Formalizing ORM Models using Alloy

Amir Jahangard Rafaanjani, Seyed-Hassan Mirian-Hosseinzabadi
Sharif University of Technology, Tehran, Iran
jahangard@ce.sharif.edu, hmirian@sharif.edu

Abstract
ORM (Object Role Modeling) is a rich and popular conceptual modeling method. Automated reasoning like satisfiability testing allows developers to detect modeling mistakes in the early stages of development. In this report we propose a lightweight formalization of ORM meta-model in Alloy. Using this meta-model as a toolkit one can easily specify ORM models in Alloy and verify various properties on them using Alloy analyzer.

1 Introduction
In this section a brief introduction to Alloy modeling language, ORM and satisfiability of ORM models is provided.

1.1 Alloy
Alloy (1) is a structural modeling language based on the first-order logic, suitable for expressing complex structural relationships and constraints. The syntax of Alloy is similar to the standard mathematical syntax of the first-order logic. The essential constructs of Alloy are as follows:

- Signatures denote sets of entity objects. They introduce basic types and collections of relations (called fields) along with the types of the fields and constraints. A signature may inherit fields and constraints from another signature.
- Facts are constraints on relations and objects that are assumed always to hold. A fact is a formula that takes no arguments and does not need to be invoked explicitly. You can give a fact a unique mnemonic name.
- Predicates are templates for a parameterized constraint. They can be applied elsewhere by instantiating the parameters and evaluate to either true or false.
- Functions are parameterized formulas intended to be used elsewhere. A function can be applied elsewhere by instantiating the parameters and evaluates to a value.
- Assertions are constraints that are intended to follow from the facts of a model. An assertion is a formula whose correctness needs to be checked, assuming the facts in the model.

Alloy is equipped with the Alloy Analyzer. It relies on recent advances in SAT (boolean satisfiability) technology. The Alloy Analyzer translates constraints to be solved from Alloy into
boolean constraints, which are fed to an off-the-shelf SAT solver. As solvers get faster, so Alloy’s analysis gets faster and scales to larger problems. Using the best solvers of today, the analyzer can examine spaces that are several hundred bits wide (that is, of \(10^{60}\) cases or more). Alloy Analyzer provides two kinds of automatic analysis: simulation in which the consistency of a fact or predicate is demonstrated by generating a snapshot of the model; and checking, in which a consequence of the specification is tested by attempting to generate a counterexample for an assertion.

### 1.2 ORM

ORM (2) is a conceptual modeling approach that views the world in terms of objects, and the roles they play. ORM allows a variety of data constraints to be defined such as mandatory role, subset, uniqueness, exclusion, cardinality and ring constraints.

Figure 1 shows an ORM diagram that models a ‘Conference Paper’ universe of discourse. This sample is taken from (3) and changed to ORM2 (4) notation.

In this diagram, object types are represented as named soft rectangles and relationship-types as named sequences of adjacent role boxes. Individual role names are written in square brackets near each role. A bar over a role or role sequence indicates an internal uniqueness constraint, and a circled underline or circled double underline denotes an external uniqueness (or primary uniqueness) constraint. Value constraints are represented as a braced list of values, and frequency constraints as a numeric range attached to one or more roles. Role and relationship subset constraints are shown as arrow with a circled subset notation, exclusion constraints as a circled ‘\(x\)’ between the relevant role-sequences, and subtype links as solid arrows between object types.
2 ORM meta-model in Alloy

In this section we introduce Alloy representation of the ORM meta-model. We have modeled structure and semantics of ORM into different Alloy modules.

2.1 ORM Structure

We introduce various signatures to represent ORM main concepts like: types, object types, predicates, roles and population as objects. Here are the signatures in Alloy.

2.1.1 Type

We use a generic type signature that object and predicates will extend it, and is used as a generalization for these two signatures.

abstract sig Type{}

2.1.2 Object Types

The signature OT which represents Object types is defined as follows:

abstract sig OT extends Type{
    subtypes: set OT,
    otInstances: set Obj,
    otRoles: set RO,
    otValues: set Obj
}

The keyword abstract indicates that OT will be refined further by signatures that extend it. In the declaration parts various fields are defined for an object type. If the object type has subtypes the set of its subtypes is shown by subtypes. Population of it is shown by otInstances field which defines a relation between object type and its population as a set of objects. Roles played
by the object type are specified by \textit{roles}. And finally if there is some value constraint on population of the object type, \textit{OTValues} field will be used.

