Master’s Project

Safety Checking for Domain Relational Calculus Queries
Using Alloy Analyzer

Abhabongse “Plane” Janthong

Department of Computer Science, University of California, Santa Barbara

UCSB

CREATED AND EDITABLE USING IPE (http://ipe.otfried.org/)
Safety Checking for Domain Relational Calculus Queries Using Alloy Analyzer

Abhabongse “Plane” Janthong

Department of Computer Science, University of California, Santa Barbara
1
INTRODUCTION

Definition of database systems, domain relational calculus queries, and query safety.
What is relational database?

<table>
<thead>
<tr>
<th>PersonalData</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>BirthYear</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>1994</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>1995</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>1994</td>
</tr>
<tr>
<td>‘David’</td>
<td>1993</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Friendship</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NameA</td>
<td>NameB</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>‘Bob’</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>‘David’</td>
</tr>
</tbody>
</table>
What is relational database?

### PersonalData

<table>
<thead>
<tr>
<th>Name</th>
<th>BirthYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Alice’</td>
<td>1994</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>1995</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>1994</td>
</tr>
<tr>
<td>‘David’</td>
<td>1993</td>
</tr>
</tbody>
</table>

### Friendship

<table>
<thead>
<tr>
<th>NameA</th>
<th>NameB</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Alice’</td>
<td>‘Bob’</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>‘David’</td>
</tr>
</tbody>
</table>

Alice (b.1994) ✿ friendship ✿ Carol (b.1994)
Bob (b.1995)    ✿ friendship ✿ David (b.1993)
Carol (b.1994) ✿ friendship ✿ David (b.1993)
What is relational database?

**Database:** a collection of **tables**.

<table>
<thead>
<tr>
<th>PersonalData</th>
<th>Friendship</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>BirthYear</strong></td>
</tr>
<tr>
<td>‘Alice’</td>
<td>1994</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>1995</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>1994</td>
</tr>
<tr>
<td>‘David’</td>
<td>1993</td>
</tr>
</tbody>
</table>
What is relational database?

Database: a collection of tables.

Table: a mathematical relation over one or more sets of scalar values (numbers, strings, etc.).

In this particular example, each table is a binary relation over sets of scalar values.
What is relational database?

Database: a collection of tables.

Table: a mathematical relation over one or more sets of scalar values (numbers, strings, etc.).
What is relational database?

**Database**: a collection of **tables**.

**Table**: a mathematical relation over **one or more** sets of **scalar values** (numbers, strings, etc.).

**Tuple**: a row of the **table**.

<table>
<thead>
<tr>
<th>PersonalData</th>
<th>Friendship</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>BirthYear</strong></td>
</tr>
<tr>
<td>Alice</td>
<td>1994</td>
</tr>
<tr>
<td>Bob</td>
<td>1995</td>
</tr>
<tr>
<td>Carol</td>
<td>1994</td>
</tr>
<tr>
<td>David</td>
<td>1993</td>
</tr>
</tbody>
</table>
**What is relational database?**

**PersonalData**

<table>
<thead>
<tr>
<th>Name</th>
<th>BirthYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Alice’</td>
<td>1994</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>1995</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>1994</td>
</tr>
<tr>
<td>‘David’</td>
<td>1993</td>
</tr>
</tbody>
</table>

**Friendship**

<table>
<thead>
<tr>
<th>NameA</th>
<th>NameB</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Alice’</td>
<td>‘Bob’</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>‘Carol’</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>‘David’</td>
</tr>
</tbody>
</table>

**Database:** a collection of **tables**.

**Table:** a mathematical relation over **one or more**

sets of **scalar values** (numbers, strings, etc.).

**Tuple:** a row of the **table**.

---

*For this project, we ignore the concept of keys*

(primary keys, foreign keys, etc.)
Database queries

**Query:** the process of fetching the stored data from the database.
Database queries

**Query:** the process of fetching the stored data from the database.

Example of **SQL query:**

```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```
Database queries

Query: the process of fetching the stored data from the database.

Example 1. All students and their year of birth who were born strictly before 1995.

Example of SQL query:

```sql
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```
# Database queries

**Query**: the process of fetching the stored data from the database.

**Example 1.** All students and their year of birth who were born *strictly* before 1995.

Example of **SQL query**:  
```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

<table>
<thead>
<tr>
<th>PersonalData</th>
<th>Query Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>BirthYear</td>
<td>BirthYear</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>‘Alice’</td>
<td>1994</td>
</tr>
<tr>
<td>‘Bob’</td>
<td>1995</td>
</tr>
<tr>
<td>‘Carol’</td>
<td>1994</td>
</tr>
<tr>
<td>‘David’</td>
<td>1993</td>
</tr>
</tbody>
</table>
Database queries

**Query:** the process of fetching the stored data from the database.

**Example 1.** All students and their year of birth who were born *strictly* before 1995.

Example of **SQL query:**

```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

Example of **Domain Relational Calculus (DRC) query:**

\[
Q_{\text{before } 1995} = \{ \text{name, year} \mid \text{PersonalData}(\text{name, year}) \land (\text{year} < 1995) \}
\]

