Safety Checking for Domain Relational Calculus Queries Using Alloy Analyzer

Abhabongse Janthong

Abstract

A domain relational calculus (DRC) query is a database query which uses the mathematical set notation to enumerate the result based on the data in the database. A DRC query is safe if and only if it is domain-independent, i.e., the result of the query is determined solely by the data in the database, not the domain of data values. In this project, we provide a framework for verifying whether a given DRC query is safe by translating the problem into a verification task in Alloy language. We also experiment with our translation framework using various examples of safe and unsafe queries.

Contents

1 Introduction

1.1 Basic definition of database system

In database systems, a **database** is a collection of tables, and each **table** stores one coherent dataset. Each row of a table represents a single data point and each column represents an attribute of the data. Theoretically speaking, a table is merely a mathematical **relation** over one or more sets of scalar values (numbers, strings, etc.) and each row of the table is a **tuple** (or an element) in such relation.

For the sake of keeping our explanation simple, the relationship of data across different tables can be expressed by declaring **keys** (usually referred as IDs) as explicit columns in the table. Each row of a table may have a **primary key** declared as an anchor point to which other tables can refer to with **foreign keys**. In practice, since integers can be used as keys, it is sufficient to portray keys no different than other scalar values. Therefore, we omit the keys from our discussion for the rest of this project.

First example of database

For example, let us say that we would like to construct a database to store information of four students: Alice, Bob, Carol, and David. The database must contain their personal data (e.g. year of birth) as well as a network graph representing their friendship relation. One possibility is that we create two tables, described below.

- **PersonalData**(Name, BirthYear): a two-column table containing each student's name and year of birth;
- **Friendship**(NameA, NameB): a (possibly non-symmetric) two-column table where each row contains names and only names of two students who are friends of each other.

An example of an instance of such database are shown in Table 1 and Table 2, and the visualization is shown in Figure 1.

Name	BirthYear
'Alice'	1994
'Bob'	1995
'Carol'	1994
'David'	1993

Table 1. Table **PersonalData** with four students: Alice, Bob, Carol, and David, and their corresponding year of birth.

NameA	NameB
'Alice'	'Bob'
'Bob'	'Carol'
'Alice'	'Carol'
'Carol'	'David'

Table 2. Table **Friendship** showing that Alice, Bob, and Carol know each other, but David knows only Carol.

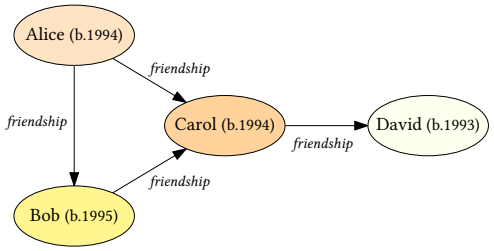

Figure 1. The visualization of data in the table **PersonalData** and **Friendship**.

1.2 Database query model

One of the most basic database function is the **query** function, i.e., the process of fetching the stored data from the database. SQL model is an example of one of the most practical query model adopted in industry. Nonetheless, it is also useful to discuss theoretical query models (such as **relational calculus** and **relational algebra**) in order to gain a better understanding of database query models.

Definition of Domain Relational Calculus

In this report, we mainly discuss one particular variant of relational calculus query model, namely the **domain relational calculus** (DRC). In this model,

- We use set comprehension notation (i.e., set defined by a predicate) to construct a query result. Such predicate must be in first-order logic.
- Each identifier in the set comprehension must represent a scalar value in the database (as opposed to representing a tuple in tuple relational calculus).

Table names as predicates

As part of the predicate in set comprehension, the following notation regarding the existence of a tuple in a database table is allowed.

Notation. Suppose that R is a database table with m columns. Let $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_m \rangle$ be an m-tuple, where each x_i represents a scalar value. Then $R(\boldsymbol{x})$ denotes a predicate whose value is true *if and only if* the table R contains the tuple $\boldsymbol{x}$. Sometimes we also write $R(x_1, x_2, \ldots, x_m)$ directly instead. ◇

For example, based on the data from Table 1, we could say that **PersonalData**(‘Alice’, 1994) is true whereas **PersonalData**(‘Bob’, 1993) is false.

Examples of DRC queries

Here are some examples of DRC queries.

Example 1. Suppose that we want to obtain all students and their year of birth who were born *strictly* before 1995 (using the database table **PersonalData**(Name, BirthYear) as described in Table 1). We use the following DRC query.

$$Q_{\text{before 1995}} = \{name, year \mid \textbf{PersonalData}(name, year) \wedge (year < 1995)\}$$

This query returns a set of three tuples: {(‘Alice’, 1994), (‘Carol’, 1994), (‘David’, 1993)}.

