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

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

1
INTRODUCTION

Defintion of database systems, domain relational caluculus queries, and query safety.

What is relational database?

PersonalData

Name	BirthYear
‘Alice’	1994
‘Bob’	1995
‘Carol’	1994
‘David’	1993

Friendship

NameA	NameB
‘Alice’	‘Bob’
‘Bob’	‘Carol’
‘Alice’	‘Carol’
‘Carol’	‘David’

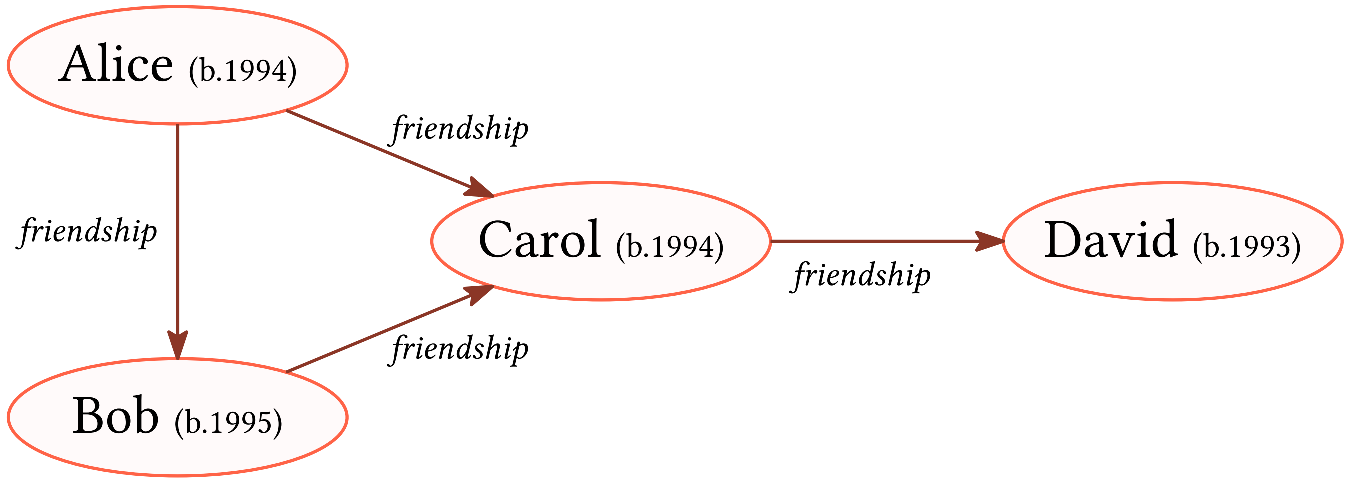
What is relational database?

PersonalData

Name	BirthYear
‘Alice’	1994
‘Bob’	1995
‘Carol’	1994
‘David’	1993

Friendship

NameA	NameB
‘Alice’	‘Bob’
‘Bob’	‘Carol’
‘Alice’	‘Carol’
‘Carol’	‘David’



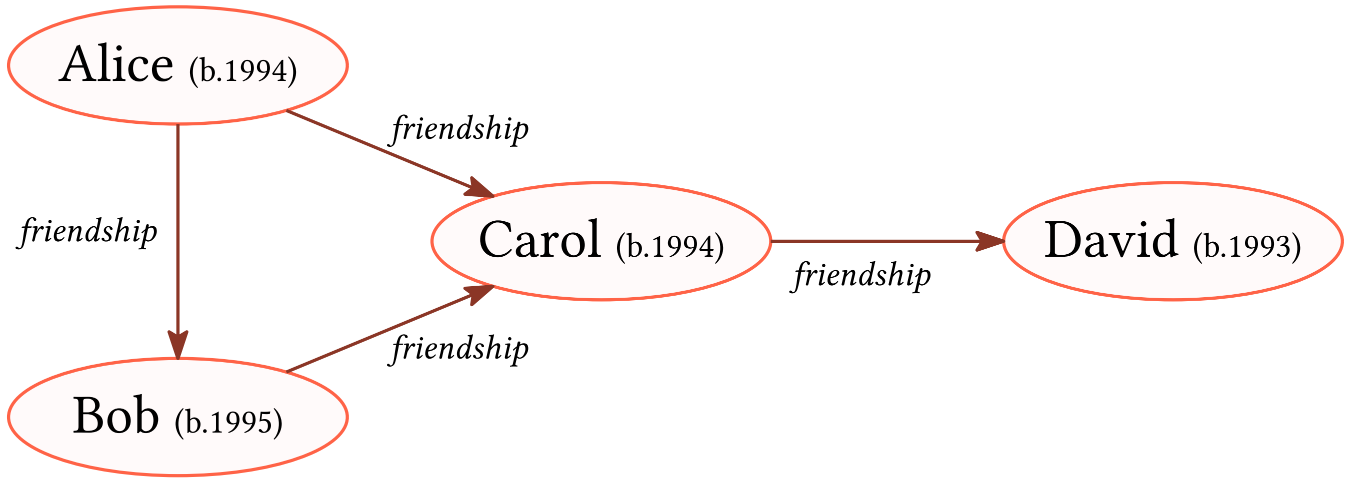
What is relational database?