<table>
<thead>
<tr>
<th>PersonalData</th>
<th>Query Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>BirthYear</td>
</tr>
<tr>
<td>'Alice'</td>
<td>1994</td>
</tr>
<tr>
<td>'Bob'</td>
<td>1995</td>
</tr>
<tr>
<td>'Carol'</td>
<td>1994</td>
</tr>
<tr>
<td>'David'</td>
<td>1993</td>
</tr>
</tbody>
</table>
Database queries

**Query:** the process of fetching the stored data from the database.

**Example 1.** All students and their year of birth who were born strictly before 1995.

Example of **SQL query:**

```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

Example of **Domain Relational Calculus (DRC) query:**

```
Q_{before \, 1995} = \{name, year \mid PersonalData(name, year) \land (year < 1995)\}
```

- Use set comprehension notation, in first-order logic.

<table>
<thead>
<tr>
<th>PersonalData</th>
<th>Query Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>BirthYear</td>
</tr>
<tr>
<td>'Alice'</td>
<td>1994</td>
</tr>
<tr>
<td>'Carol'</td>
<td>1994</td>
</tr>
<tr>
<td>'David'</td>
<td>1993</td>
</tr>
</tbody>
</table>

| 'Alice'      | 'Alice' 1994 |
| 'Carol'      | 'Carol' 1994 |
| 'David'      | 'David' 1993 |
Database queries

**Query:** the process of fetching the stored data from the database.

**Example 1.** All students and their year of birth who were born strictly before 1995.

Example of **SQL query:** 
```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

Example of **Domain Relational Calculus (DRC) query:**

\[
Q_{\text{before } 1995} = \{ \text{name, year} \mid \text{PersonalData(name, year)} \land (\text{year} < 1995) \}
\]

- Use set comprehension notation, in first-order logic.
- Identifiers always represent scalar values.
Database queries

Query: the process of fetching the stored data from the database.

Example 1. All students and their year of birth who were born strictly before 1995.

Example of SQL query:  
```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

Example of Domain Relational Calculus (DRC) query:

\[ Q_{\text{before } 1995} = \{ \text{name, year} \mid \text{PersonalData}(\text{name, year}) \land (\text{year} < 1995) \} \]

- Use set comprehension notation, in first-order logic.
- Identifiers always represent scalar values.
- Table names: predicate to indicate whether a specified tuple exists in such table.
Database queries

Query: the process of fetching the stored data from the database.

Example 1. All students and their year of birth who were born strictly before 1995.

<table>
<thead>
<tr>
<th>Name</th>
<th>BirthYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Alice'</td>
<td>1994</td>
</tr>
<tr>
<td>'Bob'</td>
<td>1995</td>
</tr>
<tr>
<td>'Carol'</td>
<td>1994</td>
</tr>
<tr>
<td>'David'</td>
<td>1993</td>
</tr>
</tbody>
</table>

Example of SQL query: `SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995`

Example of Domain Relational Calculus (DRC) query:

\[ Q \text{ before } 1995 = \{ \text{name, year} \mid \text{PersonalData}(\text{name, year}) \land (\text{year} < 1995) \} \]

For example, `PersonalData('Alice', 1994)` is true, whereas `PersonalData('Bob', 1993)` is false.

Table names: predicate to indicate whether a specified tuple exists in such table.

Identifiers always represent scalar values.
Database queries

**Query:** the process of fetching the stored data from the database.

**Example 1.** All students and their year of birth who were born *strictly* before 1995.

Example of **SQL query:**

```
SELECT Name, BirthYear FROM PersonalData WHERE BirthYear < 1995
```

Example of **Domain Relational Calculus (DRC) query:**