Based on the definition of the OT signature, some basic facts can be stated about it. Here we just define syntactical facts and semantic facts will be stated later in this section.

\begin{verbatim}
fact {
  1) roles=~player
  2) all o1,o2:OT | no o1.otValues \& o2.otValues or o1 in SameTypes[o2]
  3) all o1,o2:OT | o1 in SameTypes[o2] => (o1.otValues in o2.otValues or o2.otValues in o1.otValues)
}
\end{verbatim}

The first fact states that if an object type \( A \) plays a role \( R \) then the player of \( R \) is \( A \) (\(~r\) indicates transpose of relation \( r \)). Fact (2) states that there should not be any intersection between two object types value constraints unless they are of the same type (\textit{SameTypes} function will be defined later). Fact (3) states that if two object types are of the same type, one of their value constraints should be subset of the other.

2.1.3 Predicates (Relationship Types)

The predicates are presented using RT signature which is defined as follows:

\begin{verbatim}
abstract sig RT extends Type{
  roles: some RO,
  rInstances: disj set Row,
  unique: set RO,
  multiRoleSubset: roles ->lone RO,
  RTSuperset: set RT,
  RTEqual: set RT,
  RTDisjoint: set RT,
  Objectified: lone ET,
  relates: Obj->Obj
}
\end{verbatim}

Each RT has one or more roles and can be unary, binary or \( n \)-ary. These roles are shown by \textit{roles} field. Instances of a predicate are stated by a set of \textit{Rows} which will be defined later. \textit{multiRoleSubset} is a relation that is used to define subset relationships between the roles of current fact and another fact. Subset, equality and disjointness relations between predicates are shown by \textit{RTSuperset}, \textit{RTEqual} and \textit{RTDisjoint} fields respectively. The \textit{Objectified} field is used for objectification of predicates. The field \textit{relates} shows the objects that are related to each other in the predicate.

Some structural facts about predicates can be stated as follows:

\begin{verbatim}
fact {
  1) rel=~roles
  2) all rt1:RT, rt2:rt1.RTSuperset | rt2!=rt1 \&\& #rt2.roles=#rt1.roles
  3) all rt1:RT, rt2:rt1.RTEqual | rt2!=rt1 \&\& #rt2.roles=#rt1.roles
  4) all rt1:RT, rt2:rt1.RTDisjoint | rt2!=rt1 \&\& #rt2.roles=#rt1.roles
  5) ~RTEqual in RTEqual
  6) ~RTDisjoint in RTDisjoint
}\end{verbatim}
Fact (1) states that to relations rel and roles are transpose of each other, i.e. if role r is in the roles set of predicate rt then the predicate of r is rt. Facts 2, 3 and 4 state that if there is a subset, equality or disjoint relation between two predicate then they should have the same number of roles. Facts (5, 6) state that equality and disjointness relations are symmetric. Two predicates cannot be subset and equal to each other simultaneously (fact 7).

In order to model a predicate population we introduced a signature called Row which maps roles of the predicates to the role player instances:

```
sig Row{
  rt: one RT,
  row: RO -> lone Obj
  {row.univ = rt.roles}
}
```

Each row is related to a predicate that contains the row and has a relation that maps each role of the predicate two a single object instance. The domain of the row relation is equal to the roles of its relationship.

### 2.1.4 Roles

Roles in the ORM model are defined by RO signature which is defined here:

```
abstract sig RO{
  rel: one RT,
  player: one OT,
  roInstances: set Obj,
  RoleValues: set Obj,
  FixFrequency: lone Int,
  MinFrequency: lone Int,
  MaxFrequency: lone Int,
  superSet: set RO,
  equalSet: set RO,
  excludeSet: set RO,
  order: Int,
}
```