PersonalData

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

Friendship

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

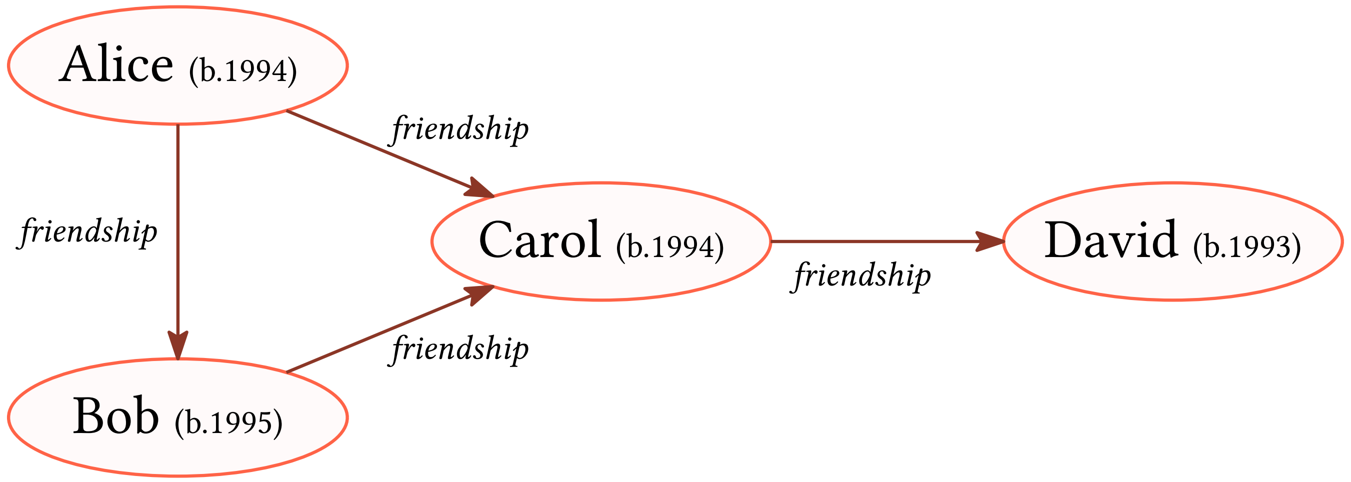


Database: a collection of **tables**.

What is relational database?

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

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



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.*

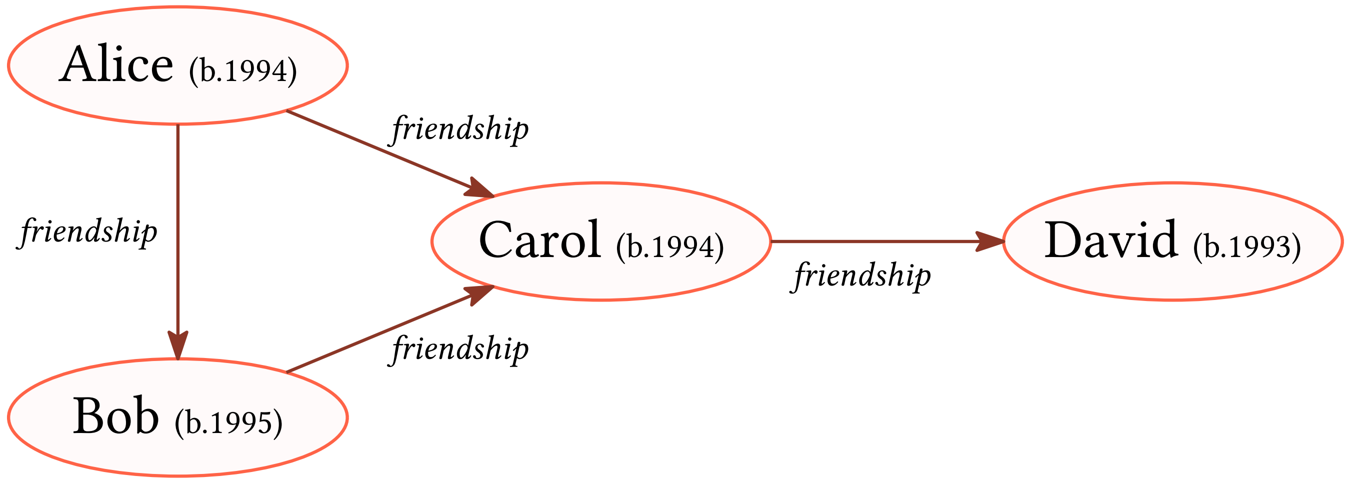
What is relational database?

PersonalData

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

Friendship

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



Database: a collection of **tables**.

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

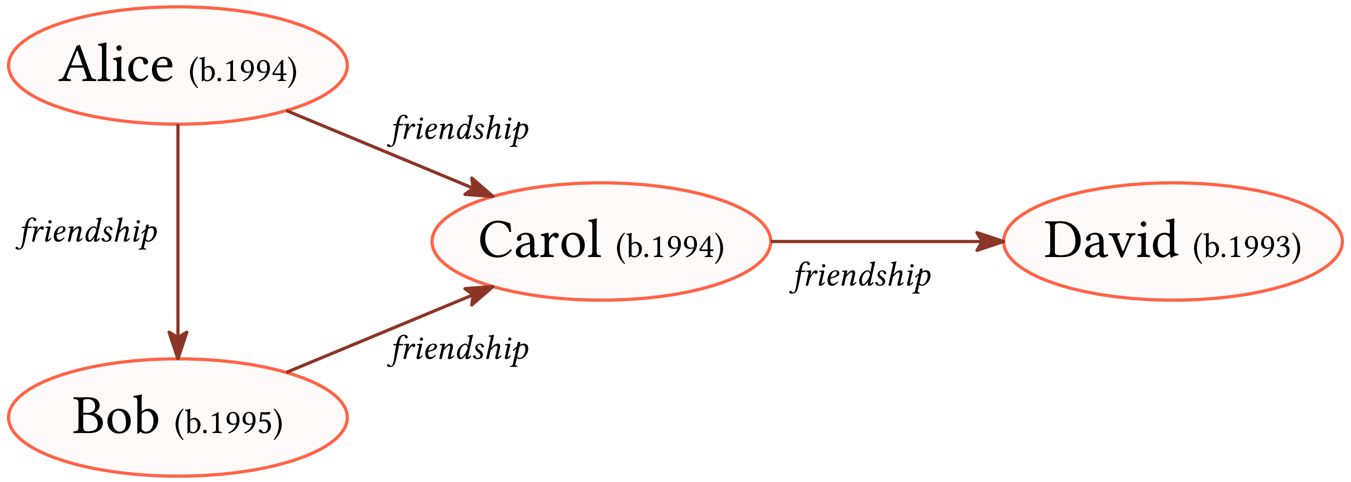
What is relational database?

