psi(t, p_1) = \psi(t, p_2)$.

5 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., $\mathcal{P}' = \{ U(s) : s \in \mathcal{E}' \}$. 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.

6 Translation of ERA to FOL

6.1 Translation

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

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 \( key(\tau) = (\tau_1, \ldots, \tau_n) \). 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 \( d_1, \ldots, d_k \) 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 \( \tau \) 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 \( X \), of the same length as \( free\_var(F) \), we let \( F(X) \) denote the formula obtained by substituting the variables in \( X \) for the free variables in \( F \) in the given order. For each identifier \( g \), we have a set of first order variables \( \{g.i : i \geq 1\} \). 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 \( Y = (s \cdot 1, \ldots, s \cdot n) \). This translation simply renames the free variables in \( \mathcal{F}(r) \) to be those in \( Y \). Formally, \( \mathcal{F}(r') = \mathcal{F}(r)(Y) \) and \( free\_var(\mathcal{F}(r')) = (Y) \).
- \( r' = \sigma_c(r) \): In this case, \( \mathcal{F}(r') = \mathcal{F}(r) \land \mathcal{F}(c) \) and \( free\_var(\mathcal{F}(r')) = free\_var(formula(r)) \).
- \( r' = \Pi_{p_1, \ldots, p_n}(r) \): In this case, for each \( i = 1, \ldots, n \), \( \mathcal{F}(p_i) \) is a single variable. Let \( C \) be the formula \( \bigwedge_{i=1}^n \mathcal{B}(p_i) \) and \( Y = \bigcup_{i=1}^n \mathcal{E}(p_i) \). Let \( X \) 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 \( Y \) or in \( \mathcal{F}(r) \) as a free variable. Then, \( \mathcal{F}(r') = \exists X (\mathcal{F}(r) \land C) \). 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 $F(r_1)$ and $F(r_2)$ do not have any common free variables. If this is not satisfied, the variables are renamed to satisfy this property. We define $F(r') = F(r_1) \land F(r_2)$ and \text{free\_var}(F(r')) is obtained by concatenating the vectors \text{free\_var}(F(r_1)) and \text{free\_var}(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 $F(r') = F(r_1) \lor F(r_2)$. If $r' = r_1 \setminus r_2$ then $F(r') = F(r_1) \land \neg F(r_2)$. Here we assume that \text{free\_var}(F(r_1)) = \text{free\_var}(F(r_2)). We define \text{free\_var}(F(r')) = \text{free\_var}(F(r_1)).

Now we define the translation for entity expressions. For an entity expression $e'$, $F(e')$, $E(e')$ and $B(e')$ 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 $F(e') = \text{Null}$, $B(e') = \text{true}$ and $E(e') = \emptyset$. If $e'$ is a legal entity expression in $r$ then the above value are defined as follows.

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

- $e' = \text{treat} e$ as $\tau$: Let $\tau = \text{id}'(\tau_1', \ldots, \tau_n')$ and let $F(e) = \text{id}(u, v \cdot 1, \ldots, v \cdot m)$ for some $u, v, m$. Here $u$ can be a constant or a variable. If $\tau$ is a super type of $\text{expr\_type}(e, r)$ then $F(e') = F(e)$, $B(e') = B(e)$ and $E(e') = E(e)$. Now, assume that $\tau$ is subtype of $\text{type}(u)$ and hence $n > m$. In this case, if the entity belongs to the subtype $\tau$ then the additional attributes of the entity are retrieved, otherwise null values are retrieved. This is done by defining $F(e')$, $B(e')$ as follows.

- $F(e') = \text{id}'(u, v \cdot 1, \ldots, v \cdot n)$;
- $B(e') = B(e) \land (\text{id}'(u, v \cdot 1, \ldots, v \cdot n) \lor g)$ where $g = \neg \exists y_{m+1}, \ldots, y_n \text{id}'(u, v \cdot 1, \ldots, v \cdot m, y_{m+1}, \ldots, y_n) \land m < i \leq n (v \cdot i = \text{cNull})$;
- $E(e') = E(e) \cup \{v \cdot j : m < j \leq n\}$.

- $e' = \ast p$: Let $\text{expr\_type}(p, r) = \text{ref}_r$ where $\tau = \text{id}(\tau_1, \ldots, \tau_n)$, $\text{key}(\tau) = (i_1, \ldots, i_k)$ and $v$ and $j$ be new names. Recall that $\text{key}(\tau)$ gives the attributes of the key. In this case $F(p)$ is a variable. Using the relation $\text{pkey}_r$ that relates pointers to key values of entities, we retrieve the key name and the entity. If this is a dangling pointer then we retrieve $\text{Null}$ values. This is done by appropriately defining $B(e')$.

- $F(e') = \text{id}(j, v \cdot 1, \ldots, v \cdot n)$;
- $B(e') := B(p) \land (g \lor h)$ where $g = \text{pkey}_r(j, F(p), v \cdot i_1, \ldots, v \cdot i_k) \land \text{id}(j, v \cdot 1, \ldots, v \cdot n)$ and $h = (\neg \exists j', y_1, \ldots, y_k) \text{pkey}_r(j', F(p), y_1, \ldots, y_k) \land 1 \leq i \leq n (v \cdot i = \text{cNull})$;
- $E(e') = E(p) \cup \{j, v \cdot 1, \ldots, v \cdot n\}$.

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

- $p = f$: In this case, $F(p) = F(f)$, $B(p) = B(f)$ and $E(p) = E(f)$.
- $p = \&e$: let $F(e) = id(u, v \cdot 1, \ldots, v \cdot n)$ 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 $pkey_r$, we retrieve the pointer to $e$ by appropriately defining $B(e)$.
- $F(p) = x$;
  - $B(p) = B(e) \land pkey_r(u, x, v \cdot i_1, \ldots, v \cdot i_k)$;
  - $E(p) = E(e) \cup \{x\}$.

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

- $f' = e.k$: Here we have two cases. In the first case, $F(e) = Null$; in this case $e$ is not legal but $e.k$ or a sub-attribute of it is a variable in $Var(r)$; we define $F(f') = e.k, B(f') = true$ and $E(f') = \emptyset$. In the second case, $F(e) = id(u, v \cdot 1, \ldots, v \cdot n)$ for some $u, v, n$. We define $F(f')$ to be $v \cdot k$,
  - $B(f') = B(e)$ and $E(f') = E(e)$.
  - $f' = f \cdot k$: If $f$ is not legal in $r$ then $F(f') = f \cdot k, B(f') = true$ and $E(f') = \emptyset$. 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 $B(f')$.
  - $F(f') = f \cdot k$;
  - $B(f') = B(f) \land g$ where $g = id(f, f \cdot 1, \ldots, f \cdot n) \lor (\neg \exists y_1, \ldots, y_n \cdot id(f, y_1, \ldots, y_n) \land \bigwedge_{j=1}^{n}(f \cdot j = \text{cNull}))$. 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 $id$;
  - $E(f') = E(f) \cup \{f \cdot 1, \ldots, f \cdot n\}$.

Now we define $F(c), B(c)$ and $E(c)$ 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.

- $c = \text{IsNull}(p)$: In this case,
  - $F(c) = \exists \mathcal{E}(p)(B(p) \land (F(p) = c\text{Null}))$.

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

### 6.2 Correctness

In this subsection, we state and prove the correctness of the translation given above. Intuitively, we want to state that for every ERA expression $r$, the set of tuples retrieved by $r$ is same as the set of tuples that satisfy the formula $F(r)$. However, the databases they operate on are of different types and the type of sequences they define are different (nested tuples vs. flat tuples). For this reason, we give a formal definition of correctness taking the different models into
consideration. Recall that $D$ is the EDM database and $E$ is the corresponding CDB (Canonical Relational Database). Let $\delta$ and $\eta$ states of the databases $D$ and $E$, respectively. We say that they are valid if they satisfy all the integrity constraints. We only consider valid database states. Let $\delta$ and $\eta$ be valid states of the $D$ and $E$, respectively.

Let $a = <a_1, \ldots, a_n>$ be a tuple of type $\tau = (\tau_1, \ldots, \tau_n)$. Let $b = (b_1, \ldots, b_n)$ be some flat tuple. We say that $a$ corresponds to $b$ with respect to $\tau$ if for every $i$, $1 \leq i \leq n$, the following conditions are satisfied.