```
Q_{before\ 1995} = \{name, year \mid \text{PersonalData}(name, year) \land (year < 1995)\}
```

Identifiers always represent scalar values.

There are other variants of Relational Calculus, *namely Tuple Relational Calculus.* Other types of queries include Datalog, etc.
More examples of DRC queries

Example 2. All friends of Bob.

\[ Q_{\text{Bob’s friend}} = \{name | \text{Friendship}(name, ‘Bob’) \lor \text{Friendship}(‘Bob’, name)\} \]
More examples of DRC queries

**Example 2.** All friends of Bob.

\[
Q_{\text{Bob's friend}} = \{ name \mid \text{Friendship}(name, \text{'Bob'}) \lor \text{Friendship}(\text{'Bob'}, name) \}
\]

**Example 3.** All pairs of students who share a common friend.

\[
Q_{\text{friend of friend}} = \{ x, y \mid (x < y) \land \exists z[(\text{Friendship}(x, z) \lor \text{Friendship}(z, x)) \land (\text{Friendship}(y, z) \lor \text{Friendship}(z, y))]]\}
\]
More examples of DRC queries

Example 2. All friends of Bob.

\[ Q_{\text{Bob's friend}} = \{name | \text{Friendship}(name, 'Bob') \lor \text{Friendship}('Bob', name) \} \]

Example 3. All pairs of students who share a common friend.

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

Notice that identifiers do not have explicit domain in the query. Is this okay?
Domain in DRC queries

Is it fine that identifiers in DRC query **do not have explicit domain**?
Domain in DRC queries

Is it fine that identifiers in DRC query *do not have explicit domain*?

**NOT ALWAYS**
Domain in DRC queries

Is it fine that identifiers in DRC query do not have explicit domain?

NOT ALWAYS

Example 1. All students and their year of birth who were born strictly before 1995.

\[ Q^*_{\text{before } 1995} = \{ \text{name, year} \mid \text{year} < 1995 \} \]
Domain in DRC queries

Is it fine that identifiers in DRC query do not have explicit domain?

**NOT ALWAYS**

Example 1. All students and their year of birth who were born strictly before 1995.

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

Mathematically speaking, we cannot determine the result if the domain is not established.
Domain in **DRC** queries

Is it fine that identifiers in **DRC** query **do not have explicit domain**?

**NOT ALWAYS**

Example 1. All students and their year of birth who were born **strictly** before 1995.

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

Mathematically speaking, we **cannot** determine the result if the **domain is not established**.

- If the domain of year is a **set of integers**, then (‘Alice’, -80) is part of the result.
Domain in DRC queries

Is it fine that identifiers in DRC query do not have explicit domain?

**NOT ALWAYS**

**Example 1.** All students and their year of birth who were born strictly before 1995.

\[ Q_{\text{before 1995}}^* = \{ \text{name, year} \mid \text{year} < 1995 \} \]

Mathematically speaking, we cannot determine the result if the domain is not established.

- If the domain of year is a set of integers, then (‘Alice’, -80) is part of the result.
- If the domain of year is a set of positive integers, then (‘Alice’, -80) is not part of the result.
Domain in DRC queries

Is it fine that identifiers in DRC query do not have explicit domain?

**NOT ALWAYS**

**Example 1.** All students and their year of birth who were born strictly before 1995.

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

Mathematically speaking, we cannot determine the result if the domain is not established.

- If the domain of year is a set of integers, then (‘Alice’, -80) is part of the result.
- If the domain of year is a set of positive integers, then (‘Alice’, -80) is not part of the result.
Domain in DRC queries

Is it fine that identifiers in DRC query do not have explicit domain?

**NOT ALWAYS**

**Example 1.** All students and their year of birth who were born strictly before 1995.

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

Mathematically speaking, we cannot determine the result if the domain is not established.

Other way to look at this: it queries for data that might not be bounded by the database.

If the domain of year is a set of positive integers, then (‘Alice’, -80) is not part of the result.
Domain in DRC queries

Is it fine that identifiers in DRC query **do not have explicit domain**?

**NOT ALWAYS**

**Example 1.** All students and their year of birth who were born **strictly** before 1995.

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

Mathematically speaking, we **cannot** determine the result if the **domain is not established**.

Other way to look at this: it queries for data that might **not be bounded by the database**.

Or even: the result is **infinite**, which implies that the result depends on the domain.
Domain-independency (safety)

A DRC query is **domain-independent** if the result of the query **depends on only the data in the database and not on the domain set.**
Domain-independency (safety)

A DRC query is **domain-independent** if the result of the query **depends on only the data in the database and not on the domain set.**

\[
Q_{\text{before } 1995} = \{ name, year \mid \text{PersonalData}(name, year) \land (year < 1995) \}
\]

\[
Q_{\text{Bob’s friend}} = \{ name \mid \text{Friendship}(name, ‘Bob’) \lor \text{Friendship}(‘Bob’, name) \}
\]

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

\[
Q^*_{\text{before } 1995} = \{ name, year \mid year < 1995 \}
\]
Domain-independency (safety)

A DRC query is **domain-independent** if the result of the query depends on only the data in the database and not on the domain set.

\[
\begin{align*}
Q_{\text{before 1995}} &= \{\text{name, year} \mid \text{PersonalData(name, year)} \land (\text{year} < 1995)\} \\
Q_{\text{Bob’s friend}} &= \{\text{name} \mid \text{Friendship(name, ‘Bob’) \lor Friendship(‘Bob’, name)}\} \\
Q_{\text{friend of friend}} &= \{x, y \mid (x < y) \land \exists z[(\text{Friendship(x, z) \lor Friendship(z, x)}) \\
&\quad \land (\text{Friendship(y, z) \lor Friendship(z, y)})]\} \\
Q^*_{\text{before 1995}} &= \{\text{name, year} \mid \text{year} < 1995\}