Each role is in one predicate and is played by an object type. Its population is modeled using a set of objects. If there is a role value constraint on population of the role it is shown by RoleValues field. Frequency constraints on role are defined using FixFrequency, MinFrequency and MaxFrequency fields. Inheritance, equality and exclusion relation between roles are defined using superSet, equalSet and excludeSet fields respectively. Order field is used to define ordering (reading order) on roles of a predicate. Here are some structural facts about roles:

```
fact {
  1) all r:RO, m: r.MinFrequency, n: r.MaxFrequency | m>0 && n>1 && n>=m
  2) ~excludeSet in excludeSet
  3) all r:RO, s: r.excludeSet | SameType[r.player, s.player]
  4) no equalSet & superset
  5) no disj r1, r2: RO | r1.rel=r2.rel && r1.order=r2.order
}
The first fact states some constraints on values of minimum and maximum frequency constraints on roles. Fact (2) states that exclusion relation between two roles is symmetric and fact (3) states that when there is an exclusion relation between two roles their players should be of same type. There should be no intersection between equal roles and super-roles of a role. For each two disjoint roles in a predicate their order should be different and order of a role should be between 1 and number of roles in its predicate (facts 5 and 6).

2.2 ORM Semantics

In this section we formalize semantics of various structural concepts which we introduce in previous section. These will include semantics of various constraints, such as, uniqueness, mandatory, cardinality, subset, equality, exclusive and ring constraints.

2.2.1 Subtypes

If instances of an object type are classified into a more specific type, this specialized type is known as a subtype. The main reason for using a subtype in modeling is to declare typing constraints. Another reason for subtyping is to encourage reuse of model components.

Subtypes in ORM are proper subtypes. For example, we say that $B$ is a proper subtype of $A$ if and only if the population of $B$ is always a subset of the population of $A$, and $A \neq B$.

As we saw earlier in OT signature, subtypes of an object type are shown using subtypes field which is the set of direct subtypes of the object type. A simple fact is used to constraint the population of a subtype:

\[
\text{fact}\{\text{all } x, y:OT | x \in y.subtypes \Rightarrow x.otInstances \in y.otInstances}\}
\]

A predicate is used to determine whether to object types are direct or indirect subtype of each other; if so we say that those are of same type.

\[
\text{pred}\ \text{SameType}[e,f:OT]\
\quad \text{f in } (e+e.*\text{subtypes} + *\text{subtypes.e} + (x:OT| \text{some } *\text{subtypes.x} \& *\text{subtypes.e}))
\]

In ORM the population of two object types is disjoint unless they are of the same type, i.e. one of them is subtype of the other one. The following fact states this:

\[
\text{fact}\{\text{all } x, y:OT | \text{some } x.otInstances \& y.otInstances \Rightarrow \text{SameType}[x, y]\}
\]

Various constraints can be defined on subtypes. These constraints are total, exclusive and mandatory constraints.
- **Total Subtypes**: The total constraint between subtypes means that the population of the supertype is exactly the union of the population of these subtypes. The following predicate is used to specify total subtype constraint in Alloy:

\[
\text{pred TotalSubtype}(x:OT, y:\text{set} OT)\{ \\
y \in x.\text{subtypes} \&\& \forall \text{obj}:x.\text{otInstances} \mid (\exists \text{ot}:y \mid \text{obj} \in \text{ot.}\text{otInstances})
\}
\]

The predicate says that for each object instance in supertype there exists an identical object instance in at least one of the subtypes.

- **Disjoint (Exclusive) Subtypes**: The exclusive constraint between subtypes means the populations of these subtypes are pairwise distinct, i.e. the intersection of the populations of each pair of the subtypes must be empty. This is specified by the following predicate:

\[
\text{pred DisjointSubtype}(x:OT, y:\text{set} OT)\{ \\
y \in x.\text{subtypes} \&\& \#y\geq2 \forall o1,o2:y \mid \text{no } o1.\text{otInstances}\&o2.\text{otInstances}
\}
\]

- **Partition (Mandatory) Subtypes**: If both total and exclusive constraints between subtypes exist it is called partition.

\[
\text{pred Partition}(x:OT, y:\text{set} OT)\{ \\
\text{DisjointSubtype}[x, y] \&\& \text{TotalSubtype}[x, y]
\}
\]

### 2.2.2 Roles

Now we introduce some semantics about roles and their population as facts in Alloy

\[
\text{fact} \{ \\
1) \forall r:RO \mid r.\text{roInstances} \in r.\text{player.}\text{otInstances} \\
2) \forall r:RO \mid r.\text{roInstances} = \text{univ.}(r<:r.\text{rel.}\text{rtInstances}\.\text{row})
\}
\]

Fact (1) indicates that instances of a role are a subset of instances of its player and fact (2) states that instances of a role are a subset of its predicate population projection over that role.

### 2.2.3 Frequency Constraints

A fixed frequency constraint \( f \) on a role states that each instance which plays the role, does so exactly \( f \) times.

\[
\text{fact} \{ \forall r:RO,n:r.\text{FixFrequency} | n>0 \Rightarrow \forall o:r.\text{roInstances} | \\
\#\{rw:r.\text{rel.}\text{rtInstances} | r.(rw\.row)=o\}=n
\}
\]

A role frequency constraint can be stated using lower or upper bounds or both. The following facts are used to state these constraints:
A frequency constraint may cover more than a role in a predicate. In this case a signature is used to specify multi-role frequency constraints:

```
abstract sig FC extends Constraint{
   fcRT:RT, fcRoles: set RO, fcMin: one Int, fcMax: lone Int
   | {fcRoles in fcRT.roles}
}
```

For each constraint the set of roles, the predicate and minimum and maximum frequency are stated using the above signature fields. The inline fact states that the roles should share a common predicate. This constraint involves the frequency of the projection of the predicate population over the constraint roles. For example if there is a constraint with minimum 3 and maximum 4 on two roles \( r \) and \( s \), in the population of this relationship, \( r \) and \( s \) must occur together (i.e. as a tuple) at least 3 times and at most 4 times. The following fact specifies this constraint:

```
fact{
   all fc:FC | all rw:fc.fcRT.rtInstances| let
      freq=\#\{rw2:fc.fcRT.rtInstances|fc.fcRoles.(rw.row)=fc.fcRoles.(rw2.row)\}|
      (one fc.fcMax => fc.fcMax>=freq) && freq>=fc.fcMin
}
```

### 2.2.4 Value Constraints
A value constraint in ORM indicates the possible instances for an object type or a role. A value constraint is denoted as a set of objects:

```
fact{all o:OT | some o.otValues => o.otInstances in o.otValues}
fact{all r:RO | some r.RoleValues => r.roInstances in r.RoleValues}
```

The above fact states that if there is a value constraint on an object type or a role then it’s population should be a subset of the value constraint.

### 2.2.5 Subset, Equality and Exclusive Constraints
The subset constraints can be defined on roles. A subset constraint between two roles states that population of one is the subset of the other. Equality constraints between roles are defined similarly by equalSet relation. Exclusion constraints between two or more roles state that populations of them are disjoint. The semantics of these constraints are formalized using the following facts:

```
fact{all r:RO, s:r.superSet | r.roInstances in s.roInstances}
fact{all r:RO, s:r.equalSet | r.roInstances = s.roInstances}
fact{all r:RO, s:r.excludeSet | no r.roInstances & s.roInstances}
```
2.2.6 Mandatory Constraints
A role is called mandatory if its population is always identical to its player population, i.e. each instance in the role player population must play that role. This is stated using a simple fact in Alloy:

\[
\text{fact } \{ \text{all } r:\text{MandatoryRole} \mid r.\text{player}.\text{otInstances} = r.\text{roInstances} \}
\]

2.2.7 Disjunctive Mandatory Constraint
The disjunctive mandatory constraint is used to constrain a set of two or more roles connected to the same object type. It means that each instance of the object type population must play at least one of the constrained roles. We use an abstract signature \text{DMC} to specify disjunctive mandatory constraints. Each DMC constraint consists of an object type and a set of roles on which the constraint is being defined. The player of all these roles should be the object specified.

\[
\text{abstract sig DMC extends Constraint} \{
\text{dmcOT:OT,}
\text{dmcRoles:set RO}
\}\{\text{all } r:\text{dmcRoles}|r.\text{player} = \text{dmcOT}\}
\]

We use the following fact in order to specify the constraint:

\[
\text{fact } \{ \text{all } dmc:\text{DMC} \mid \text{all } oins:dmc.dmcOT.\text{otInstances} \mid
\text{some } r:dmc.dmcRoles | oins \in r.\text{roInstances} \}
\]

2.2.8 Uniqueness Constraints
Two types of uniqueness constraints are possible in ORM: internal and external. Internal uniqueness constraints are defined on instances of roles in a single predicate. We consider two types of internal uniqueness constraints: \text{single role} and \text{multi-role}. Single-role constraints are uniqueness constraints on a role in a predicate but multi-role constraints span more than one role in a predicate. Here is the fact defining single-role uniqueness constraints:

\[
\text{fact } \{ \text{all } r:\text{UniqueRole} \mid \#r.\text{roInstances} = \#r.\text{rel}.\text{rtInstances} \}
\]

In order to define multi-role uniqueness constraints a new signature called \text{UC} is defined. This signature has a \text{uRoles} field which indicates the set of unique roles.

\[
\text{abstract sig UC extends Constraint} \{
\text{uRoles: set RO} \} (\#\text{uRoles}>1)
\]

Having defined this signature we can use a fact to specify multi-role uniqueness constraint on a predicate:

\[
\text{fact } \{ \text{all } r:\text{RT}, uc:r.\text{ucs} \mid
\text{all disjoint } r1, r2:r.\text{rtInstances} \mid
\text{uc.uRoles} \subsetneq r1.\text{row} \neq \text{uc.uRoles} \subsetneq r2.\text{row} \}
\]
2.2.9 Ring Constraints

In ORM ring constraints can be applied to a pair of roles in a predicate that are connected directly or indirectly - via subtypes - to the same object type. Each ring constraint consists of two roles and their relationship. We introduce a general abstract signature for all ring constraints here. All ring constraints are inherited from this general signature definition. The fact type of the following signature states that roles participating in a ring constraint should be in a predicate and their players should be of same type:

```
abstract sig RC extends Constraint{
    disj rcRole1,rcRole2:RO,
    rcRT:RT
}{ (rcRole1+rcRole2 in rcRT.roles) && SameType[rcRole1.player,rcRole2.player] }
```

In order to define ring constraint we introduce a helper predicate called `InSameRow` which gets true when two object instances are in a row in accordance with two roles.

```
pred InSameRow(r,s:RO,x,y:Obj){
    (some rw:r.rel.rtInstances | r.(rw.row)=x && s.(rw.row)=y)
}
```

Six types of ring constraints are supported by ORM:

- **Irreflexive Ring Constraint (irRC):** This constraint states that an object instance cannot participate in a relationship with itself. For example no one can be the "Parent of" himself, so the relation "parent of" is irreflexive.

  ```
  abstract sig irRC extends RC{
  }{ no o1:Obj|InSameRow[rcRole1,rcRole2,o1,o1] }
  ```

- **Symmetric Ring Constraint (symRC):** This constraint states that if a relation holds in one direction it should also hold on the other. In case of ring constraint on role pair (x, y) in a predicate, it states that if there is a row in the predicate population that with o1 as object instance of x and o2 as object instance of y then there should be another row with o2 as object instance of x and o1 as object instance of y.

  ```
  abstract sig symRC extends RC{
  }{ all o1,o2:Obj | InSameRow[rcRole1,rcRole2,o1,o2] => InSameRow[rcRole1,rcRole2,o2,o1] }
  ```

- **Asymmetric Ring Constraint (asRC):** Asymmetric is the opposite of symmetric. If a relation holds in one direction it may not hold in on the other. In case of ring constraint on role pair (x, y) in a predicate, it states that if there is a row in the predicate population that with o1 as object instance of x and o2 as object instance of y then there cannot be another row with o2 as object instance of x and o1 as object instance of y.

  ```
  abstract sig asRC extends RC{
  }{ all o1,o2:Obj | InSameRow[rcRole1,rcRole2,o1,o2] =>
  ```
• **Antisymmetric Ring Constraints (ans):** Antisymmetric is also the opposite of symmetric, but all asymmetric relations should be irreflexive which is not the case for antisymmetric relations. In case of ring constraint on role pair \((x, y)\) in a predicate, it states that if there is a row in the predicate population that with \(o_1\) as object instance of \(x\) and \(o_2\) as object instance of \(y\) then there cannot be another row with \(o_2\) as object instance of \(x\) and \(o_1\) as object instance of \(y\) unless \(o_1 = o_2\).