- If $\tau_i$ is a base type then $a_i = b_i$.
- If $\tau_i$ is of complex type $id(\tau'_1, \ldots, \tau'_k)$ then there exists a row $(b_i, c_1, \ldots, c_k)$ in the table $\eta(id)$ (i.e., it is pointed to by $b_i$) such that the nested tuple $a_i$ corresponds to the tuple $(c_1, \ldots, c_k)$ with respect to $(\tau'_1, \ldots, \tau'_k)$ in the database states $\delta, \eta$.
- If $\tau_i$ is of type pointer to an entity in the entity set $u$ of type $\tau' = id(\tau'_1, \ldots, \tau'_m)$ then there is a row $(u, b_i, c_1, \ldots, c_k)$ in the table $pkey_{\tau'}$ such that the key of the entity pointed to by $a_i$ is $(c_1, \ldots, c_k)$.

Let $u$ be an entity set and $\tau' = id(\tau'_1, \ldots, \tau'_m)$ be either same as $type(u)$ or is a subtype of $type(u)$. We say that a row $w = (u, b_1, \ldots, b_m) \in \eta(id')$ is unique if there is no row of the form $(u, c_1, \ldots, c_l) \in \eta(id'')$ having the same key values as $w$ for some subtype $id''(\tau''_1, \ldots, \tau''_k)$ of $\tau'$. Intuitively, such a row corresponds to an entity of type $\tau'$.

We say that the database state $\delta$ corresponds to the database state $\eta$ if the following conditions hold for every entity set $u$:

- For every entity $a = <a_1, \ldots, a_m> \in \delta(u)$ of type $\tau' = id'(\tau'_1, \ldots, \tau'_m)$, there exists a row $(u, b_1, \ldots, b_m) \in \eta(id')$ such that $a$ corresponds to $(b_1, \ldots, b_m)$ with respect to $(\tau'_1, \ldots, \tau'_m)$ in the pair of database states $\delta, \eta$.
- Let $\tau' = id'(\tau'_1, \ldots, \tau'_m)$ be same as $type(u)$ or is a subtype of $type(u)$. Then, for every unique row $w = (u, b_1, \ldots, b_m) \in \eta(id')$ there exists an entity $a = <a_1, \ldots, a_m> \in \delta(u)$ such that $a$ corresponds to $(b_1, \ldots, b_m)$ with respect to the type $(\tau_1, \ldots, \tau_m)$ in the pair of database states $\delta, \eta$.

Let $C$ be a set of tuples of type $\tau$ and $D$ be any other set of flat tuples. We say that $C$ corresponds to $D$ with respect to $\tau$ if for every $a \in C$ there exists a $b \in D$ such that $a$ corresponds to $b$ with respect to $\tau$ in the pair of database states $\delta, \eta$ and vice versa, i.e., for every $b \in D$, there exists an $a \in C$ such that $a$ corresponds to $b$.

Let $F$ be a formula in the multi-sorted first order logic with base types $int, bool, strings$. We assume that the set of variables that appear free in $F$ are arranged in a sequence and we let free_var($F$) denote this sequence. (Note that for an ERA expression free_var($F(r)$) is defined together with the definition of $F(r)$). Let $X = free_var(F)$ and $I$ be an interpretation for $F$. Note that the domains of the three sorts are the sets integers, booleans and strings respectively. Let $X = (x_1, \ldots, x_n)$. An evaluation for $F$ is a sequence $(a_1, \ldots, a_n)$ such that $a_i$
is a value from the domain of the type of \( x_i \) for \( 1 \leq i \leq n \). Let \( \text{eval}(I, F) \) be the set of evaluations that satisfy \( F \) in the interpretation \( I \). We have the following lemma which is proved by a straightforward induction on the structure of \( r \).

**Lemma 2:** For a well formed ERA expression \( r \), the number of variables in \( \text{free} \_\text{var}(F(r)) \) is same as the number of variables in \( \text{Var}(r) \). For each \( i \), if the \( i^{\text{th}} \) variable in \( \text{Var}(r) \) is not a new variable, introduced in the definition of \( \text{Var}(r) \), then the \( i^{\text{th}} \) variable in both the vectors are identical.

Recall the definition of \( \text{era}_\text{type}(r') \) for a well formed ERA expression \( r' \). Now the correctness condition for our translation is given by the following theorem.

**Theorem 3:** For every relational algebra expression \( r' \) over an EDM database \( D \) and for every valid database state \( \delta \) of \( D \) and for every valid state \( \eta \) of the corresponding CDB \( E \), such that \( \delta \) corresponds to \( \eta \), it is the case that \( \mathcal{G}(\delta, r') \) corresponds to \( \text{eval}(\eta, F(r')) \) with respect to \( \text{era}_\text{type}(r') \).

The above theorem is proved by a sequence of lemmas as given below. In all these lemmas and proofs, the type with respect to which the correspondence of sequences (or sets of sequences) is stated is obvious from the context and is left out.

**Lemma 4:** Let \( r \) be an ERA expression and \( w \) be an entity, or a property, or a field expression that is legal in \( r \).

1. The set of variables \( \mathcal{E}(w) \) is disjoint from \( \text{free} \_\text{var}(r) \).
2. Every variable occurring in either \( \mathcal{B}(w) \) or in \( \text{formula}(w) \) is present either in \( \mathcal{E}(w) \) or \( \text{free} \_\text{vars}(r) \).
3. If \( w \) is an entity expression then \( F(w) \) is of the form \( \text{id}(u, v.1, \ldots, v.n) \) where \( \text{id} \) is \( \text{expr}_\text{type}(w, r) \).
4. If \( w \) is a property or a field expression then \( F(w) \) is a variable.
5. If \( s \) references an entity set \( \text{eid} \) in \( r \) and \( \text{eid} \) has \( n \) properties then \( s.1, \ldots, s.n \) appear in \( \text{free} \_\text{var}(F(r)) \).

Note that in the above lemma, we assume that \( F(w), \mathcal{B}(w) \) and \( \mathcal{E}(w) \) are defined in the context \( r \). The first four parts of the lemma can be proved by a simple induction on the structure of the expression \( w \) and is straightforward. Part (5) of the lemma can be seen as follows. Since \( s \) references \( \text{eid} \) in \( r \), by definition, there is a sub-expression \( r'' \) of \( r \) which renames \( \text{eid} \) by \( s \) and such that \( r'' \) does not appear in the scope of a projection, another rename, \( \cup \) or \( \setminus \). Suppose \( \text{type}(\text{eid}) = \text{id}(\tau_1, \ldots, \tau_n) \). It should be easy to see that \( F(r'') \) is \( \text{id}(\text{eid}, s.1, \ldots, s.n) \). Since \( s \) is not renamed again in \( r \), i.e., \( r'' \) does not appear in the scope of a rename, the variables \( s.1, \ldots, s.n \) are not renamed and further, since \( r'' \) does not appear in the scope of a projection, these variables are never quantified and remain as free variables in \( F(r) \). (Note that variables appearing free in \( F(r'') \) are quantified only when they appear in the scope of a projection; other existential quantifiers appear in \( F(r) \); however, these are introduced in the translation of entity and property expressions and are over a different set of variables, as shown by property (1)).

**Proof of theorem 3:**

The theorem is proved by induction on the structure of \( r' \). If \( r' = \text{eid} \) then the theorem follows from the fact that \( \delta \) corresponds to \( \eta \) and both \( \delta, \eta \) are
valid database states. If \( r' = \rho_r(r) \) then inductive step is easily seen from the definitions. The cases when \( r' = r_1 \times r_2 \) or \( r' = r_1 \cup r_2 \) or \( r' = r_1 \setminus r_2 \) are straight forward. The difficult cases are the selection and projections.

Consider the case when \( r' = \sigma_e(r) \). To prove the inductive step for this case, we need to prove a number of lemmas about the the translations \( \mathcal{F}(w), \mathcal{B}(w) \) and \( \mathcal{E}(w) \) for the case when \( w \) is an entity or a property or a field expression.

Let \( F \) be any first order formula. Recall that \( \text{free}_{\nu}(F) \) is a sequence consisting of the variables appearing free in \( F \) arranged in some order and an evaluation \( \theta \) is a sequence of values of the same length as \( \text{free}_{\nu}(F) \). If a variable \( x \) appears as the \( i \)th element in \( \text{free}_{\nu}(F) \) then it’s value in \( \theta \) is the \( i \)th element in \( \theta \). We refer to this value as \( \text{value}(\theta, x) \). For a sequence of variables \( y_1, \ldots, y_k \), we let \( \text{value}(\theta, y_1, \ldots, y_k) \) represent the sequence of values of \( y_1, \ldots, y_k \) in \( \theta \) in that order. As an example, assume \( \text{free}_{\nu}(F) = (x, s.1, s.2, z) \) and \( \theta = (100, 3, 5, 200) \). Now \( \text{value}(\theta, s.1, s.2) = (3, 5) \). Suppose \( F' = F \land g \) and \( g \) possibly contains additional free variables. An evaluation \( \theta' \) for \( F' \) is said to be an extension of the evaluation \( \theta \) for \( F \) if for every variable \( x \) in \( \text{free}_{\nu}(F) \), \( \text{value}(\theta', x) = \text{value}(\theta, x) \). Consider the \( \text{free}_{\nu}(F) \) and \( \theta \) as given above. Suppose \( \text{free}_{\nu}(F') = (x, s.1, s.2, z, y) \). Then the evaluation \((100, 3, 5, 200, 300)\) for \( F' \) is an extension of \( \theta \).

