Automated constraint verification for databases

Shuai Yuan
advised by Véronique Benzaken, Evelyne Contejean and Mingli Song

College of Computer Science and Technology,
Zhejiang University

Outline

- Background
- Context
- Proposed solutions
- Database integrity constraints
- SQL data modification operations
- Why3
- Translation overiew
- Problem and solutions
- Experiments
- Conclusion and perspectives

Background

- Integrity constraints are important to express the semantics of databases.
  - prevent the execution of operations or transactions which will cause violation of constraints
  - semantics query optimization
- No real DBMS (database management system) have fully support the management of integrity constraints.
  i.e. assertions which are part of the SQL standard
- The alternative solution widely adopted by main-stream DBMS – triggers:

Background

- Integrity constraints are important to express the semantics of databases.
  - prevent the execution of operations or transactions which will cause violation of constraints
  - semantics query optimization
- No real DBMS (database management system) have fully support the management of integrity constraints.
  i.e. assertions which are part of the SQL standard
- The alternative solution widely adopted by main-stream DBMS – triggers:

Background

- Integrity constraints are important to express the semantics of databases.
  - prevent the execution of operations or transactions which will cause violation of constraints
  - semantics query optimization
- No real DBMS (database management system) have fully support the management of integrity constraints.
  i.e. assertions which are part of the SQL standard
- The alternative solution widely adopted by main-stream DBMS – triggers:
  - event
  - condition
  - action

Background

- Integrity constraints are important to express the semantics of databases.
  - prevent the execution of operations or transactions which will cause violation of constraints
  - semantics query optimization
- No real DBMS (database management system) have fully support the management of integrity constraints.
  i.e. assertions which are part of the SQL standard
- The alternative solution widely adopted by main-stream DBMS – triggers:
  - Integrity is spread out among several triggers and therefore the global vision of the semantics is lost.
  - The behavior of triggers is complex (cascading, conflict, *etc.*).

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
- data modification operation U.

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
- data modification operation U.

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
- data modification operation U.

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
  1. domain constraint
  2. column constraint
  3. table constraint
  4. assertion
- data modification operation U.
  1. INSERT
  2. DELETE
  3. UPDATE

We focus on relational databases and SQL.

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
- data modification operation U.

Before U is executed, $\mathscr{B} \models C$
What we want to prove: After U is executed, $U(\mathscr{B}) \models C$.

U preserves C, or U is safe with respect to C,
if for any database $\mathscr{B}$:

$$\mathscr{B} \models C \Rightarrow U(\mathscr{B}) \models C$$

Context

We have:

- database $\mathscr{B}$.
- database integrity constraint C.
- data modification operation U.

- violation detection: check after the execution of data modification operations.
  violation occurs $\rightarrow$ rollback
  efficiency problem (check at run-time, rollback, etc.)
- violation prevention: check before the execution of data modification operations.
  violation occurs $\rightarrow$ abort

Proposed solutions

Our method

- violation prevention
- mainly based on weakest precondition approach

Proposed solutions

Hoare triple:

- precondition P
- command C
- postcondition Q

$$\{P\} \;\; C \;\; \{Q\}$$

When the precondition is met, the command establishes the postcondition.

Proposed solutions

In database:
Let f, g be logical formulae, U be a data modification operation, then g is the precondition of U and f if for any database $\mathscr{B}$:

$$\mathscr{B} \models g \;\Rightarrow\; U(\mathscr{B}) \models f$$

Precondition $wpc(f, U)$ is the weakest precondition of f and U if for any database $\mathscr{B}$ and any precondition g:

$$\mathscr{B} \models g \;\Rightarrow\; \mathscr{B} \models wpc(f, U)$$

Proposed solutions

Now what we need to prove:

$$C \Rightarrow wpc(C, U)$$

$$\left.\begin{array}{l} C \Rightarrow wpc(C, U) \\ \mathscr{B} \models C \end{array}\right\} \Rightarrow \mathscr{B} \models wpc(C, U)$$

$$\left.\begin{array}{r} \mathscr{B} \models wpc(C, U) \Rightarrow U(\mathscr{B}) \models C \end{array}\right\} \Rightarrow U(\mathscr{B}) \models C$$

Proposed solutions

Firstly, we need to formalize:

- database integrity constraint C
- data modification operation U

Database integrity constraints

database integrity constraint in SQL

1. domain constraint
2. column constraint
3. table constraint
4. assertion

All the SQL integrity constraints can be reduced to logically and semantically equivalent assertions.
Assertions alone are sufficient to expressing any kind of SQL integrity constraints.