Example 2. Suppose that we want to obtain all friends of Bob (based on the database table **Friendship**(NameA, NameB) from Table 2). We use the following DRC query.

$$Q_{\text{Bob's friend}} = \{name \mid \textbf{Friendship}(name, \text{‘Bob’}) \vee \textbf{Friendship}(\text{‘Bob’}, name)\}$$

This query returns a set of two elements: {‘Alice’, ‘Carol’}. *Note that we sometimes omit the tuple notation when it contains only a single column.*

Example 3. Suppose that we want to obtain all pairs of students who share a common friend using the same data as above. We use the following query.

$$Q_{\text{friend of friend}} = \{x, y \mid (x < y) \land \exists z[(\textbf{Friendship}(x, z) \lor \textbf{Friendship}(z, x)) \land (\textbf{Friendship}(y, z) \lor \textbf{Friendship}(z, y))]\}$$

This query would return {('Alice', 'Bob'), ('Alice', 'Carol'), ('Bob', 'Carol'), ('Alice', 'David'), ('Bob', 'David')}. *This query utilizes the existential quantification in first-order logic to represent a friend in common between two students. Also assume that* < *does a lexicographical comparison of two strings.*

Notice how the domain for variables (such as *name* and *year*) are not explicit in the query. This is okay because predicates **PersonalData** and **Friendship** provides the *finite bound* of what could be in the result of the query. We discuss this in more depth in the next subsection.

1.3 Safety in DRC queries

Motivated example of unsafe queries

Let us consider the query $Q_{\text{before 1995}}$ from above once again. You may have noticed that, at least in the paradigm of database systems, it would *not* make practical sense to make such query but *without* the predicate **PersonalData**(*name*, *year*). If it was the case, then the result of the query would also have included tuples that are *not* from the table **PersonalData** in order to be consistent with the mathematical definition of set notation.

More concretely, the result of

$$Q^*_{\text{before 1995}} = \{name, year \mid year < 1995\}$$

would change depending on what is defined as the scope or the domain of the scalar values. For example, if all nonempty strings are allowed for student names, and all integers are allowed for year of birth, then ('Eve', -80) would be part of the query result. However, if the domain only allows positive integers for year of birth, then the same tuple would *not* appear in the result.

Definition of safe vs. unsafe query

We call queries like $Q^*_{\text{before 1995}}$ **unsafe** as their result changes when the domain changes (i.e., results are **domain-dependent**). On the other hand, queries such as $Q_{\text{before 1995}}$ and $Q_{\text{Bob's friend}}$ are examples of **safe** queries because the result of the query is always the same no matter what the scope of the domain is. In other words, their results rely only on data in the database. See Chapter 5 of *Foundations of Databases* [AHV95, p. 75] for more information.

The concept of domain-dependency was summarized by Fagin in his paper [Fag82] and it was historically different from the *original* meaning of safety of query formulae as introduced by Ullman [Ull83]. For this project, we adopt a certain assumption [AB88] so we refer to a query as **safe** *interchangeably* with the term **domain-independent**.

More examples of unsafe queries

Now let us consider a few more examples of unsafe queries. Assume that we have a database table **Follows**(fan, idol) representing the fact that fan is following idol on a social network. Consider the following queries in DRC.

Example 4.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{`Alice'})\}$$

The first query, $Q_{\text{not following Alice}}$, returns a set of all people who did not follow Alice. It is unsafe because, for some database instance (such as when no one follows Alice), the result of the query will be every person within the domain. Hence, the result would be domain-dependent.

Example 5.

$$Q_{\text{weird pairing}} = \{x, y \mid \textbf{Follows}(x, \text{`Alice'}) \vee \textbf{Follows}(y, \text{`Bob'})\}$$

The second query, $Q_{\text{weird pairing}}$, returns a set of pairs of people such that the first person follows Alice, or that the second person follows Bob. This query is also not safe. One counterexample is that no one follows Alice but Alice is the only person who follows Bob. Then $(x, \text{`Alice'})$ will appear in result for every x in the domain of people, and thus is domain-dependent.

Example 6.

$$Q_{\text{follows all}} = \{x \mid \forall y[\textbf{Follows}(x, y)]\}$$

The third query $Q_{\text{follows all}}$ returns a set of people who follows everyone in the domain. Notice how the result for this query is guaranteed to be bounded even if the domain was infinite. However, suppose that Alice follows everyone in the domain D_1. If the domain D_2 is defined as $D_2 = D_1 \cup \{\text{`Bob'}\}$ where $\text{`Bob'} \notin D_1$, then Alice would appear in the result when the domain D_1 is used, but would *not* if D_2 is used. Therefore, query $Q_{\text{follows all}}$ is domain-dependent.

1.4 Project goal and previous work

The goal for this project is to study and analyze the safety of a query in DRC model for a specified database schema using Alloy Analyzer (introduced in the next section). Specifically,

> "Given a database schema S and a query Q, does there exist a database instance under two *distinct* domains D_1 and D_2 such that the respective results R_1 and R_2 differ. If so, then the query Q is considered unsafe; otherwise it is safe."

The verification of data models using Alloy Analyzer has been investigated in prior work, including the data models whose specifications are in relational database schemata [CP09] and other specification language such as ORA-SS [WDSG06]. The verification of data models in the context of web applications

has also been studied [NB11, NBB15]. However, none of them involved the modeling of database relational queries. Also note that we traditionally use Codd's Theorem [Cod72] to verify whether a DRC query is safe by providing an equivalent relational algebra query statement, which is outside the scope of this project.

Although relational calculus is not a practical way to make a query to the database, this project should provide an insight into the fundamental concepts of database theory as well as provide us a framework to help us on determining whether a DRC query is safe, a task usually tediously done by humans due to its complexity.