Now, we state the following critical lemma which states that the translations of entity, property and field expressions agree with their semantics. Recall that as inductive hypothesis, we assume that \( G(\delta, r) \) corresponds to \( \text{eval}(\eta, \mathcal{F}(r)) \). Let \( a, b \) be any two sequences, respectively, belonging to the sets \( G(\delta, r), \text{eval}(\eta, \mathcal{F}(r)) \) such that \( a \) corresponds to \( b \). Recall that here \( a \) is a nested tuple, while \( b \) is a flat tuple. For any entity expression \( e, \mathcal{F}(e) \) is of the form \( \text{id}(u, v \cdot 1, \ldots, v \cdot n) \) in which \( u \) may be a variable or a constant (entity set). If \( b' \) is an evaluation for a formula containing \( \mathcal{F}(e) \) and \( u \) is a variable then \( \text{value}(b', u) \) is the value of \( u \) in \( b' \); if \( u \) is a constant then we let \( \text{value}(b', u) = u \). We want to show that \( a \) satisfies condition \( c \) iff \( b \) satisfies \( \mathcal{F}(c) \). In order to do this we use the following lemmas.

**Lemma 5:** Let \( a, b, r \) be as given above. The following hold.

1. Let \( e' \) be any entity expression legal in \( r \) and \( \mathcal{F}(e') = \text{id}(u, v \cdot 1, \ldots, v \cdot n) \) for some \( n > 0 \). Then, for every evaluation \( b' \) to the formula \( \mathcal{B}(e') \land \mathcal{F}(r) \) which is an extension of \( b \), \( b' \) satisfies \( \mathcal{B}(e') \) iff \( \psi(a, e') \) corresponds to \( \text{value}(b', v \cdot 1, \ldots, v \cdot n) \) and \( \psi(a, e') = \text{value}(b', u) \).

2. Let \( w' \) be any property or field expression legal in \( r \). Then, for every evaluation \( b' \) to the formula \( \mathcal{B}(w') \land \mathcal{F}(r) \) which is an extension of \( b \), \( b' \) satisfies \( \mathcal{B}(w') \) iff \( \psi(a, w') \) corresponds to \( \text{value}(b', \mathcal{F}(w')) \).

The first part of the lemma states that the tuple obtained by evaluating \( e' \) in \( a \) corresponds to the bindings to variables that satisfy \( \mathcal{B}(e') \). The second part says the same thing for a property and field expression. Note that, in the later case, \( \mathcal{F}(w') \) is a single variable.

**Proof of Lemma 5:**
The lemma is proved by simultaneous induction on the lengths of the expressions $e'$ and $w'$. To do this, for each $w$, which is an entity or a property or a field expression, we define $\text{length}(w)$ as follows.

- Consider the case when $w$ is an entity expression. If $w = s$ where $s$ is an entity set or a variable referencing an entity set in $r$, $\text{length}(w) = 1$. If $w = \text{treat } e \text{ as } \tau$ then $\text{length}(w) = 1 + \text{length}(e)$. If $w = \text{sp}$ then $\text{length}(w) = 1 + \text{length}(p)$.
- Consider the case when $w$ is a property expression. If $w = \&e$ then $\text{length}(w) = \text{length}(e) + 1$. If $w = f$ then $\text{length}(w) = \text{length}(f)$.
- Consider the case when $w$ is a field expression. If $w = w'' \wedge i$, where $w''$ is an entity or a field expression, then $\text{length}(w) = \text{length}(w'') + 1$.

We prove both parts (1) and (2) simultaneously, by induction on $\text{length}(e')$ and $\text{length}(w')$. The base case is when these lengths equal one. Observe that for a property or a field expression $w'$, $\text{length}(w') \geq 2$. So, it is enough if we prove the base case for part (1). In the base case of (1), $e' = s$. From the translation we see that $s$ references some entity set in $r$ and $\text{expr}_\text{type}(s,r) = \tau_1, \ldots, \tau_n$. Since $s$ references $u$ in $r$, from part (5) of lemma 4, $s, 1, \ldots, s, n$ are variables in $\text{free}_{\varphi}(\mathcal{F}(r))$ and hence are in $\text{Var}(r)$. Since $\mathcal{B}(e') = \text{true}$, it is enough if we show that $\psi(a, e')$ corresponds to $\text{value}(b', s, 1, \ldots, s, n)$ and $\psi'(a, e) = u$. Since $\mathcal{B}(e')$ has no additional variables, $b' = b$. Since $a$ corresponds to $b$, it is the case that $\text{value}(a, s, 1, \ldots, s, n)$ corresponds to $\text{value}(b, s, 1, \ldots, s, n)$. From the definition of $\psi$, we see that $\psi(a, e') = \text{value}(a, s, 1, \ldots, s, n)$. Hence $\psi(a, e')$ corresponds to $\text{value}(b, s, 1, \ldots, s, n)$. Also from the definition, $\psi'(a, e) = u$. Putting all this together, we get (1) of the lemma.

Now assume that the lemma is true for all $e', w'$ such that $\text{length}(e') \leq l$ and $\text{length}(w') \leq l$ where $l \geq 1$ is any integer. We show that the lemma holds for the cases when $\text{length}(e') = l + 1$ and $\text{length}(w') = l + 1$.

Now consider the case when $e' = \text{treat } e \text{ as } \tau$ where $\text{length}(e) = l$. Let $\mathcal{F}(e) = \text{id}(u, v \cdot 1, \ldots, v \cdot m)$. Let $b'$ be as given in the lemma. Assume $b'$ satisfies $\mathcal{B}(e')$. Since $\mathcal{B}(e') = \mathcal{B}(e) \land (\text{id}'(u, v \cdot 1, \ldots, v \cdot n) \lor g)$ where $n > m$ and $\text{id}'$, $g$ are as given in the translation, it is the case that $b'$ satisfies $\mathcal{B}(e)$ and hence from the induction hypothesis we see that $\psi(a, e)$ corresponds to $\text{value}(b', v \cdot 1, \ldots, v \cdot m)$ and $\text{value}(a, u) = \text{value}(b', u)$. From this and the fact that $b'$ satisfies $(\text{id}'(u, v \cdot 1, \ldots, v \cdot n) \lor g)$, it is easy to see that $\psi(a, e')$ corresponds to $\text{value}(b', v \cdot 1, \ldots, v \cdot n)$ and $\text{value}(a, u) = \text{value}(b', u)$. The reverse direction is also proved similarly.

Now consider the case when $e' = \text{sp}$ where $\text{length}(p) = l$. Observe that $\mathcal{B}(e') = \mathcal{B}(p) \land (g \lor h)$ where $g, h$ are as given in the translation. Now assume $b'$ satisfies $\mathcal{B}(e')$. This means $b'$ satisfies $\mathcal{B}(p)$. Using the induction hypothesis for part 2 of the lemma, we see that $\psi(a, p)$ corresponds to $\text{value}(b', \mathcal{F}(p))$ (note $\mathcal{F}(p)$ is a variable). Since $b'$ satisfies $(g \lor h)$, we see that $\text{value}(b', v \cdot 1, \ldots, v \cdot k)$ gives the key values of the entity whose pointer is given by $\text{value}(b', \mathcal{F}(p))$. From the way $\psi$ is defined and from the fact that $\delta$ corresponds to $\eta$, it can be seen that $\psi(a, e')$ corresponds to $\text{value}(b', v \cdot 1, \ldots, v \cdot n)$ and $\text{value}(a, u) = \text{value}(b', u)$. The reverse direction is also proved similarly.
We outline the proof, of the inductive step, for the second part of the lemma. Assume that \( w' = p \) is a property expression of the form \&\( e \) where \( length(e) = l \). We see that \( B(p) = B(e) \land \text{pkey}_e (u, x, v \cdot i_1, \ldots, v \cdot i_k) \) as given in the translation. Assume \( b' \) satisfies \( B(e) \). This means \( b' \) satisfies \( B(e) \) and by induction hypothesis, we see that \( \psi(a, e) \) corresponds to \( \text{value}(b', v \cdot 1, \ldots, v \cdot n) \). It should be easy to see, from the definition of \( \psi \) and the fact that \( \delta \) corresponds to \( \eta \) and the properties that \( \text{pkey}_e \) satisfies, that \( \psi(a, p) \) corresponds to \( \text{value}(b', x) \); note \( \mathcal{F}(p) = x \). The converse is similarly seen.