PersonalData

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

Friendship

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



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**.

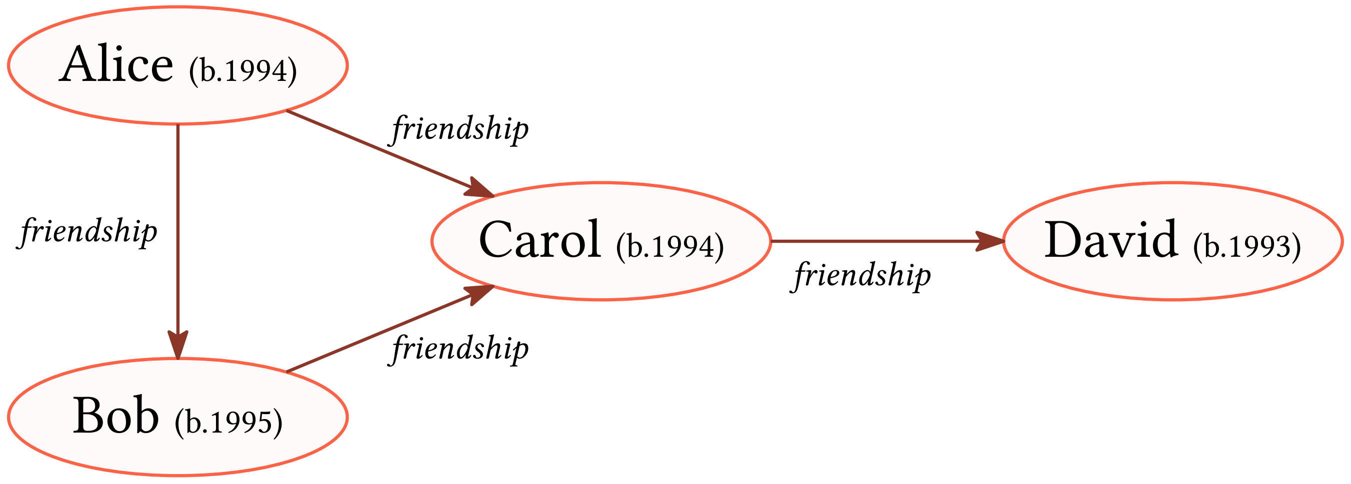
What is relational database?

PersonalData

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

Friendship

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



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**:

```
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
```

PersonalData

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

Query Result

Name	BirthYear
'Alice'	1994
'Carol'	1994
'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**:

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

PersonalData

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

Query Result

Name	BirthYear
'Alice'	1994
'Carol'	1994
'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**:

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

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

PersonalData

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

Query Result

Name	BirthYear
'Alice'	1994
'Carol'	1994
'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**:

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

- 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**:

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

- 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.

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

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

PersonalData

Name	BirthYear
‘Alice’	1994
‘Bob’	1995
‘Carol’	1994
‘David’	1993

For example, **PersonalData**(‘Alice’, 1994) *is true, whereas* **PersonalData**(‘Bob’, 1993) *is false.*

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

- 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.

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

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

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

There are other variant 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.

$$Q_{\text{Bob's friend}} = \{name \mid \textbf{Friendship}(name, \text{`Bob'}) \lor \textbf{Friendship}(\text{`Bob'}, name)\}$$

More examples of DRC queries

Example 2. All friends of Bob.

$$Q_{\text{Bob's friend}} = \{name \mid \textbf{Friendship}(name, \text{`Bob'}) \lor \textbf{Friendship}(\text{`Bob'}, name)\}$$

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

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

More examples of DRC queries

Example 2. All friends of Bob.

$$Q_{\text{Bob's friend}} = \{name \mid \textbf{Friendship}(name, \text{`Bob'}) \lor \textbf{Friendship}(\text{`Bob'}, name)\}$$

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

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

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.

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

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.

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

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.

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

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.

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

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.

COUNTEREXAMPLE

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.

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

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.

COUNTEREXAMPLE

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.

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

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**.

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.

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

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**.