\end{align*}
\]
Domain-independency (safety)

A DRC query is **domain-independent** if the result of the query **depends on only the data in the database** and not on the domain set.

\[
Q_{\text{before 1995}} = \{ name, year \mid \text{PersonalData}(name, year) \land (year < 1995) \}
\]

\[
Q_{\text{Bob's friend}} = \{ name \mid \text{Friendship}(name, 'Bob') \lor \text{Friendship('Bob', name)} \}
\]

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

\[
Q^*_{\text{before 1995}} = \{ name, year \mid year < 1995 \}
\]
Domain-independency (safety)

A DRC query is **domain-independent** if the result of the query **depends on only the data in the database and not on the domain set**.

- **SAFE**
  
  \[
  Q_{\text{before 1995}} = \{ \text{name, year} \mid \text{PersonalData}(\text{name, year}) \land (\text{year} < 1995) \}
  \]

  \[
  Q_{\text{Bob’s friend}} = \{ \text{name} \mid \text{Friendship}(\text{name, ‘Bob’}) \lor \text{Friendship(‘Bob’, name)} \}
  \]

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

- **UNSAFE**
  
  \[
  Q_{\text{before 1995}}^* = \{ \text{name, year} \mid \text{year} < 1995 \}
  \]

  ...and more...
More example of DRC unsafe query

Example 4. People who do not follow Alice.

We have the database table Follows(fan, idol) representing the fact that fan is following idol on a social network.
More example of DRC unsafe query

**Example 4.** People who do not follow Alice.

\[ Q \text{ not following Alice} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]
More example of DRC unsafe query

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Suppose that \( D_1, D_2 \) are distinct domain sets such that \( D_2 = D_1 \cup \{ c \} \) where \( \text{Follows}(c, 'Alice') \) is false. Then,
More example of DRC unsafe query

**Example 4.** People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Suppose that \( D_1, D_2 \) are distinct domain sets such that \( D_2 = D_1 \cup \{ c \} \) where \( \text{Follows}(c, 'Alice') \) is false. Then,

- Result of the query under \( D_1 \) does not contain \( (c) \).
More example of DRC unsafe query

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Suppose that \( D_1, D_2 \) are distinct domain sets such that \( D_2 = D_1 \cup \{ c \} \) where \( \text{Follows}(c, 'Alice') \) is false. Then,

- Result of the query under \( D_1 \) does not contain \( (c) \).
- Result of the query under \( D_2 \) contains \( (c) \).
More example of DRC unsafe query

**Example 4.** People who do not follow Alice.

\[ Q \text{ not following Alice} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Suppose that \( D_1, D_2 \) are distinct domain sets such that \( D_2 = D_1 \cup \{ c \} \) where \( \text{Follows}(c, 'Alice') \) is false. Then,

- Result of the query under \( D_1 \) does **not** contain \( (c) \).
- Result of the query under \( D_2 \) contains \( (c) \).

We have the database table \( \text{Follows}(\text{fan}, \text{idol}) \) representing the fact that fan is following idol on a social network.
Even more examples of DRC unsafe queries

**Example 5.** Set of pairs of people such that the first person follows Alice or the second person follows Bob.

\[ Q_{\text{weird pairing}} = \{ x, y \mid \text{Follows}(x, 'Alice') \lor \text{Follows}(y, 'Bob') \} \]
Even more examples of DRC unsafe queries

Example 5. Set of pairs of people such that the first person follows Alice or the second person follows Bob.

\[ Q_{\text{weird pairing}} = \{ x, y \mid \text{Follows}(x, 'Alice') \lor \text{Follows}(y, 'Bob') \} \]

As long as there is a person following Bob, then \((x, y)\) would be in the result for every \(x\) in the domain.

COUNTEREXAMPLE
Even more examples of DRC unsafe queries

**Example 5.** Set of pairs of people such that the first person follows Alice or the second person follows Bob.

\[
Q_{\text{weird pairing}} = \{x, y \mid \text{Follows}(x, 'Alice') \lor \text{Follows}(y, 'Bob')\}
\]

As long as there is a person following Bob, then \((x, y)\) would be in the result for every \(x\) in the domain.

**Example 6.** People who follows everyone.

\[
Q_{\text{follows all}} = \{x \mid \forall y[\text{Follows}(x, y)]\}
\]
Even more examples of DRC unsafe queries

**Example 5.** Set of pairs of people such that the first person follows Alice or the second person follows Bob.

\[
Q_{\text{weird pairing}} = \{x, y \mid \text{Follows}(x, 'Alice') \lor \text{Follows}(y, 'Bob')\}
\]

As long as there is a person following Bob, then \((x, y)\) would be in the result for every \(x\) in the domain.

**Example 6.** People who follows everyone.

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

If the result is not empty under some particular domain, then adding an alien to the domain will make the result empty.
**Even more examples of DRC unsafe queries**

**Example 5.** Set of pairs of people such that the first person follows Alice or the second person follows Bob.

\[ Q_{\text{weird pairing}} = \{ x, y \mid \text{Follows}(x, 'Alice') \lor \text{Follows}(y, 'Bob') \} \]

As long as there is a person following Bob, then \((x, y)\) would be in the result for every \(x\) in the domain.

**Example 6.** People who follows everyone.

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

The result of query in **Example 6** is **guaranteed to be bounded** even if the domain was infinite, but regardless of that, it is still **domain-dependent (unsafe)**.
2

MAIN PROBLEM

Formulation of main verification problem and introducing the main verification tool.
Main problem

Suppose that we have a database schema and a DRC query of the form

\[ Q = \{ x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m) \} \]

boolean expression
Main problem

Suppose that we have a database schema and a DRC query of the form

\[
Q = \{ x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m) \}
\]

We will get into the structure of the boolean expression \( P \) later.
Main problem

Suppose that we have a database schema and a DRC query of the form

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