Now consider the case when \( w' = f' \) is a field expression. Obviously, we see that \( f' = e.i_1 \cdot \ldots \cdot i_k \) where \( length(e) \leq l \). Note that \( B(f') \) has \( B(e) \) as a conjunct. Here we have two cases. In the first case, \( e' \) is a legal entity expression in \( r \). It should be easy to see, by induction hypothesis, that \( \psi(a, e) \) corresponds to \( \text{value}(b', v \cdot 1, \ldots, v \cdot n) \) where \( \mathcal{F}(e) = \text{id}(u, v \cdot 1, \ldots, v \cdot n) \). From this, by a simple induction on \( k \), it can be shown that \( \psi(a, f') \) corresponds to \( \text{value}(b', \mathcal{F}(f')) \). Now consider the second case when \( e \) is not legal in \( r \). However, \( f' \) is legal in \( r \). This means that there exists a \( m \leq k \) such that \( f'' = e.i_1 \cdot \ldots \cdot i_m \) is legal in \( r \). Let \( m \) be the smallest such integer. It is not difficult to see that \( \mathcal{B}(f'') = \text{true} \) and \( \mathcal{E}(f'') = \emptyset \) and hence that \( \psi(a, f'') \) corresponds to \( \text{value}(b', \mathcal{F}(f'')) \). Using a similar argument as in the case when \( e \) is legal, it can be shown that \( \psi(a, f') \) corresponds to \( \text{value}(b', \mathcal{F}(f')) \). The converse in both cases is similarly proved. \( \square \)

Now, we have the following lemma.

**Lemma 6:** For every condition \( c' \) such that is legal in \( r \), the tuple \( a \) satisfies \( c' \) iff the evaluation \( b \) satisfies \( \mathcal{F}(c') \land \mathcal{F}(r) \).

We can prove the above lemma by induction on the structure of the condition \( c' \). In proving the base case, we use lemma 5. The induction step is straightforward.

Using lemma 6, we see that the tuple \( a \) satisfies condition \( c \) in \( r' \) iff \( b \) satisfies the formula \( \mathcal{F}(c) \land \mathcal{F}(r) \).

Now, we consider the last inductive step of theorem 3. So assume that \( r' = \Pi_{p_1,\ldots,p_n}(r) \).

Let tuples \( a, b \) be as given above. From lemma 5, we see that for every evaluation \( b' \) to the formula \( g = \bigwedge_{i=1}^n (\mathcal{B}(p_i)) \land \mathcal{F}(r) \) which is an extension of \( b \), it is seen that \( \psi(a, p_i) \) corresponds to \( \text{value}(b', \mathcal{F}(p_i)) \) for \( i = 1, \ldots, n \); from this we see that \( < \psi(a, p_1), \ldots, \psi(a, p_n) > \) corresponds to \( \text{value}(b', \mathcal{F}(p_1), \ldots, \mathcal{F}(p_n)) \). Hence, \( \mathcal{G}(\delta, r') \) corresponds to \( \text{eval}(\eta, \mathcal{F}(r')) \).

This completes the proof of correctness. It is to be noted that the requirement that every projection is followed by a renaming is needed to avoid confusion about the names of variables after projection. We can remove this requirement with some changes in the translation.

## 7 Implementation

Our implementation ROUNDTRIP takes as input the object and the database schema expressed in the EDM language [9], and the query and update views expressed in the ESQL language [11] and produces the first-order logic formulae.
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 [20], on the generated formula to verify correctness. Theorem provers such as E-PROVER try to prove correctness of a FOL formula \( F(e) \) by proving unsatisfiability of the formula \( \neg F(e) \). 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 \( \neg F(e) \).

**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, 6, 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 $k \ast (1 + a \ast \log_2(c))$.

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 [3] 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 Ghz 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 [10] for SAT solving.

Figure 2 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 (marked by an asterix).

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, and “MG time” is the time taken by our custom model generator. The model generator is run only if E-PROVER is not able to finish in 120s. “Num clauses” and “Num literals” give some measure 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.

8 Discussion

Formal methods have been successful in validating hardware [4,15], low-level software [2], 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 [21,1]. However, the relationship between object oriented extensions to SQL and first order logic
has not been studied. Our definition of Extended Relational Algebra, and its translation to first order logic are nontrivial and novel.

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) [19]. Query equivalence has also been studied for such restricted fragments with extensions such as Datalog [16] or aggregate queries [7]. 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.

References

An object relational map verification system is described. In some embodiments, the object relational map verification system can verify object relational maps and identify counterexamples when an object relational map cannot be verified. The object relational map verification system can verify an object relational map by (1) receiving objects, database schemas, query views, and update views; (2) generating first order logic formulae corresponding to the received objects, database schemas, query views, and update views; and (3) proving theorems indicated by the generated first order logic formulae. When the theorems are proved, the object relational map is verified. In some embodiments, the object relational map verification system can also generate models illustrating counterexamples when the theorem cannot be proved. The counterexamples provide data that the object relational map does not consistently store and then retrieve.

17 Claims, 3 Drawing Sheets
start

receive source code for object

receive database schema

receive database views

produce first order logic formulae

invoke theorem prover

did theorem prover time out?

Y → report error

N → generate model

return

FIG. 2
流程图

1. 302: start
2. 304: receive first order logic formula
3. 306: translate formula into satisfiability problem instance
4. 308: search for model
5. 310: model found?
   - Y: 314: provide results
   - N: 312: revise parameters
6. 316: return

**FIG. 3**
OBJECT RELATIONAL MAP VERIFICATION SYSTEM

BACKGROUND

Producing sophisticated software can involve many man-hours of software developers’ effort to produce software code. To improve the understandability and maintainability of software code, software developers began using object-oriented programming techniques. Software code that was developed using object-oriented programming techniques generally encapsulated functionality into objects. As an example, when software accesses a database to store or retrieve data, the software may contain a database access object that other objects employ to access the database. Objects generally have attributes and methods. Attributes can store values. Other code in the software, such as other classes, can invoke the methods to manipulate the object’s attributes or take various actions.

Objects can define complex data relationships. As an example, an object representing an address book can reference objects corresponding to multiple person objects, with each person object having an associated name, multiple addresses, multiple phone numbers, and so forth. The person object may include a method to return a preferred telephone number. The person object can be referenced as a single value in a variable. In contrast, a database can be defined using a database schema comprising tables and columns that is relational and so different from objects. Data can be stored in rows of each database table. Each row’s data can comprise scalars, such as strings, integers, and so forth. No one column of a table contains the equivalent of a person object. Instead, the person object’s attributes can be distributed among several columns, such as name, address, phone number, and so forth.

When developing a database access object, developers generally develop code to access the database. As an example, to store and retrieve address book entries, a developer may develop a database object that maps data from a person object’s attributes to database column values and vice versa. However, every time the object’s structure or the database schema changes, the code may need to be modified to conform to the change. The code would need to determine which tables and columns are needed to satisfy a particular database operation and produce corresponding database queries. This can involve substantial software development and testing effort.

Developers sometimes employ object relational mapping (ORM) techniques to simplify the understandability and maintainability of objects that access databases or, more generally, anytime two objects of different types exchange data. An object relational map identifies the mapping between two object types, such as a database object that accesses a database and the database itself. ORM techniques can be employed whenever transformation of data occurs from one complex type to another. ORM techniques map objects to database schemas to produce database queries automatically so that developers do not need to develop and maintain code for accessing databases. Using such techniques, a developer can map an object’s attributes to database columns. ORM techniques produce suitable database access queries when an object accesses a corresponding database. However, developers and testers need to ascertain that the ORM techniques are producing correct database queries. In other words, developers need to know that an object relational map causes data that is stored to be equivalent to data that is retrieved. This is known in the art as a “round-trip” data analysis.

SUMMARY

An object relational map verification system is described. The object relational map verification system can verify object relational maps and identify counterexamples when an object relational map cannot be verified. The object relational map verification system can verify an object relational map by (1) receiving objects, database schemas, query views, and update views; (2) generating first order logic formulae corresponding to the received objects, database schemas, query views, and update views; and (3) proving theorems indicated by the generated first order logic formulae. When the theorems are proved, the object relational map is verified. The object relational map verification system can also generate models illustrating counterexamples when the theorem cannot be proved. The counterexamples provide data to indicate that the object relational map does not consistently store and then retrieve.