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

$$Q_{\text{Bob's friend}} = \{name \mid \mathbf{Friendship}(name, \text{`Bob'}) \vee \mathbf{Friendship}(\text{`Bob'}, name)\}$$

$$\begin{aligned} Q_{\text{friend of friend}} = \{x, y \mid (x < y) \wedge \exists z[(&\mathbf{Friendship}(x, z) \vee \mathbf{Friendship}(z, x)) \\ &\wedge (\mathbf{Friendship}(y, z) \vee \mathbf{Friendship}(z, y))]\} \end{aligned}$$

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

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

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

$$Q_{\text{Bob's friend}} = \{name \mid \textbf{Friendship}(name, \text{`Bob'}) \lor \textbf{Friendship}(\text{`Bob'}, name)\}$$

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

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

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

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

$$Q_{\text{Bob's friend}} = \{name \mid \mathbf{Friendship}(name, \text{`Bob'}) \vee \mathbf{Friendship}(\text{`Bob'}, name)\}$$

$$Q_{\text{friend of friend}} = \{x, y \mid (x < y) \wedge \exists z[(\mathbf{Friendship}(x, z) \vee \mathbf{Friendship}(z, x)) \wedge (\mathbf{Friendship}(y, z) \vee \mathbf{Friendship}(z, y))]\}$$

UNSAFE

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

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

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

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

$$\begin{aligned} Q_{\text{friend of friend}} = \{x, y \mid (x < y) \wedge \exists z[(&\textbf{Friendship}(x, z) \vee \textbf{Friendship}(z, x)) \\ &\wedge (\textbf{Friendship}(y, z) \vee \textbf{Friendship}(z, y))]\} \end{aligned}$$

UNSAFE

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

... 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.

We have the database table **Follows**(fan, idol) *representing the fact that fan is following idol on a social network.*

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

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.*

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

Suppose that D_1, D_2 are **distinct domain sets** such that $D_2 = D_1 \cup \{c\}$ where **Follows**(c, ‘Alice’) is FALSE. Then,

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.*

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

Suppose that D_1, D_2 are **distinct domain sets** such that $D_2 = D_1 \cup \{c\}$ where $\textbf{Follows}(c, \text{`}\texttt{Alice}\text{'})$ 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.

We have the database table **Follows**(fan, idol) *representing the fact that fan is following idol on a social network.*

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

Suppose that D_1, D_2 are **distinct domain sets** such that $D_2 = D_1 \cup \{c\}$ where **Follows**$(c, \text{‘}\texttt{Alice}\text{’})$ 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.

We have the database table **Follows**(fan, idol) *representing the fact that fan is following idol on a social network.*

UNSAFE $$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

Suppose that D_1, D_2 are **distinct** **domain sets** such that $D_2 = D_1 \cup \{c\}$ where **Follows**$(c, \text{‘}\texttt{Alice}\text{’})$ is FALSE. Then,

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

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.

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

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.

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

As long as there is a person y 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.

UNSAFE

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

Example 6. People who follows everyone.

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

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.

UNSAFE

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

Example 6. People who follows everyone.

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

If the result is not empty under some particular domain, then **adding an alien** to the domain will make the result empty.

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.

UNSAFE

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

Example 6. People who follows everyone.

UNSAFE

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

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

Main problem

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

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

Main problem

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

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)

Main problem

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

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 $Q[D_2]$).

Main problem

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

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.

Main problem

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

$$Q = \{x_1, x_2, \ldots, x_m \mid \underbrace{P(x_1, x_2, \ldots, x_m)}_{\text{boolean expression}}\}$$

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

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.

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.

Model signature definitions

Assumed facts or properties

$\overset{?}{\Longrightarrow}$

Target asserted property

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

$\overset{?}{\Longrightarrow}$

Target asserted property

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.

Model signature definitions

Assumed facts or properties

$\overset{?}{\Longrightarrow}$

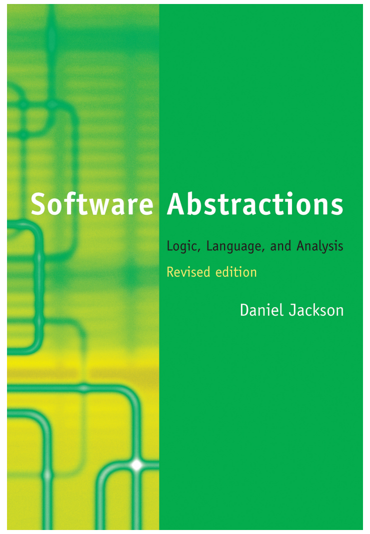
Target asserted property

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.**

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

`http://alloy.mit.edu/`



Task summary

Coming up next . . .

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

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**.

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.

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,

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

- model signatures for **domain sets** 1, **scalar values**, and optional **query result** 5
- and **safety assertion statement** 4 for the query.

3.1

TRANSLATION TO ALLOY MODEL

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

We demonstrate how to translate database schema and DRC *queries into Alloy syntax with an example.*

① Domain sets and scalar values

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\mathtt{Alice}\text{’})\}$$

① Domain sets and scalar values

Example 4. People who do not follow Alice.

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

```
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**.

① Domain sets and scalar values

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

```
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
}
```

- a set of all possible scalar values across all domains
- a collection of **exactly** two domain sets
  - 1st domain
  - 2nd domain
- a set of scalar values allowed in the actual database instances

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**.

① Domain sets and scalar values

Example 4. People who do not follow Alice.

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

```
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
}
```

fact: each `Superparticle` must belong to at least one universe

the field of `Universe` representing the subset of `Superparticle`

fact: `Particle` is the intersection of both universes

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**.

② Database instances

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

② Database instances

```
one sig Table {
    Follows: Particle -> Particle
}
```

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.

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

```
one sig Table {
    Follows: Particle -> Particle
}
```

definition of the table **Follows**

with 2 columns

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.

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

② Database instances

```
one sig Table {
    Follows: Particle -> Particle,
    User: set Particle    /* assume we have another table */
}
```

HYPOTHETICAL!

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.

② Database instances

Example 4. People who do not follow Alice.

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

```
one sig Table {
    Follows: Particle -> Particle,
    User: set Particle     /* assume we have another table */
}
```

HYPOTHETICAL!

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.

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

```
one sig Table {
    Follows: Particle -> Particle
}
```

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 `->`.

③ Query function

Example 4. People who do not follow Alice.

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

③ Query function

Example 4. People who do not follow Alice.

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

```
one sig Constant {
    Alice: Particle
}

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

the input to the `query` function in Alloy is the domain set

③ Query function

Example 4. People who do not follow Alice.