2 Model In Alloy Analyzer

Alloy Analyzer is a tool for modeling objects with specifications regarding their related structure, and formally verifying whether some properties hold for such objects based on some other pre-asserted properties. Alloy has its own specification language as well as integrated development environment (IDE), which includes a visualizer. The tool was developed by Daniel Jackson and his team at the Massachusetts Institute of Technology (MIT). See Alloy manual in his book [Jac12] or on the Alloy website `http://alloy.mit.edu/alloy/documentation.html`.

To solve the DRC query safety check problem, we rephrase the queries in terms of Alloy verification tasks. In this section, we explain the essential components of the model to we need to construct in Alloy specification language.

2.1 Model overview

The main question of this project is that, given a database schema and a DRC query as input, we need to be able to translate the input into a verification task to be solved by Alloy Analyzer. Specifically, our Alloy model would consist of the following components.

- Since we need to show whether a DRC query is domain-dependent, we need to be able to model multiple **domain sets** (see §1.4), each with different **scalar values** in the set.
- The model needs to be able to model **database instances** (i.e., collection of table instances) based from the given database schema, using **scalar values** mentioned above.
- The given DRC query should be translated to a **query function** in Alloy language. The function should output a result given a **domain set** and a **database instance**.
- As per the main goal in §1.4, we need an **Alloy predicate** which would solve for two distinct domains which yield two distinct query results based on a common database instance.
- *Optional.* For visualization, we need a **placeholder for results** in the Alloy model, declared as a model signature.

We discuss the details for each of these components in depth in the subsequent subsections.

```
1  sig Superparticle {} {
2    Superparticle = Universe.Element
3  }
4
5  abstract sig Universe { Element: some Superparticle }
6  one sig UniverseAlpha, UniverseBeta extends Universe {}
7
8  some sig Particle in Superparticle {} {
9    Particle = UniverseAlpha.Element & UniverseBeta.Element
10 }
```

Source Code 1. Alloy model signature for domain sets (*universes*) and their elements (*particles*). This stencil code will always be present in all Alloy models. The fact assert on Line 2 ensures that each `Superparticle` must be present in at least one domain, for the sake of conciseness of models generated by Alloy.

2.2 Domain sets and values

As previously mentioned, we need to be able to consider different domain sets in order to ultimately determine if a query is domain-dependent. In Source Code 1,

- **`Superparticle`**: a set of all possible scalar values across all domains.
- **`Universe`**: a collection of exactly two domains (or alternatively, *universes*): `UniverseAlpha` and `UniverseBeta`. Each domain (i.e., *universe*) has one attribute called `Element`, representing the subset of `Superparticle`s which belong to that domain. We can refer to the first domain set as `UniverseAlpha.Element`, for example.
- **`Particle`**: is the domain set of *allowable* scalar values in the actual model of database instances, which is restricted to the intersection of each of the two domain sets.

2.3 Database instances

Because database instances will heavily depend on the schema, instead of creating a static Alloy model, we need a method to translate the given database schema into additional Alloy model signatures. We provide a framework procedure of how it could be done.

Procedure 1. Let $R_1, R_2, \ldots, R_k$ be database tables, each with $n_1, n_2, \ldots, n_k$ columns, respectively. We create the following Alloy signature.

```
one sig Table {
    R_1: <field signature R_1>,
    R_2: <field signature R_2>,
              ⋮
    R_k: <field signature R_k>
}
```

where

$$\langle \textit{field signature } R_i \rangle = \begin{cases} \texttt{set Particle} & \text{if } n_i = 1 \\ \underbrace{\texttt{Particle -> Particle ->...-> Particle}}_{\text{repeated } n_i \text{ times}} & \text{if } n_i > 1 \end{cases}$$

In other words, each field `R_`i is an Alloy relation with n_i columns (ignoring the first dummy column of the `Table` signature*), and `Table.R_`i is the syntax which represents table R_i itself.

Note that the unary signature (i.e., table with single column) requires the keyword **set** in order to override the default behavior which is that `Table.R_`i would have contained a single row of data instead of any number of rows.

* The reason why we need a dummy signature `Table` is that it is impossible in Alloy to create a relation with each column consisting of object with only pre-defined signatures. Hence, this is a workaround.

Example of translation of database schema

To illustrate how the above procedure works, we consider the following example.

Example 7. Suppose that a database schema consists of three tables named **A**, **B**, and **C**. Table **A** has a single column of integers; Table **B** contains two columns of integers followed by a column of string values; and Table **C** contains two columns of string values.

Using Procedure 1, all of the tables above are translated into Alloy signature as follows.

```
one sig Table {
    A: set Particle
    B: Particle -> Particle -> Particle
    C: Particle -> Particle
}
```

Note. You may have probably noticed from the example above that we make *no distinction* between different types of data, whether it be an integer or a string (or any other types). Since data types do not make any differences in this project, they can be safely disregarded. ◇

2.4 Query function

Suppose that a (simplified) DRC query has the form

$$Q = \{x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m)\} \tag{1}$$

where each x_i represents a variable for scalar value and P is a boolean expression, which could be

- a boolean predicate in terms of table name (see page 3 under *table names as predicates*);
- an equality predicate (=) between two values;
- a conjunction ($\wedge$), a disjunction ($\vee$), a negation ($\neg$), a conditional ($\Longrightarrow$), or a bi-conditional ($\Longleftrightarrow$) of other boolean expressions; or

- a first-order, universal ($\forall$) or existential ($\exists$) quantification of other boolean expressions where a new variable also represents a scalar value.