This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating components of an object relational map verification system in various embodiments.

FIG. 2 is a flow diagram illustrating a verify routine invoked by the object relational map verification system in some embodiments.

FIG. 3 is a flow diagram illustrating a generate_model routine invoked by the object relational map verification system in some embodiments.

DETAILED DESCRIPTION

An object relational map verification system is described. The object relational map verification system verifies that an object relational map is valid. An object relational map is invalid when data that is retrieved to repopulate the attributes is not equivalent to the data that is stored based on an object’s attributes. In various embodiments, the object relational map verification system verifies an object relational map by (1) receiving objects, database schemas, query views, and update views; (2) generating first order logic formulae corresponding to the received objects, database schemas, query views, and update views; and (3) proving theorems indicated by the generated first order logic formulae. When the theorems are proved, the object relational map is verified. The object can be received in object code or source code. In some embodiments, the object relational map verification system generates a first order logic formula based on the object code or source code. In some embodiments, the object relational map verification system can receive first order logic formulae from software developers. The object relational map verification system can receive database schemas specifying portions of databases into which attributes of the received object will be stored or from which the attributes will be retrieved. In some embodiments, the object relational map verification system can determine database schemas, such as by querying a database management system. The query views specify in a query language, such as in ESQL, the information that is retrieved from the database. ESQL is a type of structured query language that relational database management systems imple-
As an example, the query view may be an ESQL SELECT statement that identifies tables and columns of a database that match a specified condition that are to be retrieved to populate an object's attributes. The update views specify information that is stored in the database. As an example, the update view may be an ESQL UPDATE statement that identifies the tables and columns that are to be updated based on the object's stored attributes. A first order logic formula is a formula that is defined by a predicate calculus comprising formation rules, transformation rules, and axioms. In various embodiments, the first order logic formula can be derived by the object relational map verification system based on the object, database schema, query view, and update view. The first order logic formula can subsume predicate logic by adding variable quantification, and can include axioms that are provided and theorems that need to be proven. The first order logic formula expresses the object relational map formally. The object relational map verification system can then prove the theorem indicated by the first order logic formula. As an example, the object relational map verification system can employ a theorem prover to prove the formula. When the theorem is proved, the object relational map is verified.

In some embodiments, the object relational map verification system generates models illustrating counterexamples. The counterexamples provide data that the object relational map does not consistently store and then retrieve. As an example, if the theorem cannot be proved, the object relational map verification system can generate examples of data that illustrate how the object relational map is incorrect. The object relational map verification system can generate counterexamples by translating the generated first order logic formula into a first order logic formula instance and searching for the model by using a Boolean satisfiability problem solver. A Boolean satisfiability problem solver receives an expression comprising multiple variables and identifies values for the variables so that the formula evaluates to true (or false). The object relational map verification system can parameterize (e.g., vary) (1) a maximum bound on the number of “tuples” in every relation k and (2) the number of constants in a domain c. A tuple is a sequence of objects, such as variables. The object relational map verification system encodes the first order logic formula into a quantified Boolean formula that can be converted to a satisfiability problem expression. The object relational map verification system can employ binary decision diagrams to optimize quantifications. Binary decision diagrams are described by R. E. Bryant, “Graph-Based Algorithms for Boolean Functional Manipulation,” IEEE Trans. Computers 35(8):677-691, which is incorporated herein by reference. The object relational map verification system can convert binary decision diagrams into a set of conjunctive normal form clauses. A conjunctive normal form clause is a conjunction of Boolean clauses. A satisfiability problem solver can then provide a model based on the conjunctive normal form clauses by assigning variables so that the expression resolves to a true (or false) result. If the satisfiability problem solver is unable to identify a model, the object relational map verification system can increase the k and c values and re-attempt to identify a solution.

Thus, the object relational map verification system can verify object relational maps and identify counterexamples when an object relational map cannot be verified. The object relational map verification system will now be described using examples. Suppose an entity set Orders(id, amount) has 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.

Suppose the object relational map verification system is given the following update view for SmallOrders:

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

Further, suppose that the object relational map verification system is given the following update view for BigOrders:

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

The object relational map verification system can translate these views into first order logic (FOL) formulae given in A1 and A2 in Table 1.

TABLE 1

<table>
<thead>
<tr>
<th>Axioms and Conjectures for Example 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 ( \forall x,y ) SmallOrders(x, y) ( \Leftrightarrow ) (Orders(x, y) ( \land ) ( y &lt; 100 ))</td>
</tr>
<tr>
<td>A2 ( \forall x,y ) BigOrders(x, y) ( \Leftrightarrow ) (Orders(x, y) ( \land ) ( y \geq 100 ))</td>
</tr>
<tr>
<td>A3 ( \forall x,y ) Ordersnew(x, y) ( \Leftrightarrow ) (SmallOrders(x, y) ( \lor ) BigOrders(x, y))</td>
</tr>
<tr>
<td>A4 ( \forall x,y ) BigOrders(x, y) ( \Leftrightarrow ) (Orders(x, y) ( \land ) ( y &gt; 100 ))</td>
</tr>
<tr>
<td>C ( \forall x,y ) Ordersnew(x, y) ( \Leftrightarrow ) Orders(x, y)</td>
</tr>
</tbody>
</table>

The object relational map verification system can translate the following query view for Orders:

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

into the FOL formula given in A3 of Table 1. Ordersnew is a new relation. The object relational map verification system can then formulate the round-trip verification condition as proving the conjecture C using the axioms A1, A2, and A3, which can be readily proved by using a conventional FOL theorem prover.

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

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

This translates to the FOL formula in A3' in Table 1. If the object relational map verification system attempts to prove conjecture C using the axioms A1, A2', and A3, it will be unable to do so because the theorem prover will fail or report an error.

By using a model generator, the object relational map verification system can generate a counter model: Orders \{- (1,100)\}, SmallOrders \{-\}, BigOrders \{-\}, and Ordersnew \{-\}.

While the above examples illustrate both correct and incorrect object relational maps, they are simplistic. Realistic
object relational maps include object-oriented features that complicate translation to FOL. To illustrate some of these complications, consider the following entity schema types: (1) CPerson(id, name), and (2) CCustomer(id, name, addr). 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 SPerson(id, name, state, zip). Further, suppose the source code defines one entity set CPersons in the object model of type CPersonType, and one entity set SPersons in the database of type SPersonType. Table 2 contains the axioms and conjectures for this example.

A complication in developing a FOL representation is that CPersons can have objects of type both CPersonType and CCustomerType. In some queries, an object of type CCustomerType might be cast into its supertype CPersonType, and then downcast into an object of type CCustomerType. To handle such cases uniformly, the object relational map verification system can 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 this current example, since there are two types CPersonType and CCustomerType, there can be two FOL relations CCustomerType(id, name) and CCustomerType(id, name, addr), where addr is of type AddressType. Conceptually, addr can be treated as being a foreign key that indexes in AddrType. Since AddrType is a complex type, the FOL relation can be represented as AddressType(a, state, zip), where a is the key of the relation. Inheritance is modeled using the axiom Ai, which states that every CCustomerType object is also a CPersonType object.

Suppose the object relational map verification system is given an update view U for SPersons as:

Translating this to FOL can be more complicated than in the pure relational case. Axiom A1 relates SPersons to CPersons, taking into account that objects in CPersons could be of type CPersonType or type CCustomerType. In the latter case, the last attribute of CCustomerType is a foreign key for the relation AddressType.

Then, the object relational map verification system can generate the FOL formulae for Q as in A1 and A2. To prove the round-trip condition, the theorem prover proves the conjecture C1, which states that CPersons and CPersons are equivalent using axioms A1, A2, A3, and A4.

In general, a view is defined by an ESQL command of the form SELECT p FROM r WHERE c, in which p, r and c may all contain object-oriented constructs. The object relational map verification system can translate such expressions to FOL formulae inductively using a function F from query expressions. F(e) is defined in terms of F(p), F(r), and F(c). To perform the translation inductively, the object relational map verification system maintains a list of variables F(e) that need to be existentially quantified and a formula B(e) that defines bindings to these variables. This inductive translation is described in further detail below.

Then, the object relational map verification system can generate the FOL formulae for Q as in A1 and A2. To prove the round-trip condition, the theorem prover proves the conjecture C1, which states that CPersons and CPersons are equivalent using axioms A1, A2, A3, and A4.