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

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

```
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

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

③ Query function

Example 4. People who do not follow Alice.

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

```
one sig Constant {
    Alice: Particle
}

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

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

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

③ Query function

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

```
one sig Constant {
    Alice: Particle
}

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

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

- For a **conjunction** ($\wedge$), a **disjunction** ($\vee$), a **negation** ($\neg$), a **conditional** ($\Longrightarrow$), a **bi-conditional** ($\Longleftrightarrow$), 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 (`->`), and the set member operation (`in`) checks if the tuple belongs to the specified table.

```
TranslateBooleanExp(P):
if P is a table-name predicate T(x₁, x₂, ..., xₘ):
    return "${x₁} → ${x₂} → ... → ${xₘ} in Table.${T}"
else if P is the equality predicate x₁ = x₂:
    return "(${x₁} = ${x₂})"
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)})"
```

④ Safety verification for query

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\mathtt{Alice}\text{’})\}$$

4 Safety verification for query

Example 4. People who do not follow Alice.

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

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

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

④ Safety verification for query

Example 4. People who do not follow Alice.

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

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

upper limit of number of objects

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.

④ Safety verification for query

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

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

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.

5 Optional results placeholder

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\mathtt{Alice}\text{’})\}$$

⑤ Optional results placeholder

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

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

5 # Optional results placeholder

Example 4. People who do not follow Alice.

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

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

output signature of the result

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

- Except for the **signature fo the** `Output` **field** of the query `Result` object, which will be **exactly the same** as the output signature of the Alloy function `query`. 3

⑤ Optional results placeholder

Example 4. People who do not follow Alice.

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

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

fact: the output for each case of a domain set is binded to the result of the query under that domain

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

- Except for the **signature fo the** `Output` **field** of the query `Result` object, which will be **exactly the same** as the output signature of the Alloy function `query`. ③

The output is binded to the query result when the domain is applied.

Summarized Alloy code

Example 4. People who do not follow Alice.

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