Therefore, we reduce database integrity constraints into SQL assertions:

CREATE ASSERTION <*assertion name*>
CHECK <exists predicate>

<exists predicate>	::=	[NOT] EXISTS (<query expression>)
<query expression>	::=	SELECT *
		FROM r_1 x_1, $\cdots$, r_n x_n
		WHERE <search condition>
<search condition>	::=	<boolean term>
	|	<search condition> OR <boolean term>
<boolean term>	::=	<boolean factor>
	|	<boolean term> AND <boolean factor>
<boolean factor>	::=	<predicate>
	|	[NOT] (<search condition>)

Therefore, we reduce database integrity constraints into SQL assertions:

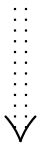
<predicate>	::=	<exists predicate>
	|	<comparison predicate>
	|	<between predicate>
	|	<in predicate>
<comparison predicate>	::=	<expression$_1$ > <comp op> <expression$_2$ >
<comp op>	::=	= | <> | < | ≤ | > | ≥
<expression>	::=	<term>
	|	<expression> {+ | −} <term>
<term>	::=	<factor>
	|	<term> {∗ | /} <factor>
<factor>	::=	(<expression>)
	|	[+ | −] <*constant*>
	|	[+ | −] *x.a*
<between predicate>	::=	<expression> [NOT] BETWEEN <*constant*$_1$ > AND <*constant*$_2$ >
<in predicate>	::=	<expression> [NOT] IN (<in value list>)
<in value list>	::=	<*constant*>
	|	<in value list>, <*constant*>

Database integrity constraint

database integrity constraints

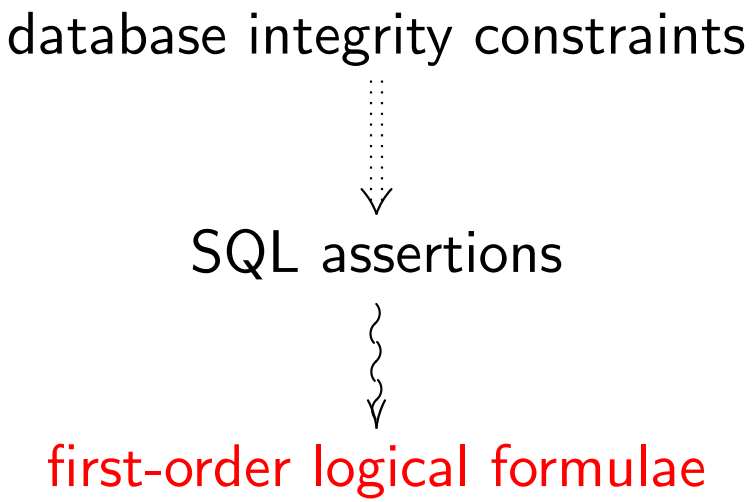
Database integrity constraint

database integrity constraints



SQL assertions

Database integrity constraint



Database integrity constraint

A logical formula, the target language we define, can contain:

- constants (0, 1, . . . , etc.)
- relation variables (r, etc.), tuple variables (x, etc.) and attribute variables (a, etc.)
- numerical symbols ($+$, -, *, $/$, . . . , etc.)
- comparison symbols ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$, . . . , etc.) and set operators ($\in$)
- logical connective symbols ($\wedge$, $\vee$, $\neg$)
- quantifiers ($\forall$, $\exists$)

Database integrity constraint

Terms of a logical formula are defined as follows:

- All constants and variables are terms.
- If x is a tuple (variable) and a is an attribute of x, then $x.a$ is a term.
- If p is a term and q is a term, then "p <numerical symbol> q" is a term.