In addition, each variable in the boolean expression P must be *binded*. That is, either the variable must be one of $x_1, x_2, \ldots, x_m$; or it must be introduced via first-order quantification in which the variable is used.

The procedure to transform a query in DRC into an Alloy function is described as follows.

Procedure 2. Given a specific `Universe` (i.e., domain set) called u as an input, we rewrite the query Q as defined in (**1**) as a new Alloy function (using a comprehension syntax) as shown.

```
fun query[u: Universe]: <output signature> {
    { x_1,x_2,...,x_m: u.Element | <predicate expression> }
}
```

The *<predicate expression>* above corresponds to the almost one-to-one translation of the boolean expression P (from the query Q as shown in (**1**)) into an Alloy predicate syntax string. The recursive translational algorithm is outlined below.

```
TranslateBooleanExp(P):
if P is a table-name predicate T(x_1,x_2,...,x_m):
    return "${x_1} → ${x_2} → ... → ${x_m} in Table.${T}"
else if P is the equality predicate x_1 = x_2:
    return "(${x_1} = ${x_2})"
else if P has the form ¬Q:
    return "(not ${TranslateBooleanExp(Q)})"
else if P has the form Q ∨ R:
    return "(${TranslateBooleanExp(Q)} or ${TranslateBooleanExp(R)})"
else if P has the form Q ∧ R:
    return "(${TranslateBooleanExp(Q)} and ${TranslateBooleanExp(R)})"
else if P has the form Q ⇒ R:
    return "(${TranslateBooleanExp(Q)} implies ${TranslateBooleanExp(R)})"
else if P has the form Q ⇔ R:
    return "(${TranslateBooleanExp(Q)} iff ${TranslateBooleanExp(R)})"
else if P has the form ∃y[Q]:
    return "(some ${y}: u.Element | ${TranslateBooleanExp(Q)})"
else if P has the form ∀y[Q]:
    return "(all ${y}: u.Element | ${TranslateBooleanExp(Q)})"
```

In other words, the translation propagates down through each boolean subexpressions, except for the case of table-name predicates in which we use "arrow products" (`->`) and "subset comparison operator" (`in`) to check if a tuple belongs to the given table. Hence, this procedure is straightforward.

For the *<output signature>*, we use the syntax similar to that in Procedure 1.

$$\textit{<output signature>} = \begin{cases} \texttt{set Superparticle} & \text{if } m = 1 \\ \underbrace{\texttt{Superparticle -> Superparticle ->...-> Superparticle}}_{\text{repeated } m \text{ times}} & \text{if } m > 1 \end{cases}$$

Example of translation of query functions

The following example demonstrates how we can translate one instance of DRC query into an Alloy function.

Example 8. Suppose that there exists a table $\mathbf{T}(\mathrm{a}, \mathrm{b})$ and we have a DRC query Q such that

$$Q = \{x, y \mid (x = y) \vee \mathbf{T}(x, y) \vee \exists z[\mathbf{T}(x, z) \wedge \mathbf{T}(z, y)]\}$$

Using Procedure 2, query Q can be rewritten as an Alloy function as follows.

```
1  fun query[u: Universe]: Superparticle -> Superparticle {
2      { x, y: u.Element |
3          ((x = y) or (x -> y in Table.T) or
4           (some z: u.Element | (x -> z in Table.T) and
5                                (z -> y in Table.T))) }
6  }
```

We have discussed earlier at the end of §1.2 about how the domain of a DRC query is *not* explicit. In an attempt to make the domain more explicit, we provide the following definition and notation.

Definition 1. Let Q be a DRC query as mentioned in (◢ **1**). If the domain set is D, then $Q[D]$ denotes

$$Q[D] = \{(x_1, x_2, \ldots, x_m) \in D^m \mid P(x_1, x_2, \ldots, x_m)\}$$

That is, each row of $Q[D]$ will be a tuple of m scalar values from the domain D.

We also sometimes interpret $Q[D]$ as the **result** of the query Q where D is the domain. This interpretation directly corresponds to how the `query` function works as defined by Procedure 2

2.5 Query safety verification

Let us revisit the definition of the "*safety* of DRC query" as described in §1.4. We rephrase the definition slightly so that it exactly fits the Alloy verification framework.

Definition 2. Suppose there exists a database schema S and a DRC query syntax Q (as defined in (◢ **1**)). The query Q is **safe** *if and only if*

- *for all* two domain sets D_1 and D_2, and
- *for every* database instances I based on the schema S, which are also valid under both domains D_1 and D_2 (i.e., all scalar values in I must belong to both sets D_1 and D_2)

we have $Q[D_1] = Q[D_2]$ (i.e., the result of query Q under D_1 and D_2 are the same).

The definiton of query safety can be translated into an Alloy assertion statement as shown on the first three lines in Source Code 2. The last line invokes the Alloy Analyzer to verify such assertion statement. The hightlighted number 4 indicates the upper limit of the number of objects of each model

```
assert queryIsSafe {
    all u, u': Universe | query[u] = query[u']
}
check queryIsSafe for 4
```

Source Code 2. Alloy assertion statement indicating that the `query` is true.

to be constructed by Alloy Analyzer. This number could be changed to a larger or smaller amount. The larger the amount, the more likely that Alloy Analyzer finds the answer, but in exchange for more computation resources.