In general, a view is defined by an ESQL command of the form SELECT p FROM r WHERE c, in which p, r and c may all contain object-oriented constructs. The object relational map verification system can translate such expressions to FOL formulæ inductively using a function F from query expressions. F(e) is defined in terms of F(p), F(r), and F(c). To perform the translation inductively, the object relational map verification system maintains a list of variables F(e) that need to be existentially quantified and a formula B(e) that defines bindings to these variables. This inductive translation is described in further detail below.

Extended Relational Algebra

The following describes the syntax and semantics of an Extended Relational Algebra (ERA). The full formal semantics are described at the document referenced by "research.microsoft.com/~srimat/rndriv1r.pdf," which is incorporated herein by reference.

The syntax of the ERA can be described using Backus-Naur Form (BNF) notation given in Table 3.
The basic operators in the ERA are \( \sigma \) (Selection), \( \rho \) (Renaming), \( \Pi \) (Projection), \( \times \) (cross product), and the set union and difference. The syntax also has type casting, type checking, pointer referencing, and dereferencing, etc.

The grammar has six non-terminal symbols. Among these, \( \tau \) defines types. The object relational map verification system can assume that in the definition of a type the name id uniquely identifies the type. Each expression generated from a relation \( r \) is an ERA expression. Semantically, it denotes a set of entities or simply a relation. The non-terminal symbols \( e \), \( p \), and \( f \) define entity, property, and field expressions, respectively. Finally, \( c \) defines conditions. Each expression generated from the non-terminating symbol \( e \) 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. In the entity expressions, the expressions treat \( e \) as \( \tau \) casts an entity of a base type to one of its subtypes or vice versa. It also generates the address (\&) and dereferences an address (*). Note that "\( \cdot \)" is used to access attributes of an entity, while the "\( \cdot \)" notation is used to access subfields of a complex type.

**ERAS Semantics**

An EDM database \( D \) is a triple \((T, ESN, type)\), where \( T \) is a set of types, ESN is a set of entity sets, and type is a function that associates a type with each entity set. The object relational map verification system assumes that there is a function key that given an entity type \( \tau \) gives the fields in the entity that define the key for the entity sets of that type.

An ERA expression is a string generated from the non-terminating symbol \( r \). An ERA expression is said to be a rename operation if it is of the form \( \rho_r(e)\). A rename operation renames an entity set \( e \) with the variable \( s \) if either \( r \cdot e \) or \( r \cdot s \) itself renames \( e \) with some variable \( s \). A variable \( s \) refers to the entity set \( e \) in ERA expression \( r \) if there is a subexpression \( r \cdot f \) that renames \( e \) 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 \( \backslash \).

An as example, consider the entity type \( \text{CPersonType}(id, \text{name}) \) and its subtype \( \text{CCustomerType}(\text{addr} : \text{AddressType} \langle \text{state}, \text{zip} \rangle) \): \( \text{CPersonType} \) as described above. Let \( \text{COrderType}(\text{Oid}, \text{Desc}), \text{COrder} \) be another entity type. Let \( \text{CPerson} \) and \( \text{COrders} \) be entity sets of types \( \text{CPersonType} \) and \( \text{COrderType} \), respectively. In the ERA expression \( \rho_r(\text{CPersons}) \), \( \text{CPerson} \) refers to the entity set \( \text{CPersons} \). On the other hand, in the ERA expression \( \rho_r(\rho_s(\text{CPersons}))(\text{COrders}) \), \( \text{CPersons} \) does not refer to the entity set \( \text{CPersons} \).

For every \( w \) which is either an entity, or a property or a field expression, and for every ERA expression \( r \), the object relational map verification system can define \( \text{expr_type}(w, r) \), which is the type of \( w \) in \( r \), and also define \( \text{Var}(w, r) \), which is the variable that \( w \) denotes in \( r \). For an entity expression \( w \) that does not contain *, \( \text{Var}(w) \) is the entity set variable specified in \( w \). Furthermore, if \( w \) does not contain treat, then \( \text{expr_type}(w, r) \) is the entity type referenced by \( \text{Var}(w) \). If \( w \) contains treat, then \( \text{expr_type}(w, r) \) may be the type mentioned in the treat clause. If \( w \) contains *, then \( \text{Var}(w) = \text{Null} \); and \( \text{expr_type}(w, r) \) is the entity of the pointer points to. For an entity expression \( w = e \), \( \text{Var}(e) \) and \( \text{expr_type}(e, r) \) can be defined as follows:

If \( e = \text{then} \), \( \text{Var}(e) = r \). Furthermore, if \( e \) references an entity set in \( r \), then \( \text{expr_type}(e, r) = \text{type}(e) \).

If \( e = \text{treat} \) \( e \) as \( \tau \) then \( \text{Var}(e) = \text{Var}(r) \). In this case, if \( \tau \) is a subtype of \( \text{expr_type}(e, r) \) then \( \text{expr_type}(e, r) = \tau \); otherwise, \( \text{expr_type}(e, r) = \text{expr_type}(e) \).

If \( e = p \) and \( \text{expr_type}(p, r) \) then \( \text{Var}(e) = \text{Null} \) and \( \text{expr_type}(e, r) = \tau \).

For a property or a field expression \( w \), \( \text{Var}(w, r) \) and \( \text{expr_type}(w, r) \) can be defined as follows:

If \( w \) is the property expression \( \& e \), then \( \text{Var}(w, r) = \text{Null} \). In this case, if \( \text{expr_type}(e, r) = \tau \) then \( \text{expr_type}(w, r) = \tau \).

For a property expression \( w \), if \( w = f \) where \( f \) is a field expression, \( \text{expr_type}(w, r) = \text{expr_type}(e, r) \), and \( \text{Var}(w, r) = \text{Var}(e, r) \).

If \( w \) is a field expression then the object relational map verification system can do as follows:

- \( w = e \cdot i_1, \ldots, i_k \) for some entity expression \( e \). If \( \text{expr_type}(e, r) = \text{id}(i_1, \ldots, i_k) \) then \( \text{expr_type}(w, r) \) is defined to be the type of the subattribute \( \langle i_1, \ldots, i_k \rangle \).

If \( e \) does not contain * then \( \text{Var}(w, r) = \langle i_1, \ldots, i_k \rangle = \text{Var}(e, r) \) and \( \text{Var}(e, r) \) as \( \text{Null} \).

**Formulation**

Let \( P \) be the set of all physical database states, and \( E \) be a set of all entity database states. Consider two ERA relational expressions:

1. \( U : E \rightarrow P \) and \( P : P \rightarrow E \).

The round-trip condition from the entity side suggests that \( Q \rightarrow U \) 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. The pair of maps \( (U, Q) \) satisfy the round-trip condition from the entity side, if \( \forall e \in \text{EQ}(U(s)) \rightarrow s \) where \( E' \) is the set of all elements of \( E \) that satisfy the integrity constraints of the EDM data declaration and \( P' \) is the image of \( U' \) under the mapping \( U \) (e.g., \( P' = \{ U(s) : E' \} \)). This condition is equivalent to requiring that \( Q' \rightarrow U' \) be the identity function on domain \( E' \) where \( U' \) is the mapping, which is a restriction of \( U \) to \( E' \). Similarly, the pair \( (U, Q) \) satisfies the round-trip condition from the database side if \( \forall e \in \text{EQ}(Q(s)) \rightarrow s \). This condition provides the following Lemma that can be proved by a theorem prover:

**Lemma 1**: \( \forall e \in \text{EQ}(Q(s)) \rightarrow s \).
The object relational map verification system translates the ERA relational expressions U and Q into FOL, and uses FOL theorem prover to check the round-trip condition.

Translation of ERA to FOL

Corresponding to the EDM database D=(τ, ESN, type), a canonical relational database E can be defined. As indicated earlier, for each type τ=τ id (τ₁, ..., τₙ); there is a relation id(name, a₁, ..., aₙ) where name is an entity set (its name) if τ is an entity type; otherwise name is of type integer that is a key for the relation. If τ is of complex type τ id τ₂, ..., τₙ), then aₙ is of type integer and is a foreign key referencing the table id(τ₁, ..., τₙ).

Some complex types and subtypes are entity types. For each such type τ=τ id(τ₁, ..., τₙ), a relation key captures the one-to-one relationship between pointers to entities in entity sets and keys of such entities. The "arity" of key is k+2 where k=|τ id(τ₁, ..., τₙ)|. The object relational map verification system can check that the following are true: (a) p is the key of this relation (e.g., no two tuples have the same p values); (b) the attributes u₁, ..., uₙ form a key indicating that this is a one-to-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 key with the same values for u₁, ..., uₙ; and (d) for each entity set e and for every tuple t in key, such that t u-eid, there exists a tuple in key with the same key values as t₁, ..., tₙ.

