2012-5-12

浙江大學

本科生毕业论文

题目： **数据库约束的自动化验证**

姓　　名：袁帅

学　　号：3080100590

指导教师：宋明黎

专　　业：2008 级　计算机科学与技术

学　　院：计算机学院

A Dissertation Submitted to Zhejiang University for the Degree of Bachelor of Engineering

TITLE: Automated constraints verification for databases

Author: Shuai Yuan 3080100590

Supervisor: Mingli Song

Major: Computer Science and Technology

College: College of Computer Science and Technology

Submitted Date:

浙江大学本科生毕业论文（设计）诚信承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。
2. 本人在毕业论文（设计）中引用他人的观点和参考资料均加以注释和说明。
3. 本人承诺在毕业论文（设计）选题和研究内容过程中没有抄袭他人研究成果和伪造相关数据等行为。
4. 在毕业论文（设计）中对侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

毕业论文（设计）作者签名：

_____ 年 ____ 月 ____ 日

摘　要

数据库完整性约束是表达数据库语义的重要工具，但实际数据库管理系统DBMS却无法完全支持完整性约束的验证。而作为被DBMS广泛采用的替代方案的触发器，一方面数据库的语义分散在一系列触发器中变得难以理解，另一面触发器会引发冲突问题又增加了它们本身行为的复杂性。这使得我们仍然有必要对完整性约束的验证问题进行研究。

在本文中，我们实现了一种数据库完整性约束的自动验证策略。我们的方法基于最弱前置条件和谓词转化器。首先，我们把数据库完整性约束规约到了SQL断言上，并把断言转化为一阶逻辑表达式。相应地，我们也给出了SQL语言的三大数据库修改操作INSERT、DELETE和UPDATE的逻辑语义。在对以上两者所进行的逻辑形式化的基础上，我们利用程序验证平台Why3实现了对数据库完整性约束的验证。对于输入的SQL语句，我们的程序将它们转化为WhyML程序，然后调用Why3来计算最弱前置条件和生成传递给后端的定理自动证明器（如Alt-Ergo、CVC等）的检验条件，由定理自动证明器给出执行完修改操作后的数据库是否仍然满足约束的最终判断。以上整个过程实现了完全的自动化。

此外，对于某些形式的完整性约束，我们得到了更加精确的谓词转化器。实验表明，在这些情况下，对于这些谓词转化器的验证要比采用最弱前置条件方法效率更高。

关键词：数据库，完整性约束保持，最弱前置条件，谓词转化器

Abstract

Integrity constraints are important tools to express the semantics of databases and can be used for query optimization. But no real DBMS (database management system) have fully support the management of integrity constraints. Instead, they use triggers to replace integrity constraints. However, the behavior of triggers is complex and the semantics of them are hard to understand. Therefore, it is necessary to develop approaches for integrity constraints verification, and we expect the verification can be done at compile-time so that efficiency problems can be avoided.

In this paper, we present a strategy to automatically verify the integrity constraints of databases. Our method is based on the weakest precondition and predicate transformer approaches. First we reduce database integrity constraints in SQL into SQL assertions, and then transfer assertions into FOL (first-order logic) formula. We also investigate the semantics of the three data modification operations in SQL (INSERT, DELETE and UPDATE). Based on the logical formalization of both SQL assertions and data modification operations, we implement integrity constraints checking for databases with the help of the program verification platform Why3. For the input SQL statements, our program translate them into WhyML program, later Why3 is called to compute the weakest preconditions and generate the verification conditions for the back-end provers (such as Alt-Ergo, CVC, *etc.*). Finally the provers will check whether the databases after executing the data modification operations satisfy the constraints. All the process is fully automatic.

In addition, we obtain more precise backward predicate transformers for some specified constraints. Experiments show that in these special cases, verifying these predicate transformers is more efficient than the weakest precondition approach.

Keywords: database, integrity constraint checking, weakest precondition, predicate transformer

目 录

绪论

1 课题背景

完整性约束（integrity constraint）是数据库语义信息的重要方面：一方面，数据库完整性约束可以防止违反约束的不合法操作或事务（transaction）被执行；另一方面，数据库完整性约束可以用来实现语义查询优化（semantic query optimization），从而提高数据库的查询性能。