```
/* 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
```

Verification outcome

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\, \mathbf{Follows}(x, \text{‘}\mathtt{Alice}\text{’})\}$$

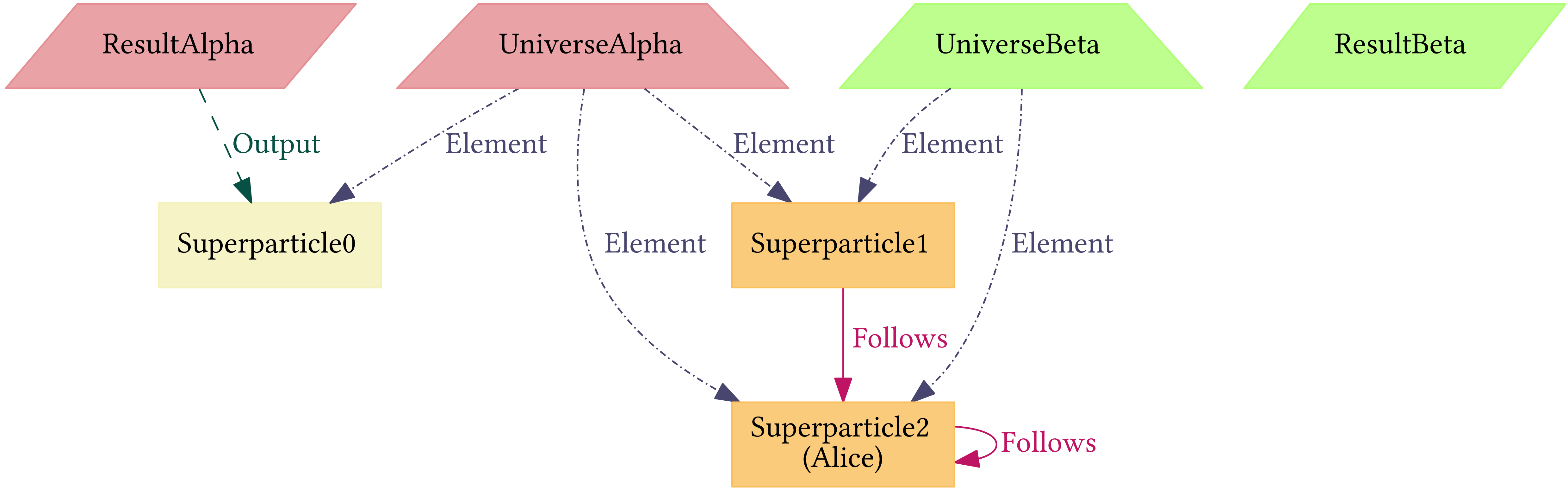
Once the code is run, Alloy Analyzer finds a **counterexample**.

Verification outcome

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\,\textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

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

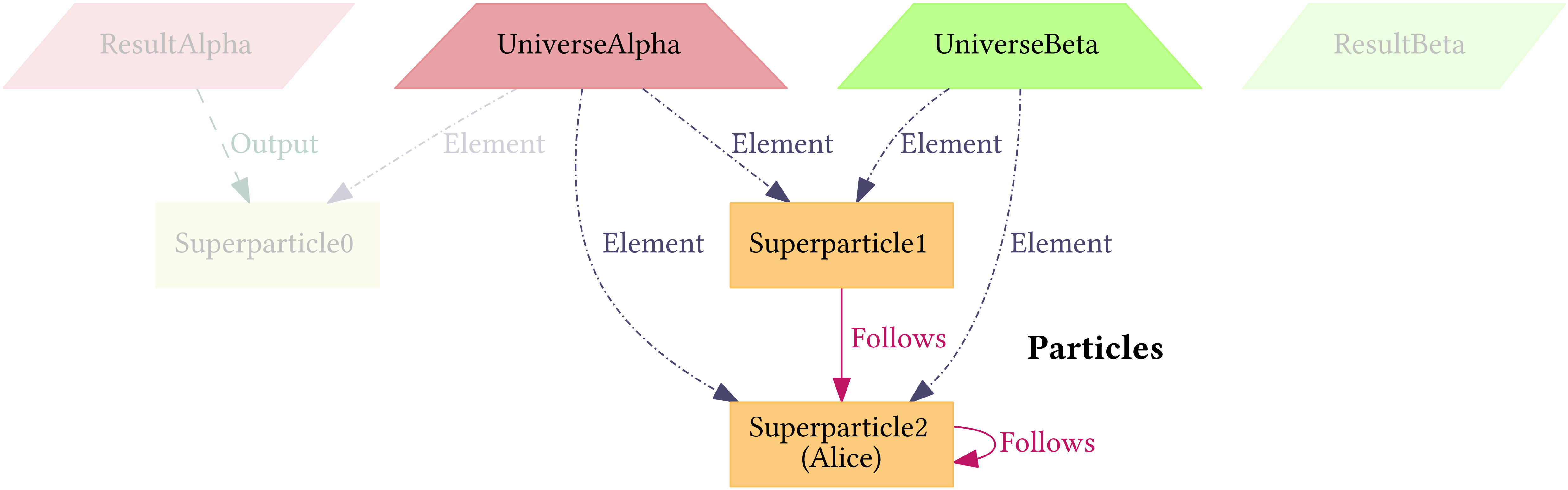


Verification outcome

Example 4. People who do not follow Alice.

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

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

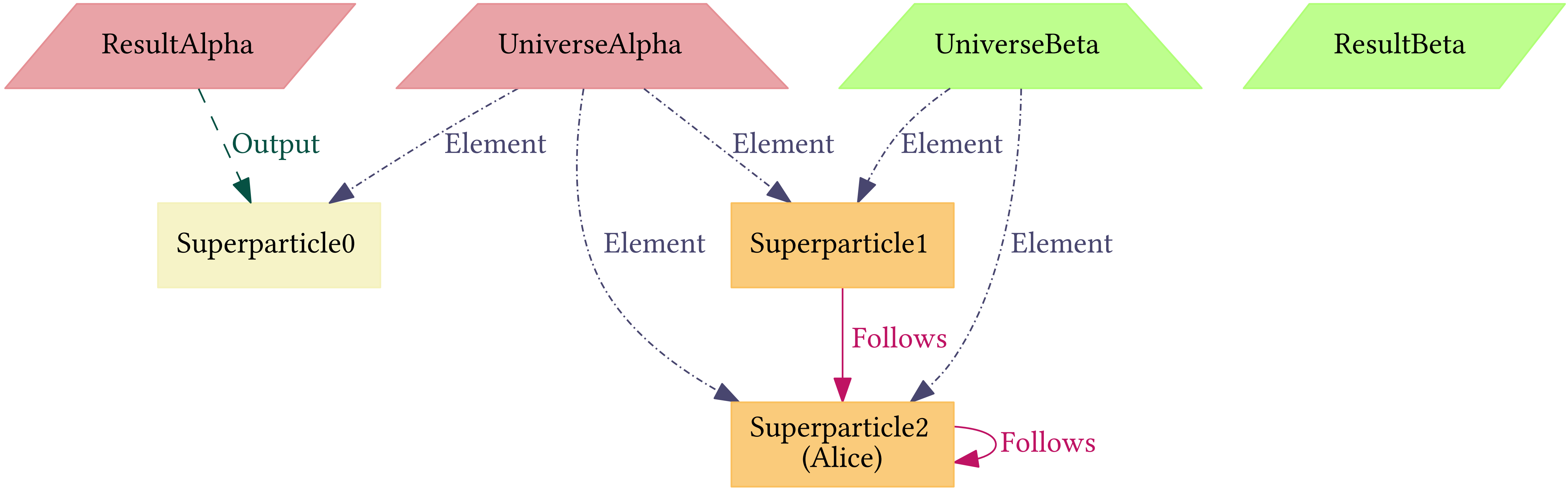


Verification outcome

Example 4. People who do not follow Alice.

$$Q_{\text{not following Alice}} = \{x \mid \neg\,\textbf{Follows}(x, \text{‘}\texttt{Alice}\text{’})\}$$

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

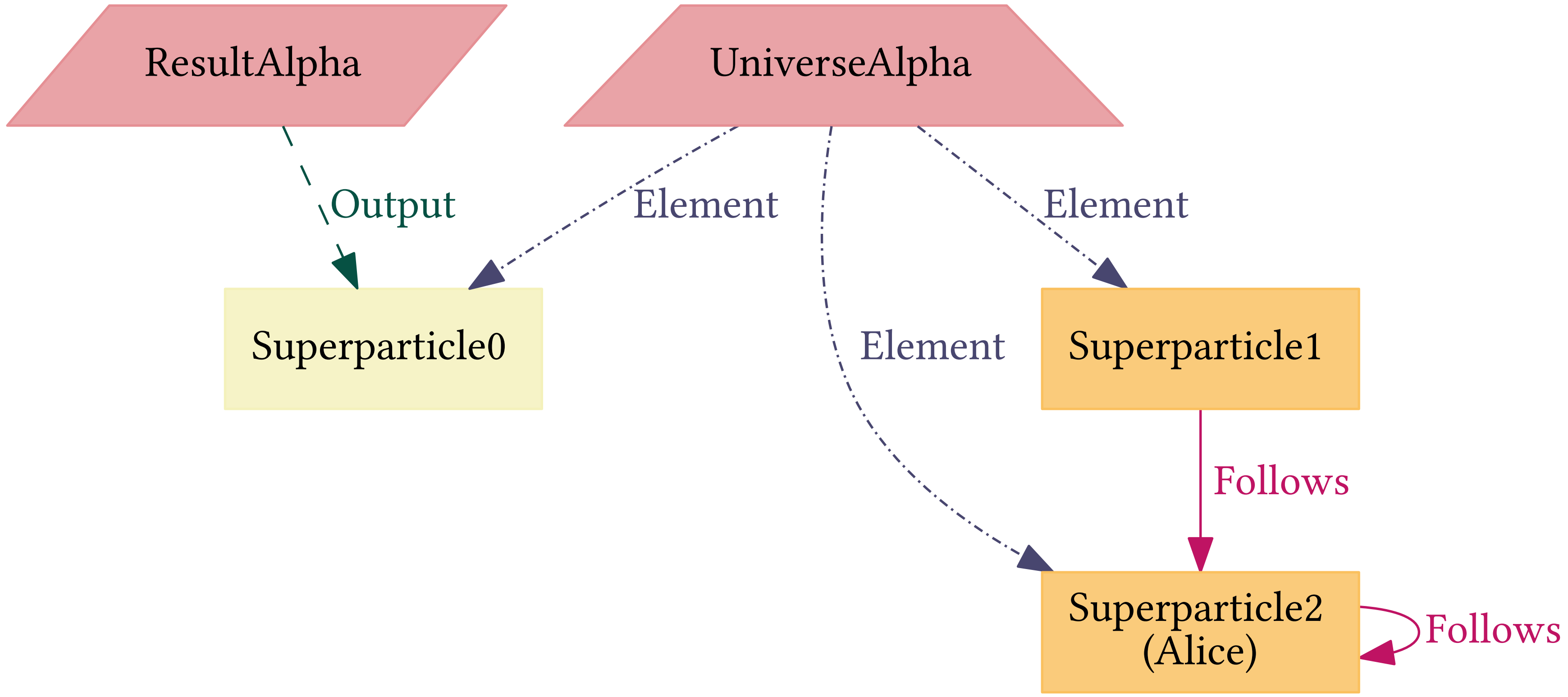


Verification outcome

Example 4. People who do not follow Alice.

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

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.

Verification outcome

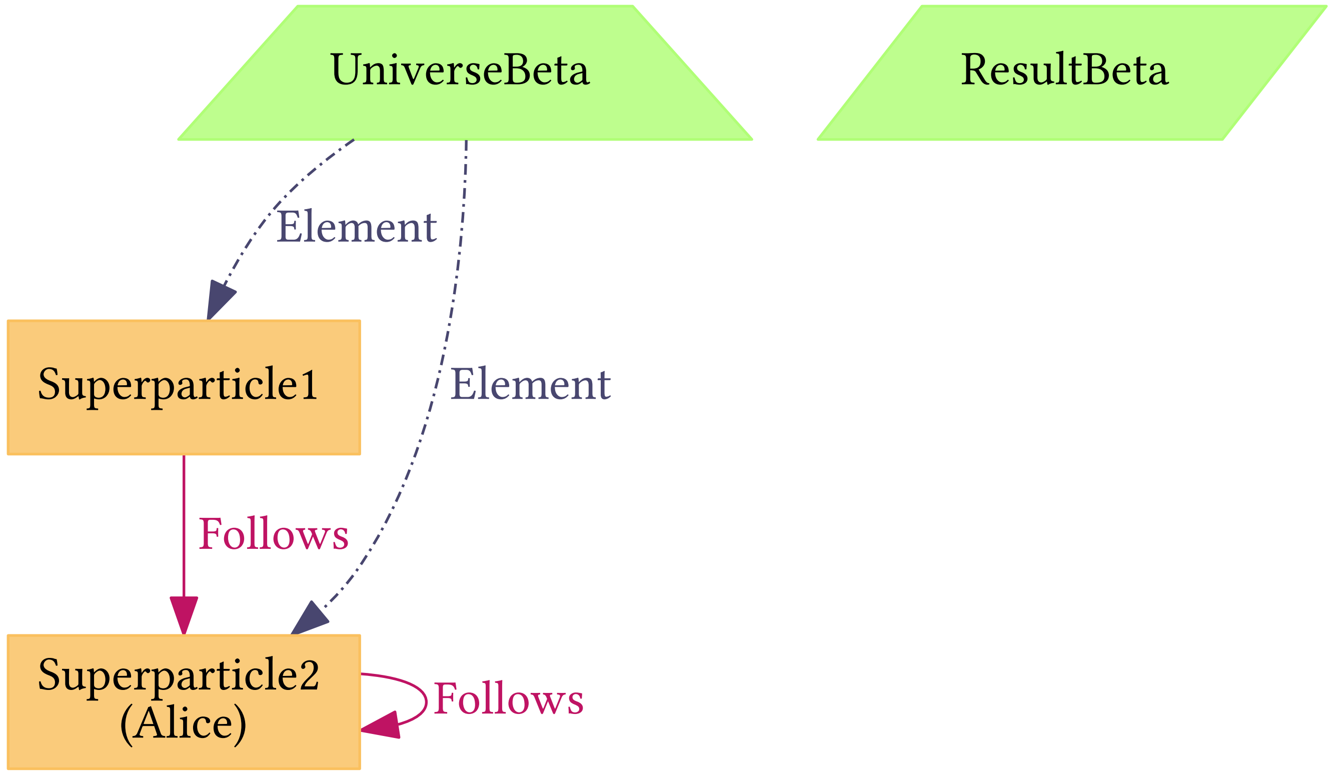
Example 4. People who do not follow Alice.

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

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

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