Caveat of Alloy Analyzer

Note. It is vital to point out that Alloy Analyzer will only attempt to find a counterexample of a given assertion statement. If Alloy Analyzer fails to find such counterexample, it does *not* imply that the assertion is true. That is, the Alloy Analyzer is **not complete**. ◇

So far we have discussed all essential ingredients of our Alloy model to analyze the safety of a DRC query input. However, when the query safety assertion statement has a counterexample, and we would like to see the explicit visualization of the query result of the counterexample, we will need to provide the Alloy model signature for the result objects too.

2.6 Results placeholder

As we have previously discussed in Definition 1 that the query result is the interpretation of a query for a given domain set. Considering from the mathematical point of view, it is easy to see that the query result actually has the same anatomy as database tables. That is, a **query result** is a mathematical relation with a positive number of columns of scalar values. Therefore, we can use the syntax similarly to the declaration signature of table objects.

However, there is one major difference: for each query, we need two query results, one for domain `UniverseAlpha` and another for `UniverseBeta`. In Alloy, we use the following procedure to translate the DRC query to the Alloy object signature as shown below.

Procedure 3. Let Q be a given DRC query as defined in (◀ **1**). That is, each row of the result has m columns. We create Alloy objects with the following signatures.

```
abstract sig Result {
    Output: <field signature>
}
one sig ResultAlpha, ResultBeta extends Result {} {
    ResultAlpha.@Output = query[UniverseAlpha]
    ResultBeta.@Output = query[UniverseBeta]
}
```

where

$$\langle \textit{field signature} \rangle = \begin{cases} \texttt{set Superparticle} & \text{if } m = 1 \\ \underbrace{\texttt{Superparticle -> Superparticle ->...-> Superparticle}}_{\text{repeated } m \text{ times}} & \text{if } m > 1 \end{cases}$$

Notice that there are two separate results of the query (called the `Output`) from applying `query` function to each domain.

3 Examples Of Alloy Model Construction

Let us recap the previous section with a couple of comprehensive examples.

Example 9. Let us revisit Example 4. We have the database table **Follows**(fan, idol) and the following query.

$$Q_{\text{not following Alice}} = \{x \mid \neg \textbf{Follows}(x, \text{`Alice'})\}$$

We construct the corresponding Alloy program to verify whether the query is safe. The full source code is shown in Source Code 3. Notice that, at Line 22, we created the singleton constant table called `Alice`. This represents the scalar value which can be used as a constant in the query (such as on Line 27).

Once the source code was executed, the Alloy Analyzer finds multiple counterexamples to the safety assertion of the query. One counterexample instance among multiple counterexamples is shown in Figure 2.

It is easy to see that these results are correct with respect to their own domain sets, which implies that the aforementioned query $Q_{\text{not following Alice}}$ is **not safe**. ■

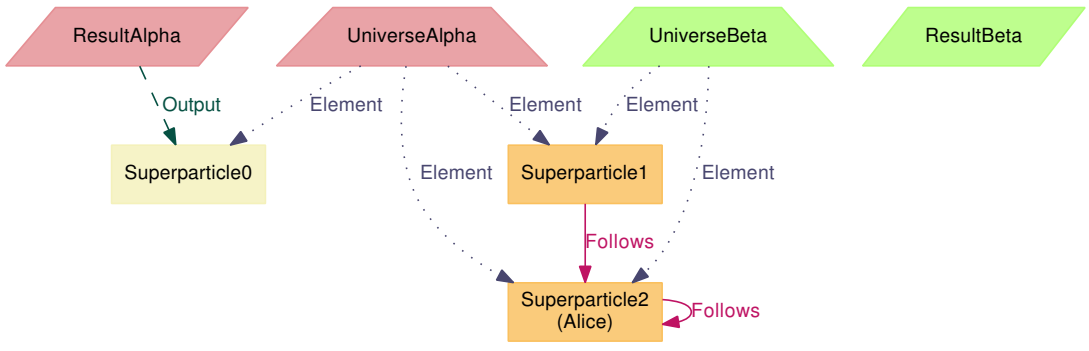

Figure 2. One counterexample of the Alloy program in Source Code 3. For an extensive explanation of this diagram, see the gray box on page 13.

```
/* Scalar values */
sig Superparticle {} {
  Superparticle = Universe.Element
}

/* Domains */
abstract sig Universe { Element: some Superparticle }
one sig UniverseAlpha, UniverseBeta extends Universe {}

/* Common domain */
some sig Particle in Superparticle {} {
  Particle = UniverseAlpha.Element & UniverseBeta.Element
}

/* Database Instance */
one sig Table {
    Follows: Particle -> Particle
}

/* Constant Values */
one sig Constant {
    Alice: Particle
}

/* Lists all people who are not following Alice */
fun query[u: Universe]: set Superparticle {
    { x: u.Element | not (x -> Constant.Alice in Table.Follows) }
}

/* Safety assertion */
assert queryIsSafe {
    all u, u': Universe | query[u] = query[u']
}

/* Results placeholder */
abstract sig Result {
    Output: set Superparticle
}
one sig ResultAlpha, ResultBeta extends Result {} {
    ResultAlpha.@Output = query[UniverseAlpha]
    ResultBeta.@Output = query[UniverseBeta]
}

/* Invoke the verification on the assertion */
check queryIsSafe for 4
```

Source Code 3. The complete Alloy program which verifies the query $Q_{\text{not following Alice}}$ as shown in section 9. Alloy codes highlighted in yellow indicate portions of code which are translated according to and depending on the original database schema and the DRC query.