For each relational expression r generated by r, define a FOL formula F(r). This formula has the property that in any database state s, the result of evaluating r on s provides a relation that is equivalent to the set of "tuples" that satisfy F(r) in the interpretation s. Similarly, for each expression f, which is an entity expression or property or field or condition, the object relational map verification system generates a formula F(f). For an expression f, which is an entity expression or property or field, the object relational map verification system generates a set E(f) of variables that need to be quantified and a formula B(f) that defines a binding on these variables. F(f), E(f), and B(f) not only depend on f, but they also depend on the ERA expression r in whose context they are being defined.

For any FOL formula F, let free_free_var(F) denote the sequence of free variables that appear in F. The ordering of these variables can be specified. For a sequence of variables X of the same length as free_var(F), let F(X) denote the formula obtained by substituting the variables in X for the free variables in F in the specified order. For each identifier g, there is a set of first order variables {g: i ∈ I₁}. For each relational expression r, F(r) is defined inductively based on its outer most connective, as follows. The object relational map verification system also defines a vector free_var(F(r)) of variables that appear free in F(r). r:id eid(F):id(eid, eid-1, ..., eid-n) and free_var(F(r))= (ein, ..., ein) where type(eid)=id(τ₁, ..., τₙ).

r:p=τ(r): F(r)=F(τ(r)) Y and free_var(F(r))= (Y) where n is the number of free variables in F(r) and Y=(e₁, ..., eₙ). This translation renames the free variables in F(r) to be those in Y.

r:τ'(τ): F(r)=F(τ(r)) C and free_var(F(r))=free_var(formula(τ)).

r:τ¹, ..., τₙ(r): F(r)=∃X(F(τ(r)) C where C is the formula "¬C", and Y=U, e F(p) X is the set of variables y such that y is not in {F(p): 1 ≤ i ≤ n}, and such that y appears either in Y or in F(r) as a free variable. F(p) is a single variable for each i=1, ..., n. Define free_var(F(r))= F(p₁), ..., F(pₙ).

r:τ x=r:τ x: In this case, the object relational map verification system assumes that theformula F(r) and F(x) do not have any common free variables. If this condition is not satisfied, the variables are renamed to satisfy this property. The object relational map verification system can define F(r)=F(r) ∨ F(x), and free_var(F(r))=F(x) obtained by concatenating the vectors free_var(F(r)) and free_var(F(x)) in that order.

r:τ¹, U τ x r:τ x: If F(r)=F(r) A F(x). If F(r) A F(r) and F(r)=F(r) A F(x). Here, the object relational map verification system assumes that F(x)=free_var(F(r))=free_var(F(r)). It defines free_var(F(r))=free_var(F(r)).

Now define the translation for entity expressions. For an entity expression e, F(e), E(e), and B(e) are defined inductively as follows. 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 F(e)=null, B(e)=true, and E(e)=∅. If e is a legal entity expression in r, then the values are defined as follows:

e:id(eid, s₁, ..., sₙ): B(e)=true, and E(e)=∅ where expr_type(s)=id(τ₁, ..., τₙ) and eid is the entity set referenced by s in r.

e=t: e then e: and e is a subtype of τ and hence n ≥ m. 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 F(e) and B(e) as follows:

F(e)=id(u, v₁, ..., vₙ); and
B(e)=B(e){v∈(u, v₁, ..., vₙ)} where

\[
E(e)e=e\{v: m\leq n\}
\]

e:τ: F(e)=F(τ) C and free_var(F(e))=free_var(formula(τ)).

e:p: Let expr_type(p)=e, where τ=id(τ₁, ..., τₙ), key=τ₁, ..., τₙ, and v and j be new names. key(e) gives the attributes of the key. In this case, F(p) is a variable. Using the relation key, which relates pointers to key values of entities, the object relational map verification system retrieves the key fields, the entity name, and the entity. If this is a dangling pointer, then the object relational map verification system retrieves NULL values. This is done by defining B(e) appropriately.

F(e)=id(v₁, ..., vₙ); and
B(e)=B(e){v∈(v₁, ..., vₙ)} where

\[
E(e)e=e\{v: m\leq n\}
\]

The object relational map verification system then translates property expressions. For a property expression p, F(p), B(p), and E(p) are defined as follows:

p: F(p)=F(f), B(p)=B(f), and E(p)=E(f).

p: Let expr_type(p)=e for some id, u, v, and n, where u can be a variable or an entity name. Let expr_type(e)=τ id(τ₁, ..., τₙ), key=τ₁, ..., τₙ, and E be a new first order variable. Using the key values of u and the relation key, the object relational map verification system can retrieve the pointer to e by appropriately defining B(e).
For any field expression \( f \), \( F(f') \), \( B(f') \), and \( E(f') \) are defined as follows:

\[
F(f) = \text{X};
\]

\[
B(f) = B(e) \cup \{ x_u, v_1, \ldots, v_n \};
\]

\[
E(f) = E(e) \cup \{ x \}.
\]

For any field expression \( f \), \( F(f') \), \( B(f') \), and \( E(f') \) are defined as follows:

\[
F(e) = k \text{ when } F(e) = \text{Null}, e \text{ is not legal but } e \text{ or } a \text{ sub}-
\]

\[
\text{attribute of it is a variable in } \text{Var}(e). \text{ In this case, the object}
\]

\[
\text{relational map verification system can define } F(f) = k,
\]

\[
B(f') = \text{true}, \text{ and } E(f) = \emptyset. \text{ When } F(e) = \text{id}(u, v_1, \ldots, v_n)
\]

\[
\text{for some } u, v, n, \text{ the object relational map verification system can define } F(f')
\]

\[
to be } k, B(f') = B(e), \text{ and } E(f') =
\]

\[
E(e).
\]

\[
f = k: \text{ If } f \text{ is not legal in } r, \text{ then } F(f') = f, B(f') = \text{true}, \text{ and }
\]

\[
E(f') = \emptyset. \text{ Otherwise, } f \text{ is an attribute of complex type }
\]

\[
\text{whose subfields are stored in the relation id and expr_type(id, expr)} = \text{id}(\tau_1, \ldots, \tau_n). \text{ The attribute } f \text{ is a key of this}
\]

\[
\text{relation and the subfield } f_k \text{ is retrieved from this relation. \text{This is achieved using } B(f').}
\]

\[
F(f) = f_k,
\]

\[
B(f) = B(f) \cup \{ g \} \text{ where } g = \text{id}(f, f_1, \ldots, f_n), \text{ and } \text{expr_type}(f, f_1, \ldots, f_n) = \text{id}(\tau_1, \ldots, \tau_n). \text{ The first}
\]

\[
\text{distinct in } g \text{ retrieves the subfields from the relation id and the second retrieves null values if no}
\]

\[
such tuple exists in id; and \text{ and } \\ E(f') = E(f') \cup \{ f_1, \ldots, f_n \}.
\]

\[
F(c), B(c), \text{ and } E(c) \text{ are defined for a condition } c. \text{ All the}
\]

\[
\text{bindings of various fields referenced in } c \text{ and the main formula and all existential variables are existentially}
\]

\[
\text{quantified.}
\]

\[
c = \text{IsNull}(p); \text{ F(c) = E(f) \cup B(p) \cup \text{forall}(F(p) = \text{Null})}, \text{ where } \pi \text{ = id}(\tau_1', \ldots, \tau_n') \text{ and } F(e) = \text{id}(u, v_1, \ldots, v_m) \text{ for some } u, v, m.
\]

The object relational map verification system will now be described with reference to the Figures. FIG. 1 is a block diagram illustrating components of an object relational map verification system in various embodiments. The object relational map verification system 100 can include source code 102 (or object code) specifying an object, a database 104 having a database schema, an analyzer component 106, a theorem prover component 108, and a model generator component 110. These components will now be described in further detail.

The source code or object code can employ an object relational map to map the object's attributes to information in the database schema. As an example, the code may employ a library of classes to map attributes to portions of a database, such as tables or columns. The database can be a relational or other type of database, such as a relational database that employs MICROSOFT SQL SERVER®. The database can also be a document, such as an extensible markup language (XML) document. An analyzer component can analyze source code, object code, or the database to produce one or more corresponding first order logic formulae. In some embodiments, the analyzer component can receive first order logic formulae from a software developer. The theorem prover component proves theorems associated with expressed first order logic formulae. In some embodiments, the object relational map verification system may employ conventional theorem provers. The model generator component can generate a model, such as a model having counter-examples that demonstrate that the object relational map is invalid. In some embodiments, the object relational map verification system may employ conventional satisfiability problem solvers to assist in generating models.