To verify that query $Q$ is safe, we check that

We will get into the structure of the boolean expression $P$ later.
Main problem

Suppose that we have a database schema and a DRC query of the form

\[ Q = \{x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m)\} \]

To verify that query \( Q \) is safe, we check that

- for every pair of domain sets \( D_1 \) and \( D_2 \), and
- for every database instance under the schema (which is also valid under both domains \( D_1 \) and \( D_2 \))

We will get into the structure of the boolean expression \( P \) later.
Main problem

Suppose that we have a database schema and a DRC query of the form

\[ Q = \{ x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m) \} \]

boolean expression

To verify that query \( Q \) is safe, we check that

- for every pair of domain sets \( D_1 \) and \( D_2 \), and
- for every database instance under the schema (which is also valid under both domains \( D_1 \) and \( D_2 \))

Then, the result of the query under the assumption of domain \( D_1 \) (denoted \( Q[D_1] \)) is equal to that under the assumption of domain \( D_2 \) (denoted \( Q[D_2] \)).

We will get into the structure of the boolean expression \( P \) later.
Suppose that we have a database schema and a DRC query of the form

\[ Q = \{ x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m) \} \]

To verify that query \( Q \) is safe, we check that

- for every pair of domain sets \( D_1 \) and \( D_2 \), and
- for every database instance under the schema (which is also valid under both domains \( D_1 \) and \( D_2 \))

Then, the result of the query under the assumption of domain \( D_1 \) (denoted \( Q[D_1] \)) is equal to that under the assumption of domain \( D_2 \) (denoted \( Q[D_2] \)).

i.e., the result is always the same, \( Q[D_1] = Q[D_2] \), for any pairs of domains \( D_1 \) and \( D_2 \).
Suppose that we have a database schema and a DRC query of the form

\[ Q = \{ x_1, x_2, \ldots, x_m \mid P(x_1, x_2, \ldots, x_m) \} \]

To verify that query \( Q \) is safe, we check that

1. for every pair of domain sets \( D_1 \) and \( D_2 \), and
2. for every database instance under the schema (which is also valid under both domains \( D_1 \) and \( D_2 \))

Then, the result of the query under the assumption of domain \( D_1 \) (denoted \( Q[D_1] \)) is equal to that under the assumption of domain \( D_2 \) (denoted \( Q[D_2] \)).

i.e., the result is always the same, \( Q[D_1] = Q[D_2] \), for any pairs of domains \( D_1 \) and \( D_2 \).

We can model all of this in Alloy.
What is 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-assumed properties.
What is 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-assumed properties.

---

*Model signature definitions*
What is 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-assumed properties.

- **Model signature definitions**
- **Assumed facts or properties**
What is 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-assumed properties.
What is 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-assumed properties.

Actually, Alloy Analyzer will attempt to find a counterexample to the asserted property.

If Alloy does not find a counterexample, it does not mean that the asserted property is true.
What is 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-assumed properties.

The tool was developed by Daniel Jackson and his team at the Massachusetts Institute of Technology (MIT).

http://alloy.mit.edu/

Actually, Alloy Analyzer will attempt to find a counterexample to the asserted property.

* If Alloy does not find a counterexample, it does not mean that the asserted property is true.
For a given database tables $R_1, \ldots, R_k$ and a given DRC query $Q$, Coming up next ...
Task summary

Coming up next …

For a given **database tables** \(R_1, \ldots, R_k\) and a given **DRC query** \(Q\),

- we provide a method to translate the **tables into Alloy model signature**
Task summary

Coming up next …

For a given database tables $R_1, \ldots, R_k$ and a given DRC query $Q$,

- we provide a method to translate the tables into Alloy model signature
- and the query into an Alloy function.
Coming up next …

For a given database tables $R_1, \ldots, R_k$ and a given DRC query $Q$,

1. we provide a method to translate the tables into Alloy model signature
2. and the query into an Alloy function.

We also provide additional components to set-up the verification task in Alloy to determine whether the given query is safe or not.
**Task summary**

Coming up next …

For a given database tables $R_1, \ldots, R_k$ and a given DRC query $Q$,

- we provide a method to translate the tables into Alloy model signature
- and the query into an Alloy function.

We also provide additional components to set-up the verification task in Alloy to determine whether the given query is safe or not.

Additional components include

- model signatures for domain sets, scalar values, and optional query result
- and safety assertion statement for the query.
Task summary

Coming up next …

For a given database tables $R_1, \ldots, R_k$ and a given DRC query $Q$,

1. we provide a method to translate the tables into Alloy model signature ②
2. and the query into an Alloy function. ③

We also provide additional components to set-up the verification task in Alloy to determine whether the given query is safe or not.

Additional components include

1. model signatures for domain sets, scalar values, and optional query result ⑤
2. and safety assertion statement for the query. ④
3.1 TRANSLATION TO ALLOY MODEL

Example 4. People who do not follow Alice.

\[ Q \text{ not following Alice} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

We demonstrate how to translate database schema and DRC queries into Alloy syntax with an example.
Domain sets and scalar values

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Example 4. People who do not follow Alice.
Domain sets and scalar values

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