Database integrity constraint

A logical formula is constructed according to the following rules:

- If x is a tuple and r is a relation variable, then "$x \in r$" is a formula.
- If p is a term and q is a term, then "p <comparison symbol> q" is a formula.
- If ϕ and φ are formulae, then $\phi \wedge \varphi$, $\phi \vee \varphi$, $\neg\phi$ are formulae.
- If ϕ is a formula and x is a tuple variable, then $\forall x, \phi$ and $\exists x, \phi$ are formulae.

Translate SQL assertions into FOL formulae

$\mathscr{T}$: function mapping a SQL assertion phrase into a first-order logical formula.

A simple example:

$$\mathscr{T}[\ \text{CREATE ASSERTION example}\ \text{CHECK}\ (\ \text{NOT EXISTS}\ (\ \text{SELECT}\ *\ \text{FROM}\ r\ x\ \text{WHERE}\ x.a = 1))]$$

Translate SQL assertions into FOL formulae

$\mathscr{T}$[CREATE ASSERTION example
CHECK (NOT EXISTS (SELECT $*$
FROM $r\ x$
WHERE $x.a = 1$))]

example: $\mathscr{T}$[NOT EXISTS (SELECT $*$
FROM $r\ x$
WHERE $x.a = 1$))]

$\rightsquigarrow$

Translate SQL assertions into FOL formulae

$$\mathscr{T}[\ \text{CREATE ASSERTION example CHECK (NOT EXISTS (SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1))]$$

example: $\mathscr{T}[\text{NOT EXISTS (SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1))]$

$\rightsquigarrow$

example: $\neg\exists(\mathscr{T}[\text{SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1])$

$\rightsquigarrow$

Translate SQL assertions into FOL formulae

$$\mathscr{T}[\ \text{CREATE ASSERTION example CHECK (NOT EXISTS (SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1))]$$

$$\text{example: } \mathscr{T}[\text{NOT EXISTS (SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1))]$$

$$\rightsquigarrow$$

$$\text{example: } \neg\exists(\mathscr{T}[\text{SELECT } * \text{ FROM } r\ x \text{ WHERE } x.a = 1])$$

$$\rightsquigarrow$$

$$\text{example: } \neg\exists(x \in r; \mathscr{T}[x.a = 1])$$

$$\rightsquigarrow$$

Translate SQL assertions into FOL formulae

$\mathscr{T}[$ CREATE ASSERTION example
CHECK (NOT EXISTS (SELECT $*$
FROM $r\ x$
WHERE $x.a = 1))]$

example: $\mathscr{T}[$NOT EXISTS (SELECT $*$
$\rightsquigarrow$ FROM $r\ x$
WHERE $x.a = 1))]$

example: $\neg\exists(\mathscr{T}[$SELECT $*$ FROM $r\ x$
$\rightsquigarrow$ WHERE $x.a = 1])$

$\rightsquigarrow$ example: $\neg\exists(x \in r; \mathscr{T}[x.a = 1])$

$\rightsquigarrow$ example: $\neg\exists(x \in r; \mathscr{T}[x.a]\mathscr{T}[=]\mathscr{T}[1])$

Translate SQL assertions into FOL formulae

$\mathscr{T}[$ CREATE ASSERTION example
CHECK (NOT EXISTS (SELECT $*$
FROM $r\ x$
WHERE $x.a = 1))]$

example: $\mathscr{T}[$NOT EXISTS (SELECT $*$
$\rightsquigarrow$ FROM $r\ x$
WHERE $x.a = 1))]$

example: $\neg\exists(\mathscr{T}[$SELECT $*$ FROM $r\ x$
$\rightsquigarrow$ WHERE $x.a = 1])$

$\rightsquigarrow$ example: $\neg\exists(x \in r; \mathscr{T}[x.a = 1])$

$\rightsquigarrow$ example: $\neg\exists(x \in r; \mathscr{T}[x.a]\mathscr{T}[=]\mathscr{T}[1])$