Both are following `Alice` so the result is **empty**.

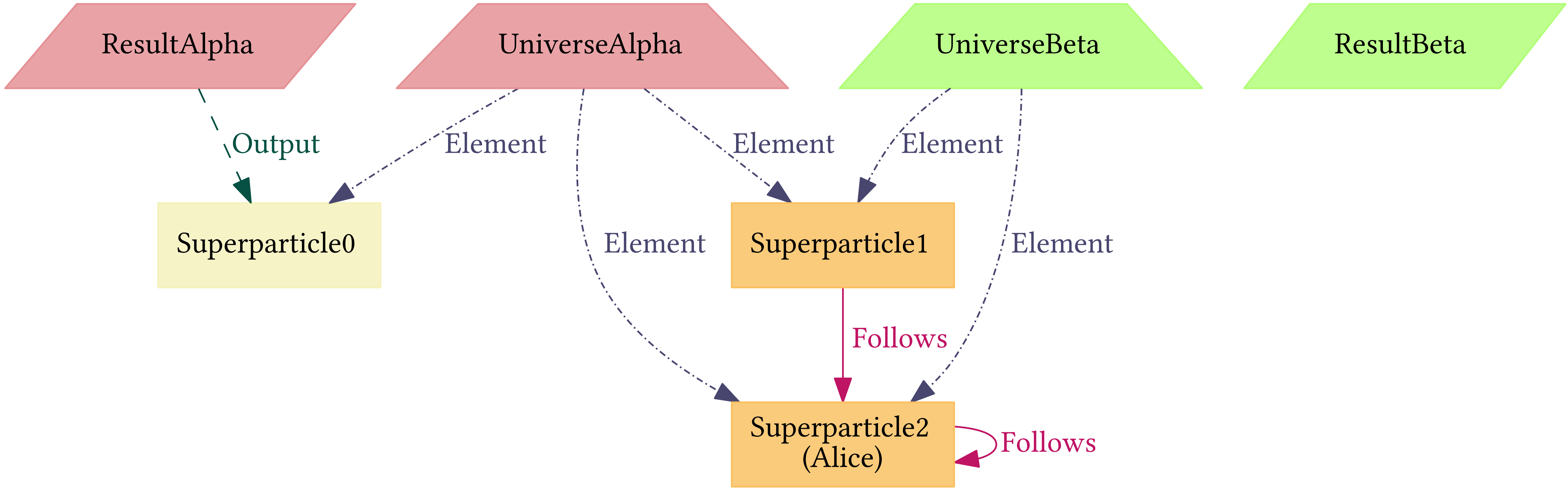


Verification outcome

Example 4. People who do not follow Alice.

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

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.

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

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

Fixing the query

Example 6. People who follows everyone.

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

Fixing the query

Example 6. People who follows everyone.

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

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

Fixing the query

Example 6. People who follows everyone.

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

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.

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

$$\downarrow$$

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

Fixing the query

Example 6. People who follows everyone.

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

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.

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

$$\downarrow$$

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

Now let us check if the improved query is indeed **safe**.

Summarized Alloy code

Example 6. People who follows everyone.

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