```
sig Superparticle {} {
    Superparticle = Universe.Element
}

abstract sig Universe { Element: some Superparticle }

one sig UniverseAlpha, UniverseBeta extends Universe {}

some sig Particle in Superparticle {} {
    Particle = UniverseAlpha.Element & UniverseBeta.Element
}
```

This definition is always static for all verification tasks.
We need to be able to consider different domain sets in order to ultimately determine if a query is domain-dependent.
Domain sets and scalar values

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

This definition is always static for all verification tasks.

We need to be able to consider different domain sets in order to ultimately determine if a query is domain-dependent.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{x \mid \neg \text{Follows}(x, 'Alice')\} \]

This definition is always **static for all verification tasks**.
We need to be able to consider **different** domain sets in order to ultimately determine if a query is **domain-dependent**.
Example 4. People who do not follow Alice.

\[ Q \text{ not following Alice} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Each database table is declared as a field of the main signature Table, and the **multiplicity** must reflect the **number of columns** in the table.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Each database table is declared as a field of the main signature `Table`, and the **multiplicity** must reflect the **number of columns** in the table.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Each database table is declared as a field of the main signature `Table`, and the `multiplicity` must reflect the number of columns in the table.

- If there is more than 1 table in the schema, then the field signature of each table must be separated by comma.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Each database table is declared as a field of the main signature `Table`, and the `multiplicity` must reflect the **number of columns** in the table.

- If there is **more than 1 table** in the schema, then the field signature of each table must be separated by comma.

- If the table has **exactly 1 column**, then the field signature is `set Particle`
  Otherwise, it is the keyword `Particle` repeated with the number of times equal to the number of columns, separated by `->`. 
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Each database table is declared as a field of the main signature `Table`, and the **multiplicity** must reflect the **number of columns** in the table.

- If there is **more than 1 table** in the schema, then the field signature of each table must be separated by comma.

- If the table has **exactly 1 column**, then the field signature is `set` `Particle`. Otherwise, it is the keyword `Particle` repeated with the number of times equal to the number of columns, separated by `->`. 

```alloy
one sig Table {
    Follows: Particle -> Particle
}
```
Query function

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]
### Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x | \neg \text{Follows}(x, 'Alice') \} \]

---

```alloy
14 one sig Constant {
15   Alice: Particle
16 }
17
18 fun query[u: Universe]: set Superparticle {
19   { x: u.Element | not (x -> Constant.Alice in Table.Follows) }
20 }
```

- The input to the `query` function in Alloy is the domain set.
### Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{x \mid \neg \text{Follows}(x, 'Alice') \} \]

The translation of **boolean expression** is mostly straightforward.

```alloy
one sig Constant {
   Alice: Particle
}

fun query[u: Universe]: set Superparticle {
   { x: u.Element | not (x -> Constant.Alice in Table.Follows) }
}
```

- definition of constant appeared in query
- output signature (same format as table’s field signature)
- boolean expression
- all identifiers separated by commas

Non-highlighted codes are always static for all verification tasks.

[SAFETY CHECKING FOR DRC QUERIES USING ALLOY ANALYZER | ABHABONGSE JANTHONG]
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

The translation of **boolean expression** is mostly straightforward.

- For a **conjunction** (\(\wedge\)), a **disjunction** (\(\vee\)), a **negation** (\(\neg\)), a **conditional** (\(\Rightarrow\)), a **bi-conditional** (\(\Leftrightarrow\)), or a **universal** (\(\forall\)) or **existential** (\(\exists\)) quantification of other boolean expressions; the translation **propagates** down the expression tree.
Query function

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

The translation of **boolean expression** is mostly straightforward.

- For a **conjunction** (\( \wedge \)), a **disjunction** (\( \vee \)), a **negation** (\( \neg \)), a **conditional** (\( \Rightarrow \)), a **bi-conditional** (\( \Leftrightarrow \)), or a **universal** (\( \forall \)) or **existential** (\( \exists \)) quantification of **other boolean expressions**; the translation propagates down the expression tree.

- For a boolean predicate in terms of **table name**; the **tuple** is constructed using arrow products (\( \rightarrow \)), and the set member operation (\( \text{in} \)) checks if the tuple belongs to the specified table.
3 Query function > Translating boolean expression

```plaintext
1  TranslateBooleanExp(P):
2  if P is a table-name predicate T(x₁, x₂, ..., xₘ):
3      return "${x₁} → ${x₂} → ... → ${xₘ} in Table.${T}"
4  else if P is the equality predicate x₁ = x₂:
5      return "(${x₁} = ${x₂})"
6  else if P has the form ¬Q:
7      return "(not ${TranslateBooleanExp(Q)})"
8  else if P has the form Q ∨ R:
9      return "(${TranslateBooleanExp(Q}) or ${TranslateBooleanExp(R)})"
10  else if P has the form Q ∧ R:
11     return "(${TranslateBooleanExp(Q}) and ${TranslateBooleanExp(R)})"
12  else if P has the form Q ⇒ R:
13     return "(${TranslateBooleanExp(Q}) implies ${TranslateBooleanExp(R)})"
14  else if P has the form Q ⇔ R:
15     return "(${TranslateBooleanExp(Q}) iff ${TranslateBooleanExp(R)})"
16  else if P has the form ∃y[Q]:
17     return "(some ${y}: u.Element | ${TranslateBooleanExp(Q)})"
18  else if P has the form ∀y[Q]:
19     return "(all ${y}: u.Element | ${TranslateBooleanExp(Q)})"
```
Safety verification for query

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, \text{Alice}) \} \]

Example 4. People who do not follow Alice.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

This definition is always static for all verification tasks.
Safety verification for query

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{x \mid \neg \text{Follows}(x, 'Alice') \} \]

This definition is always static for all verification tasks.

Except for the upper limit of the number of object of each model to be constructed by Alloy Analyzer while looking for counterexample.
### Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

This definition is always **static for all verification tasks**.

- Except for the **upper limit** of the number of object of each model to be constructed by Alloy Analyzer while looking for counterexample.

All of the Alloy codes up to this point is sufficient for the verification.

- Unless the visualization of the counterexample is wanted.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{x \mid \neg \text{Follows}(x, 'Alice')\} \]
Example 4. People who do not follow Alice.

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

This definition is always **static for all verification tasks**.
Optional results placeholder

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