$\rightsquigarrow$ example: $\neg\exists(x \in r; x.a = 1)$

Semantics of SQL data modification operations

SQL INSERT

$$\begin{aligned} \text{INSERT INTO} \quad & r \\ \text{VALUES} \quad & t \end{aligned}$$

Semantics:

$$(x \in r \vee x = t) \Leftrightarrow x \in U(r)$$

Semantics of SQL data modification operations

SQL DELETE

DELETE FROM r
WHERE <search condition>

Let g be the logical formula of <search condition>.
Semantics:

$$(x \in r \wedge \neg g(x)) \Leftrightarrow x \in U(r)$$

Semantics of SQL data modification operations

SQL UPDATE

UPDATE r
SET <set clause>
WHERE <search condition>

Let σ be the assignment function defined by <set clause> and g be the logical formula of <search condition>.
Semantics:

$$\forall x \in r : \begin{cases} g(x) \Rightarrow U(x) = \sigma(x) \\ \neg g(x) \Rightarrow U(x) = x \end{cases}$$

Why3

Why3 is a set of tools for program verification which uses first-order logic.

input: programs

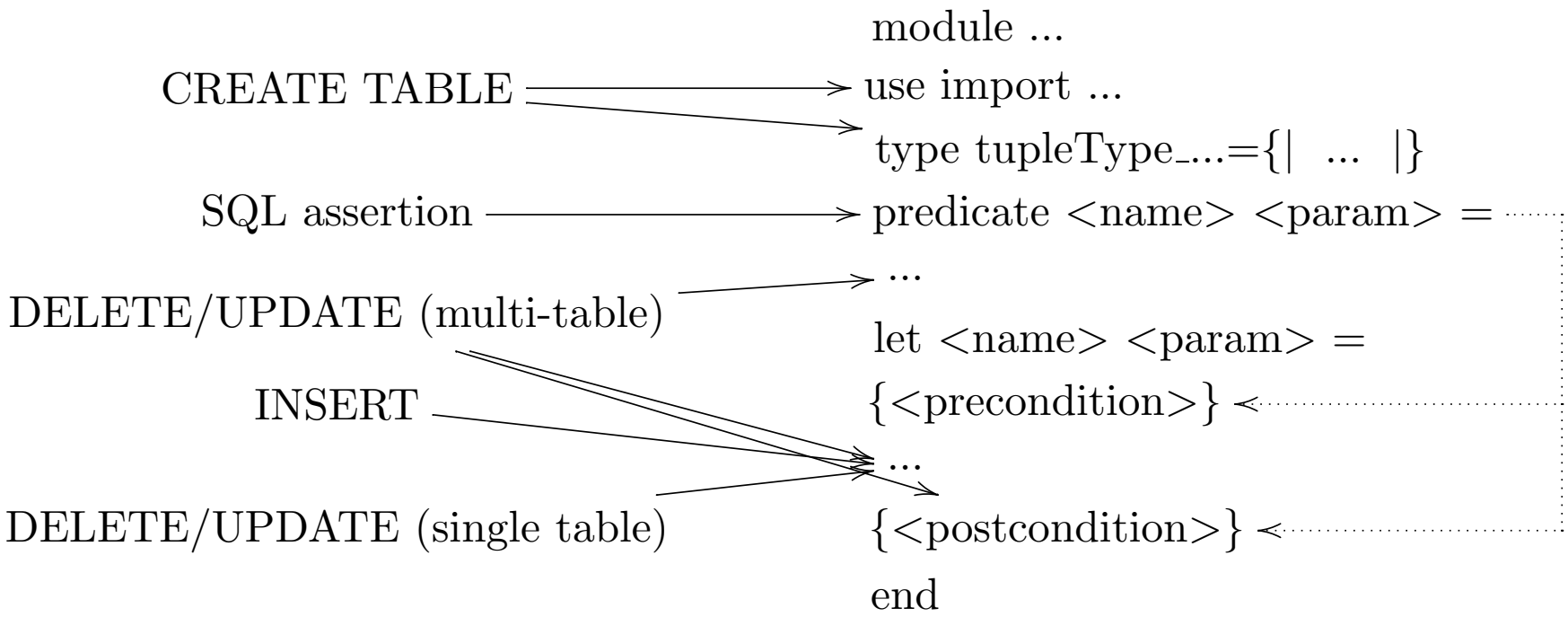
output: logical declarations + goals, in the syntax of the selected prover