除了主键（primary key）、外键（foreign key）之外，SQL 已经支持断言（assertion）和 CHECK 子句限定的约束，这些约束可以表达数据库中的复杂关系。但目前实际应用的数据库管理系统（Database Management System，简称 DBMS）却仍无法支持早在 1992 年就被纳入 SQL2 标准的断言，一些数据库管理系统对 CHECK 子句的支持也仍然有限。造成数据库管理系统对完整性约束支持不佳的原因是：每当执行完一次更新操作（例如执行 SQL 的 INSERT、DELETE、UPDATE 命令去相应地增加元组、删除元组或更新元组），数据库管理系统都要去遍历数据库中的数据来检验是否满足约束，这无疑会使数据库的执行效率大打折扣。而对于常常要处理大数据量的实际数据库管理系统来说，这种效率是无法忍受的，这也促使一些主流数据库管理系统（Oracle、DB2 等）转而采取触发器（trigger）作为解决方案。

触发器通常是由数据库管理员（Database Administrator，简称 DBA）定义好的会由数据库管理系统动态执行的程序，它由三部分组成：

1. 事件（event）：激活触发器的数据库数据的改变。
2. 条件（condition）：当触发器被激活后会执行的查询或检验条件。
3. 动作（action）：当触发器被激活且条件满足时会执行的程序。

与完整性约束不同的是，触发器是由操作激活的，不需要永久保持，这使数据库管理系统较容易实现，但与此同时也使触发器不具备与完整性约束一样的可理解性，为了实现特定的约束需要数据库管理员有较高的编程水平来写出一系列正确的触发器，而且触发器也难以用来实现语义查询优化。更为重要的问题是：触

发器是允许级联（cascade）的，也就是说一个触发器的动作可以激活另一个触发器甚至其本身——这种触发器称为递归触发器（recursive trigger）。当一条 SQL 命令激活了一系列触发器时，就产生了冲突问题：有些数据库管理系统定义了一些规则（例如 Oracle 是根据创建时间戳按升序执行被激活的触发器），但这些规则很容易造成被激活触发器的实际行为与预想行为不一致，从而增加了管理难度；有些数据库管理系统则会以任意顺序执行被激活的触发器，这使得触发器的实际行为更加难以预测和管理。

因此，目前被一些主流数据库管理系统所采用的触发器无法替代一般的完整性约束，我们仍然需要实现一种不影响数据库的执行效率的数据库完整性约束验证方法，并且希望这一验证过程需要尽量少的人工干预、做到自动化。这也正是我们这个课题的意义所在。

2 本文研究目标和内容

SQL 完整性约束可以分为四种：

1. 域约束（domain constraint）
2. 列约束（column constraint）
3. 表约束（table constraint）
4. 断言（assertion）

*域约束*是由 CHECK 关键字和 CHECK 之后的搜索条件构成，通常作为 CREATE DOMAIN 语句的一部分出现，会对以该域为类型的列进行约束。域约束的一般形式在 SQL 标准中被定义如下：

```
CREATE DOMAIN <domain name> AS <predefined type>
[ CONSTRAINT <constraint name> ] CHECK ( <search condition> )
```

其中，<predefined type> 指的是 SQL 语言内置的数据类型（如 INT、REAL、FLOAT、DOUBLE 等），<search condition> 表示搜索条件。

*列约束*和*表约束*通常作为 CREATE TABLE 语句的一部分出现，二者都可以分为如下三类：

1. 键约束（unique constraint），由关键字 UNIQUE 或 PRIMARY KEY 限定。
2. 外键约束（referential constraint），由关键字 FOREIGN KEY 限定。

3. 由 CHECK 子句限定的约束（CHECK constraint），根据关键字 CHECK 之后的搜索条件进行一般性的限定。

列约束和表约束的不同在于，列约束只能对特定的列进行约束，而表约束还可以对同一表中的不同列进行约束，因而列约束可以被视为是表约束的特殊形式。事实上，根据 [15] 和 [4]，任意的列约束也都可以转化为等价的表约束。

*断言*是 SQL 静态约束的最一般形式，允许对单个表或多个表进行约束。断言在 SQL3 标准中定义如下：

```
CREATE ASSERTION <assertion name>
CHECK <search condition>
```

在我们的课题中，数据库的更新操作主要是 SQL 的 INSERT、DELETE、UPDATE 语句，根据 SQL 标准所给出的语法定义，这些操作都是单表更新，但搜索条件可以涉及多表。设数据库的完整性约束为 C（在这里由 SQL 断言表示），数据库的更新操作为 U（在这里由 SQL 的 INSERT 或 DELETE 或 UPDATE 语句表示），我们的目标是：已知数据库 $\mathcal{B}$ 在执行 U 前满足 C（记为 $\mathcal{B} \models C$）的前提下，验证 $\mathcal{B}$ 在执行 U 后是否依然满足 C（记为 $U(\mathcal{B}) \models C$）。