Figure 2. One counterexample of the Alloy program in Source Code 3, *continued from page 12.*

In this database model, ■ dotted blue `Element` arrows indicate that the first domain, `UniverseAlpha`, contains three scalar values whereas the second domain, `UniverseBeta`, has only two values (and is incidentally the subset of the first). In addition, `Alice` is another name for `Superparticle2`.

The instance of the **Follows** (denoted with ■ solid pink `Follows` arrows) contains two rows, namely {(‘`Superparticle1`’, ‘`Superparticle2`’), (‘`Superparticle2`’, ‘`Superparticle2`’)} This table instance is *valid* because both values ‘`Superparticle1`’ and ‘`Superparticle2`’ belong to *both* domains.

The result of query under `UniverseAlpha` is a single row data {‘`Superparticle0`’} as shown by ■ dashed green `Output` dashed arrows). On the other hand, `UniverseBeta` yields empty results.

```
/* Scalar values */
sig Superparticle {} {
  Superparticle = Universe.Element
}

/* Domains */
abstract sig Universe { Element: some Superparticle }
one sig UniverseAlpha, UniverseBeta extends Universe {}

/* Common domain */
some sig Particle in Superparticle {} {
  Particle = UniverseAlpha.Element & UniverseBeta.Element
}

/* Database Instance */
one sig Table {
    Follows: Particle -> Particle
}

/* Lists all follows who follows every idols */
fun query[u: Universe]: set Superparticle {
    { x: u.Element | all y: u.Element |
        (some z: u.Element | z -> y in Table.Follows)
        implies (x -> y in Table.Follows) }
}

/* Safety assertion */
assert queryIsSafe {
    all u, u': Universe | query[u] = query[u']
}

/* Results placeholder */
abstract sig Result {
    Output: set Superparticle
}
one sig ResultAlpha, ResultBeta extends Result {} {
    ResultAlpha.@Output = query[UniverseAlpha]
    ResultBeta.@Output = query[UniverseBeta]
}

/* Invoke the verification on the assertion */
check queryIsSafe for 4
```

Source Code 4. The complete Alloy programw which verifies the query $Q_{\text{follows all v2}}$ as shown in section 10. Alloy codes highlighted in yellow indicates portions of code which are translated according and depending on the original database schema and DRC query. Also noted that, unlike Source Code 3, this Alloy program does not require the `Constant` model.

Now let us consider another example, particularly Example 6. In the original scenario, the query $Q_{\text{follows all}}$ is unsafe because the universal quantification may probe on domain values which lie outside the database. We are going to fix this query in the next example.

Example 10. Consider the database table **Follows**(fan, idol) and the DRC query

$$Q_{\text{follows all v2}} = \{x \mid \forall y[\exists z[\textbf{Follows}(z, y)] \Rightarrow \textbf{Follows}(x, y)]\}$$

By additionally injecting "$\exists z[\textbf{Follows}(z, y)] \Rightarrow$" into the original query, it *hopefully* guarantees that only idols (represented by y) having *at least* one follower in the database are considered. This should disregard the effect of dealing with different domain sets.

We translated the database schema as well as the above query into the Alloy program as shown in Source Code 4. However, once Alloy Analyzer was executed, we got unexpected results: it produces counterexamples. After a careful consideration the table **Follows** in every counterexample is empty.

Using Alloy to debug the query

Upon the second glance of the query $Q_{\text{follows all v2}}$, the result returned by Alloy Analyzer actually makes sense. If the table **Follows** is empty, then the query result would be the same set of people in the domain. *This is one prime example of Alloy could help us finding bugs in the design.*

We need to fix the program to guarantee that we only considered nonempty table **Follows**, which could be stated as an additional fact towards the `Table` signature declaration. Therefore, after the closing brace at the beginning of Line 18, we insert the fact (as highlighted) that the table `Follows` must contain some (i.e., at least one) rows.

```
15  /* Database Instance */
16  one sig Table {
17      Follows: Particle -> Particle
18  } { some Follows }
```

Once the the modified Alloy program is executed again, it no longer finds a counterexample. Even if we increase the upper limit of the number of each type of model objects from `4` to `8`, or even to `12`, Alloy still does not find a counterexample. Therefore, we may say that "the DRC query $Q_{\text{follows all v2}}$ *could be* safe, at least as long as the table **Follows** is *nonempty*." This is only a speculation since we could only hope that if Alloy were to find a counterexample, it should have found a small one already.

Alternative Approach

Instead of checking for database instances in which the table **Follows** is not empty, we could modify the query to make sure that the fan must reside in the database. Hence, let us say that the new DRC query is

$$Q_{\text{follows all v3}} = \{x \mid \exists w[\textbf{Follows}(x, w)] \wedge \forall y[\exists z[\textbf{Follows}(z, y)] \Rightarrow \textbf{Follows}(x, y)]\}$$

The first clause of the conjuction demands that the fan x must follow at least one person. In other words, that fan must at least exists in the database.