Why3

Given a constraint C and a date modification operation U:

- Translate SQL statements into WhyML programs.
- Use Why3 to generate the weakest precondition $wpc(C, U)$.
- Call the provers to prove $C \Rightarrow wpc(C, U)$, if it is proven, then $U(\mathscr{B}) \models C$.

Translation Overview



Problem and Solution

Our former experiments show that our methods cannot detect some safe date modification operators when the integrity constraint C is in the form of:

$$\exists x \in R, f(x)$$

Before U is executed:

$$r \models C(r) \equiv \exists x \in r, f(x)$$

What we want to proof:

$$U(r) \models C(U(r)) \equiv \exists x \in U(r), f(x)$$

Problem and Solution

We adopt the predicate transformer to improve it:

e.g. backward predicate transformer $\overleftarrow{U}$

$$\mathscr{B} \models \overleftarrow{U}(C) \Rightarrow U(\mathscr{B}) \models C$$

Use this method, our problem becomes: $C \Rightarrow \overleftarrow{U}(C)$

Problem and Solution

For the constraints in the following form:

$$\exists x \in R, f(x)$$

We find precise predicate transformer C':

$$\mathscr{B} \models C' \Leftrightarrow U(\mathscr{B}) \models C$$

Problem and Solution

Theorem (INSERT)

The execution of any SQL INSERT statements will not affect the constraint C.

Theorem (DELETE)

$U(r) \models C \Leftrightarrow \exists x \in r, f(x) \wedge \neg g(x)$

Theorem (UPDATE)

$U(r) \models C \Leftrightarrow \exists x \in r, (f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x)))$

Problem and Solution

Similar results can be derived when the constraint C is in the form of:

$$\forall x \in R, f(x)$$

Before U is executed:

$$r \models C(r) \equiv \forall x \in r, f(x)$$

What we want to proof:

$$U(r) \models C(U(r)) \equiv \forall x \in U(r), f(x)$$

Problem and Solution

Theorem (INSERT)

$U(r) \models C \Leftrightarrow f(t)$

Theorem (DELETE)

The execution of any SQL DELETE statements will not affect the constraint C.

Theorem (UPDATE)

$U(r) \models C \Leftrightarrow \forall x \in r, \neg g(x) \vee (g(x) \wedge f(\sigma(x)))$

Experiments

Provers

Alt-Ergo, CVC3, Yices and Gappa.

Unsafe data modification operations can be detected correctly.
For safe data modification operations:

	INSERT	DELETE	UPDATE
least mean time (s)	0.205	2.338	1.149

Experiments

When the integrity constraint C is in the form of:

$$\exists x \in R, f(x)$$

We try a newer version of Why3 (0.72) and find that the weakest precondition approach can detect safe data modification operations.

	INSERT	DELETE	UPDATE
weakest precondition (s)	0.208	0.119	0.381
precise predicate transformer (s)	0	0.047	0.054

Experiments

When the integrity constraint C is in the form of:

$$\forall x \in R, f(x)$$

	INSERT	DELETE	UPDATE
weakest precondition (s)	0.199	0.16	0.323
precise predicate transformer (s)	0.056	0	0.055

Conclusion & Prospect

Our work:

- rules of reducing SQL database integrity constraints to assertions.
- translating SQL assertions into first-order logical formulae.
- semantics of SQL data modification operations (INSERT/DELETE/UPDATE).
- using weakest precondition and Why3 to implement automated verification of database integrity constraints.
- precise predicate transformer.

Conclusion & Prospect

Future work:

- more complex SQL grammar.
- aggregate functions.