```plaintext
25 abstract sig Result {
26     Output: set Superparticle
27 }
28 one sig ResultAlpha, ResultBeta extends Result {} {
29     ResultAlpha.@Output = query[UniverseAlpha]
30     ResultBeta.@Output = query[UniverseBeta]
31 }
```

This definition is always static for all verification tasks.

Except for the signature for the Output field of the query Result object, which will be exactly the same as the output signature of the Alloy function query.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}( x, 'Alice' ) \} \]

This definition is always static for all verification tasks.

Except for the signature of the Output field of the query Result object, which will be exactly the same as the output signature of the Alloy function query.

The output is binded to the query result when the domain is applied.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

```alloy
1 /* Scalar values */
2 sig Superparticle () {
3    Superparticle = Universe.Element
4 }
5
6 /* Domains */
7 abstract sig Universe { Element: some Superparticle }
8 one sig UniverseAlpha, UniverseBeta extends Universe {}
9
10 /* Common domain */
11 some sig Particle in Superparticle () {
12    Particle = UniverseAlpha.Element & UniverseBeta.Element
13 }
14
15 /* Database Instance */
16 one sig Table {
17    Follows: Particle -> Particle
18 }
19
20 /* Constant Values */
21 one sig Constant {
22    Alice: Particle
23 }
24
25 /* Lists all people who are not following Alice */
26 fun query[u: Universe]: set Superparticle {
27    { x: u.Element | not (x -> Constant.Alice in Table.Follows) }
28 }
29
30 /* Safety assertion */
31 assert queryIsSafe {
32    all u, u': Universe | query[u] = query[u']
33 }
34
35 /* Results placeholder */
36 abstract sig Result {
37    Output: set Superparticle
38 }
39 one sig ResultAlpha, ResultBeta extends Result () {
40    ResultAlpha.@Output = query[UniverseAlpha]
41    ResultBeta.@Output = query[UniverseBeta]
42 }
43
44 /* Invoke the verification on the assertion */
45 check queryIsSafe for 4
```
Verification outcome

Example 4. People who do not follow Alice.

\[ \text{Q not following Alice} = \{x \mid \neg \text{Follows}(x, 'Alice') \} \]

Once the code is run, Alloy Analyzer finds a counterexample.
Verification outcome

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Once the code is run, Alloy Analyzer finds a counterexample.
**Example 4.** People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{x \mid \neg \text{Follows}(x, 'Alice')\} \]

Once the code is run, Alloy Analyzer finds a **counterexample**.
Verification outcome

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, \text{Alice}) \} \]

Once the code is run, Alloy Analyzer finds a **counterexample**.
Verification outcome

Example 4. People who do not follow Alice.

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

Once the code is run, Alloy Analyzer finds a counterexample.

UniverseAlpha has 3 elements: Superparticle0, Superparticle1, and Superparticle2 (a.k.a Alice).

Superparticle0 is the only person not following Alice so it is the only person in the output.
Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Verification outcome

Once the code is run, Alloy Analyzer finds a **counterexample**.

UniverseBeta has only 2 elements:
- Superparticle1
- Superparticle2 (or Alice).

Both are following Alice so the result is **empty**.
**Verification outcome**

Example 4. People who do not follow Alice.

\[ Q_{\text{not following Alice}} = \{ x \mid \neg \text{Follows}(x, 'Alice') \} \]

Once the code is run, Alloy Analyzer finds a **counterexample**.

Therefore, this query is **unsafe (domain-dependent)**.
3.2

TRANSLATION TO ALLOY MODEL

Example 6. People who follows everyone.

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

We demonstrate how this verification process can help us debug unsafe queries with another example.
Fixing the query

Example 6. People who follows everyone.

\( Q_{\text{follows all}} = \{ x \mid \forall y [\text{Follows}(x, y)] \} \)
Fixing the query

Example 6. People who follows everyone.

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

Need to make sure that we only consider idols in the database, i.e., they must have at least one follower.
Fixing the query

Need to make sure that we only consider idols in the database, i.e., they must have at least one follower.

So here is the fixed version of the query.

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

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

Example 6. People who follows everyone.

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

Need to make sure that we only consider idols in the database, i.e., they must have at least one follower.

So here is the fixed version of the query.

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

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

Now let us check if the improved query is indeed safe.
Example 6. People who follows everyone.

\[ Q \text{ follows all } v_2 = \{ x \mid \forall y[\exists z(\text{Follows}(z, y)) \Rightarrow \text{Follows}(x, y)] \} \]
Verification outcome

Example 6. People who follows everyone.

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

Once the code is run, Alloy Analyzer still finds a counterexample.
Verification outcome

Example 6. People who follows everyone.

\[ Q_{\text{follows all} \forall v_2} = \{ x \mid \forall y[\exists z [\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)] \} \]

Once the code is run, Alloy Analyzer still finds a **counterexample**.

By browsing all counterexamples, we found that the table **Follows** is always empty.