3 本文结构安排

本文的结构安排如下：

第二章：总结整个完整性约束执行（integrity constraint enforcement）领域的研究问题、分析和比较各种研究方案。

第三章：对作为我们研究对象的数据库完整性约束和数据库修改操作进行逻辑上的形式化，并给出我们用在 SQL 语言的数据库完整性约束保持的具体方案。

第四章：介绍 Why3 自动验证平台以及基于 Why3 的数据库约束自动验证的实现。

第五章：展示和分析实验结果。

第六章：总结概括本文工作, 指出本文工作的贡献和存在的不足之处, 并对可以进一步研究的问题进行了展望。

数据库完整性约束执行综述

数据库的完整性约束（integrity constraint）指定了数据库在每个状态都必须满足的条件。由于每次数据修改操作都会对数据库的状态造成改变，可能会使一些约束得不到保持，因此数据库需要一些机制来保证每次执行完数据修改操作后，数据库的完整性约束依然满足。完整性约束执行（integrity constraint enforcement）领域所要研究的就是这一问题。

1 问题定义及分类

为了描述清楚问题的需要，以下我们给出相关的形式化定义:

设 $\mathcal{U}$ 为常量的可数无限集构成的全局域，f 为逻辑表达式（formula），则从 f 中的自由变量到 $\mathcal{U}$ 上的常量的映射 ϕ 称为对 f 的一个的赋值（valuation）。

对于数据库 $\mathcal{B}$，公式 f 和对 f 的一个赋值 ϕ，若有 $\mathcal{B}_\phi \models f$ 成立，则我们称 $\mathcal{B}$ 在赋值 ϕ 下满足 f。

修改操作 U 是一个从数据库 $\mathcal{B}$ 到数据库 $U(\mathcal{B})$ 的映射，约束 C 是一个逻辑表达式。对于修改操作 U 和约束 C，如果对于所有数据库 $\mathcal{B}$ 有：

$$\mathcal{B} \models C \;\Rightarrow\; U(\mathcal{B}) \models C$$

成立，则我们称 U 保持 C，或 U 在 C 下安全（safe）。

根据对在约束 C 下不安全的修改操作 U 的处理方式的不同，完整性约束执行问题可以分为两大类：

1. 完整性约束检查（integrity constraint checking）：拒绝执行在约束 C 下不安全的修改操作 U。根据检查在修改操作 U 执行前或是执行后又分为两类：
    - 冲突检测（violation detection）：在执行完数据库修改操作 U 后对完整性约束进行检查，如果数据库不满足约束，则对 U 执行回滚（rollback）操作。
    - 冲突预防（violation prevention）：在执行数据库修改操作 U 前对

其进行约束检查，如果执行完该操作后数据库不满足约束，则该操作被中止（abort）。

2. 完整性约束保持（integrity constraint maintenance）：对数据库修改操作 U 进行修正，使得修正后的修改操作 U’ 保持完整性约束 C。

2 完整性约束检查

2.1 冲突检测

冲突检测的一种方案是将数据库完整性约束转化为一系列触发器来进行检查。给定数据库的修改操作 U，这种方案将根据数据库的完整性约束 C 生成触发器，这些触发器涵盖了 U 可能违反 C 的情况，当 U 被执行时，这些触发器被激活，对 U 的安全性进行检查。

Chen、Hull 和 Mcleod[10] 采用了一种可以自动推导得到的动态规则 Limited Ambiguity Rules（简称 LAR），并针对面向对象数据库应用了两种 LAR：一种是上行 LAR，把基类（base class）的改变传递给继承类（derived class），与之对应的是另一种把继承类的改变传递给基类的下行 LAR。由于 LAR 与特定的数据库的修改操作相对应（event），同时又是一种 condition-action 形式的规则，所以 LAR 实际上就是触发器的抽象表示，由特定的数据库的修改操作激活并进行完整性约束检查。LAR 被激活后，将按照先下行 LAR 再上行 LAR 的原则执行。LAR 方法较好地考虑了面向对象数据库的继承特征对完整性约束的影响，但却存在一个主要缺陷：对于一些数据库的修改操作，无法自动推导出动态规则 LAR。Decker[17, 18] 的方法则基于 SLD 归结法（Selective Linear Definite clause resolution）的一种扩展 SLDAI（SLD with Abduction and Integrity maintenance）归结法。这种方法采用了一种反驳式的归结过程：给定数据库的修改操作 U 作为归结目标和归结过程的起点，以后每一步均根据数据库的完整性约束从目标中选取一个子目标做归结，若当前子目标无法被归结时，引起回溯并选择下一个子目标进行归结。这个过程被递归执行，直到数据库的修改操作被转化为一系列触发器为止。但 Decker 的方法同样无法适用于所有的数据库修改操作，对于一些修改操作 SLDAI 无法给出归结或者归结有误。