The Alloy function implementation of this query can be written as

```
20  /* Lists all follows who follows every idols */
21  fun query[u: Universe]: set Superparticle {
22      { x : u.Element |
          (some w: u.Element | x -> w in Table.Follows) and
          (all y: u.Element |
              (some z: u.Element | z -> y in Table.Follows)
              implies (x -> y in Table.Follows)) }
23  }
```

The only difference between this new code and the original one is shown as highlighted. Everything else remains the same apart from reformatting for clarity.

When we run this new code, the Alloy Analyzer also does not find any counterexamples, and thus the new query $Q_{\text{follows all v3}}$ may be safe.

Now that we have established how any simple database schemata and DRC queries can be translated into a verification task in Alloy language, in the next section we will experiment with more examples of safe and unsafe queries and see how accurate Alloy performs the verification tasks.

4 The Experiment

4.1 Experiment set-up

In this section, we evaluate how well our proposed Alloy verification framework performs on various DRC database queries. Below are the list of database tables in the schema.

- **Employee**(id, security-level): each employer ID and their security levels; and
- **Supervisor**(employee-id, boss-id): stores employee–boss pairs
- **Reviewer**(employee-id, year, reviewer-id): stores the employee ID of the performance reviewer for each employee in each year

We also consider the following DRC queries.

SAFE

- Q_1: Pairs of employees who share the same boss, and at least one of them has the same security level as their boss.

$$Q_1 = \{x, y \mid \exists b[\textbf{Supervisor}(x, b) \land \textbf{Supervisor}(y, b) \land \exists \ell[\textbf{Employee}(b, \ell) \land (\textbf{Employee}(x, \ell) \lor \textbf{Employee}(y, \ell))]]\}$$

UNSAFE

- Q_2: Employees without their own bosses.

$$Q_2 = \{x \mid \neg\exists b[\textbf{Supervisor}(x, b)]\}$$

- Q_3: Pairs of employees, one of which has the same security level as one of their bosses. UNSAFE

$$Q_3 = \{x, y \mid \exists b \exists \ell [\textbf{Employee}(b, \ell) \land (\textbf{Employee}(x, \ell) \land \textbf{Supervisor}(x, b) \lor \textbf{Employee}(y, \ell) \land \textbf{Supervisor}(y, b))]\}$$

- Q_4: Pairs of employees who review each other within the same year. SAFE

$$Q_4 = \{x, y \mid \exists t [\textbf{Reviewer}(x, t, y) \land \textbf{Reviewer}(y, t, x)]\}$$

- Q_5: *Super* employees who (a) has reviewed everyone else at some point (excluding themselves), and (b) is a boss of at least one employee. UNSAFE

$$Q_5 = \{b \mid \forall x \exists t [\textbf{Reviewer}(x, t, b)] \land \exists y [\textbf{Supervisor}(y, b) \land \neg(y = b)]\}$$

How to verify each specific query

The Alloy model for these DRC queries are shown in Source Code 5. By default the Alloy program verifies the query Q_1. If we wish to verify other queries, we need to modify some parts of the code (shown highlighted). Specifically,

- The query function name `query1` on lines 50, 59, and 60 should be replaced by names of other functions such as `query2`, `query3`, etc.
- The result placeholder `TwoColOutput` may need to be changed to `OneColOutput` depending of the output signature of the query function.

4.2 Verification result and analysis

As we expect, Alloy Analyzer has *correctly* determined whether all of these queries are safe. For unsafe queries Q_2, Q_3, and Q_5, Alloy finds the first counterexample in relatively short time (underlined in the output below). The console output for each of these unsafe queries are reproduced here (emphasis added).

```
query2:  Executing "Check queryIsSafe for 10"
            Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
            4658 vars. 1464 primary vars. 6953 clauses. 15ms.
            Counterexample found. Assertion is invalid. 17ms.
```

```
query3:  Executing "Check queryIsSafe for 10"
             Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
             38148 vars. 1464 primary vars. 128213 clauses. 2068ms.
            Counterexample found. Assertion is invalid. 189ms.
```

```
/* Scalar values */
sig Superparticle {} {
  Superparticle = Universe.Element
}

/* Domains */
abstract sig Universe { Element: some Superparticle }
one sig UniverseAlpha, UniverseBeta extends Universe {}

/* Common domain */
some sig Particle in Superparticle {} {
  Particle = UniverseAlpha.Element & UniverseBeta.Element
}

/* Database Instance */
one sig Table {
    Employee: Particle -> Particle,
    Supervisor: Particle -> Particle,
    Reviewer: Particle -> Particle -> Particle
}

/* Query functions */
fun query1[u: Universe]: Superparticle -> Superparticle {
    { x, y: u.Element | some b: u.Element |
        (x -> b in Table.Supervisor) and (y -> b in Table.Supervisor) and
        (some l: u.Element | (b -> l in Table.Employee) and
                             ((x -> l in Table.Employee) or (y -> l in Table.Employee))) }
}
fun query2[u: Universe]: set Superparticle {
    { x: u.Element | not some b: u.Element | x -> b in Table.Supervisor }
}
fun query3[u: Universe]: Superparticle -> Superparticle {
    { x, y: u.Element | some b, l: u.Element |
        (b -> l in Table.Employee) and
        ((x -> l in Table.Employee) and (x -> b in Table.Supervisor) or
         (y -> l in Table.Employee) and (y -> b in Table.Supervisor)) }
}
fun query4[u: Universe]: Superparticle -> Superparticle {
    { x, y: u.Element | some t: u.Element |
        (x -> t -> y in Table.Reviewer) or (y -> t -> x in Table.Reviewer) }
}
fun query5[u: Universe]: set Superparticle {
    { b: u.Element |
        (all x: u.Element | some t: u.Element | x -> t -> b in Table.Reviewer) and
        (some y: u.Element | (y -> b in Table.Supervisor) and not (y = b)) }
}

/* Safety assertion */
assert queryIsSafe {
    all u, u': Universe | query1[u] = query1[u']
}

/* Results placeholder */
abstract sig Result {
    OneColOutput: set Superparticle,
    TwoColOutput: Superparticle -> Superparticle
}
one sig ResultAlpha, ResultBeta extends Result {} {
    ResultAlpha.@TwoColOutput = query1[UniverseAlpha]
    ResultBeta.@TwoColOutput = query1[UniverseBeta]
}

/* Invoke the verification on the assertion */
check queryIsSafe for 4
```