Once the code is run, Alloy Analyzer still finds a **counterexample**.

By browsing all counterexamples, we found that the table **Follows** is always empty.

So \( \exists z [ \text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y) \) is vacuously true.

**Example 6. People who follows everyone.**

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

Example 6. People who follows everyone.

$Q_{\text{follows all }} v_2 = \{ x \mid \forall y [ \exists z [ \text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)] \}$

Once the code is run, Alloy Analyzer still finds a **counterexample**.

By browsing all counterexamples, we found that the table **Follows** is always empty.

So $\exists z [\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)$ is vacuously true.

And thus the boolean expression of the set comprehension always holds.
Verification outcome

Example 6. People who follows everyone.

\[ Q_{\text{follows all } v^2} = \{ x \mid \forall y[\exists z[\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)] \} \]

Once the code is run, Alloy Analyzer still finds a counterexample.

By browsing all counterexamples, we found that the table \text{Follows} is always empty.

So \( \exists z[\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y) \) is vacuously true.

And thus the boolean expression of the set comprehension always holds.

We forgot to check that each person in the result must follow at least one person.
Verification outcome

Example 6. People who follows everyone.

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

Once the code is run, Alloy Analyzer still finds a counterexample.

By browsing all counterexamples, we found that the table \text{Follows} is always empty.

So \( \exists z[\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y) \) is vacuously true.

And thus the boolean expression of the set comprehension always holds.

We forgot to check that each person in the result must follow at least one person.

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

\[ Q_{\text{follows all v3}} = \{ x \mid \exists w[\text{Follows}(x, w)] \land \forall y[\exists z[\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)] \} \]
Once the code is fixed

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

Example 6. People who follows everyone.

20 /* Lists all follows who follows every idols */
21 fun query[u: Universe]: set Superparticle {
22     { x: u.Element | all y: u.Element |
23         (some z: u.Element | z -> y in Table.Follows) 
24         implies (x -> y in Table.Follows) }
25 }

20 /* Lists all follows who follows every idols */
21 fun query[u: Universe]: set Superparticle {
22     { x: u.Element |
23         (some w: u.Element | x -> w in Table.Follows) and
24         (all y: u.Element |
25             (some z: u.Element | z -> y in Table.Follows) 
26             implies (x -> y in Table.Follows)) }
27 }
Example 6. People who follows everyone.

\[ Q \text{ follows all}_{v3} = \{ x \mid \exists w[\text{Follows}(x, w)] \land \forall y[\exists z[\text{Follows}(z, y)] \Rightarrow \text{Follows}(x, y)] \} \]

This time, Alloy Analyzer no longer finds a counterexample.
Once the code is fixed

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

Example 6. People who follows everyone.

This time, Alloy Analyzer **no longer** finds a **counterexample**.

Even **bumping up the upper limit** of the number of objects, **no counterexample is found**.
Once the code is fixed

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

20 /* Lists all follows who follows every idols */
21 fun query[u: Universe]: set Superparticle {
22     { x : u.Element | all y: u.Element |
23         ( some z: u.Element | z -> y in Table.Follows) implies (x -> y in Table.Follows) }
24 }

This time, Alloy Analyzer **no longer** finds a **counterexample**.

Even **bumping up the upper limit** of the number of objects, **no counterexample is found**.

41 check queryIsSafe for 4
41 check queryIsSafe for 12

We **might** conclude that this latest version of the query is safe.

Based on the assumption that if a **counterexample exists**, then a **small one exists**.
4

CONCLUSION

What have we done and what is next?
Conclusion

**What we did:** Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.
Conclusion

**What we did:** Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.

**What can we do next:**
Conclusion

**What we did:** Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.

**What can we do next:**

- Automate the translation process by implementing a translator.
Conclusion

What we did: Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.

What can we do next:

- Automate the translation process by implementing a translator.
- Add support for all scalar value comparison operators, to reflect total ordering.
Conclusion

What we did: Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.

What can we do next:

- Automate the translation process by implementing a translator.
- Add support for all scalar value comparison operators, to reflect total ordering.
- Extend the framework to support bounded integer operations.
Conclusion

**What we did:** Establish that we could use Alloy Analyzer to verify if a drc query is safe under a given database schema.

**What can we do next:**

- Automate the translation process by implementing a translator.
- Add support for all scalar value comparison operators, to reflect total ordering.
- Extend the framework to support bounded integer operations.
- Add support for the modeling of functional dependencies in database schema.
References