```
/* 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
```

Verification outcome

Example 6. People who follows everyone.

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

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

Verification outcome

Example 6. People who follows everyone.

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

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

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

Verification outcome

Example 6. People who follows everyone.

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

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

Verification outcome

Example 6. People who follows everyone.

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

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

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

Verification outcome

Example 6. People who follows everyone.

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

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 $\exists z[\mathbf{Follows}(z, y)] \Rightarrow \mathbf{Follows}(x, y)$ 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

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 $\exists z[\textbf{Follows}(z, y)] \Rightarrow \textbf{Follows}(x, y)$ 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**.

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

$$\downarrow$$

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

Once the code is fixed

Example 6. People who follows everyone.

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

```
/* 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) }
}
```

→

```
/* Lists all follows who follows every idols */
fun query[u: Universe]: set Superparticle {
    { 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)) }
}
```

Once the code is fixed

Example 6. People who follows everyone.

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

```
/* 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) }
}
```

→

```
/* Lists all follows who follows every idols */
fun query[u: Universe]: set Superparticle {
    { 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)) }
}
```

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

Once the code is fixed

Example 6. People who follows everyone.

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

```
/* 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) }
}
```

```
/* Lists all follows who follows every idols */
fun query[u: Universe]: set Superparticle {
    { 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)) }
}
```

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

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

```
check queryIsSafe for 4
```

```
check queryIsSafe for 12
```

Once the code is fixed

Example 6. People who follows everyone.

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

```
/* 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) }
}
```

```
/* Lists all follows who follows every idols */
fun query[u: Universe]: set Superparticle {
    { 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)) }
}
```

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

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

```
check queryIsSafe for 4
```

```
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 verity if a drc query is safe under a given database schema.

Conclusion

What we did: Establish that we could use Alloy Analyzer to verity 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 verity 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 verity 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 verity 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 verity 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

[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.