Source Code 5. The complete Alloy program which verifies queries Q_1 through Q_5 as defined earlier in §4. To verify the safety of other query functions, some portions of the code (shown highlighted) needs to be modified.

```
query5:  Executing "Check queryIsSafe for 10"
            Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
            9498 vars. 1464 primary vars. 24953 clauses. 58ms.
            Counterexample found. Assertion is invalid. 99ms.
```

For safe queries Q_1 and Q_4, here is the console output (emphasis added).

```
query1:  Executing "Check queryIsSafe for 10"
            Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
            32858 vars. 1464 primary vars. 105823 clauses. 457ms.
            No counterexample found. Assertion may be valid. 16593ms.

query4:  Executing "Check queryIsSafe for 10"
            Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
            7898 vars. 1464 primary vars. 17663 clauses. 39ms.
            No counterexample found. Assertion may be valid. 132ms.
```

Note. In terms of computational cost,

- Queries with more complex syntax tend to take more time to verify (cf. `query1` vs. `query4`) even if the maximum numbers of objects for each type in the search space are the same.
- When comparing queries with similar complexity (such as `query1` vs. `query3`), unsafe queries take less computation time as it only requires to find one counterexample whereas safe queries need to exhaust the search space.

In addition, if we choose to vary the maximum number of objects for each model type in Alloy verification task, then undoubtedly, as the number increases, the computational cost also increases. This is a normal Alloy behavior so we would not include the result of such experiment here. ◇

5 Conclusion

So far we have established that we could use Alloy Analyzer to verity if a DRC query is safe under a given database schema. For future work, we could implement a translator to automate the verification process; that is, given as input the database specification and the query compatible with relational calculus, the translator converts the input into Alloy programming syntax to be verified by Alloy Analyzer.

On the other hand, the presented framework itself is still limited as it could deal only with a small subset of what DRC queries are capable of. Here are the list of features which could be added in this project in the future.

- **Support for all scalar value comparison operators.** In this project, the only allowed operation between one or more scalar values is the equality comparison operator (=). This does not fully reflect the behavior of the domain in real world.

  One possible improvement is to make sure that `Superparticle`s are totally ordered so that other comparison operators such as < and ≥ would work. On the same note, we should also be

able to refer to primitive constant values, particularly the maximum and minimum values in the domain.

This should be relatively easy to implement since there is an undocumented Alloy library which supports total ordering out of the box. This improvement will make operations such as number comparison possible to model.

- **Support for bounded and unbounded integers.** This is an extended improvement over the previous point. However, this should be much more complicated since the support for integers in Alloy is very limited, apart from basic arithmetic operations like addition, subtraction, etc.
- **Support for functional dependencies in database schema.** Functional dependency in a database systems enforces the values of some columns of data based on other columns. For example, the table **GeoData**(city, state, zipcode) may have the functional dependency

$$\text{zipcode} \mapsto \text{city}, \text{state}$$

That is, based on zipcode alone, there is a unique city and a unique state in the table **GeoData**.

This improvement will greatly benefit the query safety verification because (a) modeling functional dependency also implies modeling of primary keys, candidate keys, unique keys etc., and (b) the safety of DRC queries also depends on functional dependencies.

The modeling of functional dependencies in Alloy should be simple at least for tables with at most two columns. For tables with three or more columns, being able to specify the multiplicity between columns is much more complicated.

References

[AB88] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. Research Report RR-0846, INRIA, 1988.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*, chapter 5. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Cod72] Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.

[CP09] Alcino Cunha and Hugo Pacheco. Mapping between alloy specifications and database implementations. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 285–294, Washington, DC, USA, 2009. IEEE Computer Society.

[Fag82] Ronald Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, October 1982.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[NB11] Jaideep Nijjar and Tevfik Bultan. Bounded verification of ruby on rails data models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 67–77, New York, NY, USA, 2011. ACM.

[NBB15] Jaideep Nijjar, Ivan Bocić, and Tevfik Bultan. Data model property inference, verification, and repair for web applications. *ACM Trans. Softw. Eng. Methodol.*, 24(4):25:1–25:27, September 2015.

[Ull83] Jeffrey D. Ullman. *Principles of Database Systems*. W. H. Freeman & Co., New York, NY, USA, 2nd edition, 1983.

[WDSG06] Lin Wang, Gillian Dobbie, Jing Sun, and Lindsay Groves. Validating ora-ss data models using alloy. In *Proceedings of the Australian Software Engineering Conference*, ASWEC '06, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society.