```alloy
abstract sig ansRC extends RC{}
{all disj o1,o2:Obj | InSameRow[rcRole1,rcRole2,o1,o2] => !InSameRow[rcRole1,rcRole2,o2,o1]}
```

Note that the only difference between \(asRC\) definition and \(ansRC\) definition is that in \(ansRC\) two object instances are disjoint.

• **Intransitive Ring Constraint:** A relation is intransitive if, whenever it relates some \(A\) to some \(B\), and that \(B\) to some \(C\), it cannot relate that \(A\) to that \(C\). In case of ring constraint on role pair \((x, y)\) in a predicate, it states that if there is a row in the predicate population that with \(o_1\) as object instance of \(x\) and \(o_2\) as object instance of \(y\) and another row with \(o_2\) as object instance of \(x\) and \(o_3\) as object instance of \(y\) then there cannot be another row with \(o_1\) as object instance of \(x\) and \(o_3\) as object instance of \(y\).

```alloy
abstract sig itRC extends RC{}
{all o1,o2,o3:Obj | (o1!=o2 && o2!=o3) =>
 (InSameRow[rcRole1,rcRole2,o1,o2] &&
 InSameRow[rcRole1,rcRole2,o2,o3] ) =>
 not InSameRow[rcRole1,rcRole2,o1,o3] )}
```

• **Acyclic Ring Constraints (ac):** The acyclic constraint is a special case of the irreflexive constraint. It states that a relation cannot directly or indirectly (through a chain) relates an object with itself. In ORM this constraint is preserved as difficult constraint. “Because of their recursive nature, acyclic constraints maybe expensive or even impossible to enforce in some database systems.” (2). Here we use closure on the relate field of predicate signature definition to define this constraint:

```alloy
abstract sig acRC extends RC{}
{all o1:rcRole1.player.otInstances | o1 not in o1.^+(rcRT.relates)}
```

### 3 Case Studies
In this section we apply our approach to various ORM models of different size and complexity. Here is the Alloy model of the ORM diagram presented in Figure 1.
open ORM_Model

one sig Person, Paper, Accepted, Status, Review, Rating, Institution, Country extends ET{}
one sig Email, Phone, PName, Title, PageNr, IName extends VT{}

one sig Refereed, Authors, Presents, Attends, HasEmail, HasPhone, HasPName, HasTitle, HasStatus, HasPageNr, HasRating, IsFrom, HasIName, IsBasedIn extends RT{}

one sig c1, c2 extends EUC{}

one sig acc, rej, und in Obj{}

fact {
player = PPap->Person + PapP->Paper + PrPap->Person + PaprP->Paper +
PAP->Person + APP->Accepted + PA->Person + EAP->Email + PEA->Person +
PNP->Phone + PName->Person + PNP->PageNr + PT->Paper +
TP->Title + SP->Status + PS->Paper + APnr->Accepted + NrAP->PageNr +
RRev->Rating + RevR->Review + PI->Person + IP->Person +
IIN->Institution + INI->IName + IC->Institution + CI->Country

roles = Refereed->{PrPap+ PapP} + Authors->{PPap+ PapP} +
Presents->{PAP+APP} + Attends->PA + HasEmail->{PEA+EAP} +
HasPhone->{PPN+PNP} + HasPName->{PPN+PNP} + HasTitle->{PT+TP} +
HasStatus->{SP+PS} + HasPageNr->{APnr+NrAP} + HasRating->{RRev+RevR} +
IsFrom->{PI+IP} + HasIName->{IIN+INI} + IsBasedIn->{CI+IC}

MandatoryRole = {PapP+PEA+PPN+PT+PS+APnr+PI+PPN+PNP+IIN}
UniqueRole = {EAP+PEA+PPN+PNP+PT+PS+APnr+RevR+PI+IIN+INI}
Refereed->Authors in RTDisjoint
RTSubset = Authors->Presents
FixFrequency = PapP->2
Subtypes = {Paper->Accepted}
superSet = PAP->PA
Objectified = Refereed->Review
otValues=Status->{acc+rej+und}
c1.eucr1=IP& c1.eucr2=PNP& c1.ucr1=PI& c1.ucr2=PPN& c1.eucOT=Person

c2.eucr1=CI& c2.eucr2=INI& c2.ucr1=IC& c2.ucr2=IIN& c2.eucOT=Institution
}

Analyzing this model in Alloy Analyzer, i.e. creating a satisfying model of the schema took only 3.4 seconds on a 2.7 GHz Intel Core i7(64-Bit) machine. But creating a valid instance having population for all concepts or roles took long times around one hour. Our main focus of on future work is to make our approach more scalable.

References