The computing devices on which the object relational map verification system operates may include one or more central processing units, memory, input devices (e.g., keyboard and pointing devices), output devices (e.g., display devices), storage devices (e.g., disk drives), and network devices (e.g., network interfaces). The memory and storage devices are computer-readable media that may store instructions that implement the object relational map verification system. In addition, the data structures and message structures may be stored or transmitted via a data transmission medium, such as a signal on a communications link. Various communications links may be employed, such as the Internet, a local area network, a wide area network, or a point-to-point dial-up connection.

The object relational map verification system may use various computing systems or devices, including personal computers, server computers, hand-held or laptop devices, multiprocessor systems, microprocessor-based systems, programmable consumer electronics, electronic game consoles, network PCs, minicomputers, mainframe computers, distributed computing environments that include any of the above systems or devices, and the like. The object relational map verification system may also provide its services to various computing systems, such as personal computers, cell phones, personal digital assistants, consumer electronics, home automation devices, and so on.

The object relational map verification system may be described in the general context of computer-executable instructions, such as program modules, executed by one or more computers or other devices. Generally, program modules include routines, programs, objects, components, data structures, and so on that perform particular tasks or implement particular abstract data types. Typically, the functionality of the program modules may be combined or distributed as desired in various embodiments.

FIG. 2 is a flow diagram illustrating a verify routine invoked by the object relational map verification system in some embodiments. The object relational map verification system can invoke the verify routine 200 when a software developer commands the object relational map verification system to verify an object relational map. The routine begins at block 202. At block 204, the routine receives source code for an object. In some embodiments, the routine receives object code for the object. At block 206, the routine receives a database schema for a database into which the object stores data or from which the object retrieves data. In some embodiments, the routine receives an indication of a database and determines the schema, such as by evaluating the configuration of the database. At block 208, the routine receives database views corresponding to the retrieval and storage operations that are made to retrieve or store data relating to the object. The routine may receive these update and query views by analyzing an object relational map corresponding to the object. At block 210, the routine produces first order logic formulae corresponding to the object relational map. In various embodiments, the routine may produce the first order logic formulae or may receive the first order logic formulae from a software developer. At block 212, the routine can invoke a theorem prover to prove the first order logic formulae. In some embodiments, the routine can employ conventional theorem provers. At decision block 214, the routine determines whether the theorem prover timed out. As an example, if the theorem prover is taking an excessive amount of time, the object relational map verification system may determine that the theorem prover cannot verify the object relational map. Alternatively, the theorem prover may return an error indicating that it could not verify the object relational map. When this is the case, the routine continues at block 218. Otherwise, the routine continues at block 216. At block 216, the routine invokes a generate_model subroutine to identify
counterexamples that demonstrate that the object relational map is incorrect. The generate_model subroutine is described in further detail below in relation to FIG. 3. The routine then continues at block 220, where it returns. At block 218, the routine reports an error indicating that the object relational map could not be verified and then continues at block 220. FIG. 3 is a flow diagram illustrating a generate_model routine invoked by the object relational map verification system in some embodiments. The object relational map verification system can invoke the generate_model routine 300 to generate a model containing a counterexample that demonstrates that an object relational map is invalid. The routine begins at block 302. At block 304, the routine receives a first order logic formula. This first order logic formula can be equivalent to the first order logic formula that the verify routine described above in relation to FIG. 2 received or produced at block 210. At block 306, the routine translates the first order logic formula into a satisfiability problem instance. The object relational map verification system translates a first order logic formula into a satisfiability problem instance by translating each predicate of the first order logic formula into a set of k rows. Each row is a set of tuples identifying a relation between objects and database portions and contains a Boolean variable indicating whether that row is valid, and a set of variables can be evaluated to determine validity. There may also be a set of constants relating to each row. The object relational map verification system can encode a first order logic formula into a quantified Boolean formula and then convert the quantified Boolean formula into a satisfiability problem instance. The object relational map verification system can employ binary decision diagrams to compute quantifications. At block 308, the routine searches for a model. The routine may employ a conventional satisfiability problem solver to identify a model. At decision block 310, the routine determines whether a model could be found. If no model could be found, the routine continues at block 312. Otherwise, the routine provides the model at block 314 and returns. At block 312, the routine revises the maximum bound on the number of tuples or the number of constants and returns to block 308.

Thus, the object relational map verification system can verify object relational models and create models containing counterexamples when an object relational model cannot be verified.

Although the embodiments illustrate mapping objects to databases, one skilled in the art will recognize that the described techniques apply equally to translations between any two different types of data. As examples, the technique can be applied to verify correct database usage after a database migration, conversion of data between two different network protocols, and so forth. In some embodiments, the object relational map verification system can receive first order logic formulas from software developers.

Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims. Accordingly, the invention is not limited except as by the appended claims.

We claim:

1. A method performed by a computer system for verifying an object relational map for software, comprising:
   receiving by the computer system an object employed by the software;
   receiving a database schema, the database schema specifying portions of a database into which an attribute of the received object will be stored or from which the attribute of the received object will be retrieved;
   receiving a query view that specifies in a language what is retrieved from the database;
   receiving an update view that specifies in the query language what is stored in the database;
   generating a first order logic formula corresponding to the received object, database schema, query view, and update view, the first order logic formula expressing the object relational map formulaically; and
   proving a theorem indicated by the generated first order logic formula by invoking a first order logic formula theorem prover, wherein when the theorem is proved, the object relational map is verified and generating a model when the theorem could not be proved.

2. The method of claim 1 wherein generating the model further comprises:
   translating the generated first order logic formula into a Boolean satisfiability problem instance; and
   searching for the model.

3. The method of claim 2 wherein the searching is performed by a Boolean satisfiability problem solver.

4. The method of claim 2 wherein the translating is parameterized by a maximum bound on a number of tuples in a set of relations identified by the first order logic formula.

5. The method of claim 2 wherein the translating is parameterized by a number of constants in a set of constants identified by the first order logic formula.

6. The method of claim 2 wherein the translating further comprises encoding the first order logic formula into a quantified Boolean formula and converting the quantified Boolean formula into the satisfiability problem instance.

7. The method of claim 6 wherein the encoding further comprises representing quantifications in a binary decision diagram.

8. The method of claim 6 wherein the encoding further comprises representing quantifications in a binary decision diagram and converting the binary decision diagram into a set of conjunctive normal form clauses and providing the conjunctive normal form clauses to a satisfiability problem solver.

9. The method of claim 2 wherein when a model cannot be found, increasing a number of tuples, increasing a number of constants, or increasing both and then again searching for the model.

10. A system for verifying an object relational map for software, comprising:
   a memory and processor;
   a database having a database schema that stores data employed by the software;
   an object employed by the software for accessing the database;
   and
   an analyzer that receives a query view that specifies in a query language what is retrieved from the database and an update view that specifies in a query language what is stored in the database, and generates a first order logic formula corresponding to the object, database schema, query view, and update view wherein the first order logic formula expresses the object relational map formulaically; and
   a theorem prover that proves the generated first order logic formula.

11. The system of claim 10 further comprising a model generator that generates a counterexample.
12. The system of claim 10 further comprising a satisfiability problem solver that solves a satisfiability problem corresponding to the generated first order logic formula.

13. The system of claim 10 further comprising:
   a theorem prover that proves the generated first order logic formula;
   a model generator that generates a counterexample based on the generated first order logic formula; and
   a satisfiability problem solver that solves a satisfiability problem corresponding to the generated first order logic formula to generate a model identifying the counterexample.

14. A computer-readable medium storing computer-executable instructions that, when executed, cause a computer system to perform a method for verifying an object relational map for software, the method comprising:
   receiving source code defining an object employed by the software;
   determining a database schema, the database schema specifying portions of a database into which an attribute of the received object will be stored or from which the attribute of the received object will be retrieved;
   receiving a query view that specifies in a query language what is retrieved from the database and an update view that specifies in the query language what is stored in the database; and
   generating a first order logic formula corresponding to the received object, database schema, query view, and update view that is employed by a first order logic theorem prover to prove a theorem, the first order logic formula expressing the object relational map formally, wherein when the theorem is proved, the object relational map is verified.

15. The computer-readable medium of claim 14 wherein the method further comprises:
   translating the generated first order logic formula into a Boolean satisfiability problem instance; and
   searching by a Boolean satisfiability problem solver for a model that satisfies the satisfiability problem instance.

16. The computer-readable medium of claim 15 wherein the method further comprises identifying a counterexample.

17. The computer-readable medium of claim 15 wherein the method further comprises identifying a counterexample based on the model.

* * * * *