总的来看，将数据库完整性约束转化为触发器的方案有如下优势：

- 触发器目前被较多主流数据库管理系统（Oracle、DB2 等）所采用，这使得该方案在实际中具有较好的可行性和可应用性。
- 根据该方案所发展出来的方法都做到了自动性，只有对一些无法推导或归结的数据库修改操作才需要人工干预，这就较好地解决了触发器管理和维护难的问题。

但仍无法避免触发器本身的局限性：

- 一方面，数据库完整性约束被分散在一系列触发器的定义中，这减低了数据库整体的语义信息的可理解性，因而很难基于这一系列触发器来实现实现语义查询优化；
- 另一方面，当一条 SQL 命令激活了一系列触发器时，由于一个触发器的动作可以激活另一个触发器甚至其本身，因而被激活触发器的实际行为难以预测。

与以上方法相对应的，是传统的在每次执行完数据库修改操作后、遍历数据库进行完整性约束检查的方案，这种方案有着不低于 O(n) 的开销，对于经常要处理海量数据的数据库管理系统来说，这样的开销会带来很大的性能损失。尽管 [33, 29, 27] 等提出了一些优化方法，但这些优化仍无法弥补这种效率的降低。不过值得一提的是，其中一些工作已经开始把逻辑程序设计（logic programming）、形式化验证（formal verification）领域的相关理论应用进来，这一思路在冲突预防问题得到了广泛的采用。例如 Lawley[27] 在演绎数据库和面向对象数据库中应用最弱前置条件方法，来避免对约束不安全的操作进行回滚操作所带来的开销。Lawley 的方法的一大缺陷是用来推导最弱前置条件的编程语言却过于简单，甚至不包含 if 语句等基本结构，这就使它的应用受到很大的限制。但是 Lawley 的方法却提供了一个合理的完整性约束检查的思路和框架，只不过这一思路多被应用到 compile-time 验证中来进行冲突预防，而不像 Lawley 那样仍然以 run-time 验证为主、将最弱前置条件方法当做冲突检测的一种补充。

2 .2 冲突预防

与冲突检测在 run-time 进行完整性约束检查不同，冲突预防在 compile-time 就对数据库修改操作的安全性进行验证。与冲突检测相比，这一方案具有如下优势：

- 没有每次执行完数据库修改操作后都要动态检查完整性约束的开销。
- 数据库修改操作在 compile-time 被验证为不安全的将不允许被执行，这样就不用考虑这样的操作被执行后的回滚问题。

所以，冲突预防方案可以较好地解决完整性约束检查引发的数据库管理系统的性能和效率降低的问题。

冲突预防方面的研究主要关注数据库完整性约束的逻辑特征和数据库修改操作的程序语义，逻辑方法和推导规则被广泛地应用到这一问题中来。早在 [23]，G.Gardarin 等就提出了应用 Hoare 逻辑到数据库完整性约束验证问题的方法，但 G.Gardarin 等只是通过实例来说明 Hoare 逻辑如何应用到冲突预防上，并没有给出形式化的表述和对相关应用过程进行证明，这使得 G.Gardarin 的方法不能被有效地、自动化地应用到实际的完整性约束检查上来。[9, 25] 提出了 partial subsumption 方法去对数据库完整性约束进行简化。这种方法可以处理简单的数据库增加、删除操作，而且可以用在数据库语义优化 [24, 26] 中。但是 partial subsumption 方法将复杂的数据库修改操作（例如数据库事务 transaction）看做一系列的插入和删除操作的序列的假设却值得商榷，因为这一序列的正确构造依赖于数据库中元组的具体数据，这事实上已经隐含了对数据库本身的遍历、而非冲突预防方案所提倡的对修改操作的逻辑结构的分析。而在 [24, 26] 中，对复杂的数据库修改操作的处理也只是通过一些例子阐述 partial subsumption 方法的应用，并没有将它一般化。Qian[34, 35] 承继之前 G.Gardarin 等的工作，应用了 Hoare 逻辑并构建了一个数据库事务（transaction）分析的原型系统。但 Qian 所处理的数据库事务却局限于单一的数据库修改操作（插入元组、删除元组、更新元组等），与数据库实际所用的“事务”有一定的差距。

在冲突预防方面较重要的一种方法就是上文提及的最弱前置条件方法（或与之相应的最强后置条件方法）。McCune 和 Henschen[32] 给出了在关系数据库（relational database）上应用最弱前置条件方法的理论基础，Stemple 和 Sheard[36] 则基于 Boyer-Moore 可计算逻辑，实现了一个复杂的推导最弱前置条件并验证的专家系统。之后 Lawley 和 Topor 等 [28] 基于一阶逻辑，实现了比 Stemple 和 Sheard 的方法更复杂的数据库修改操作的最弱前置条件推导和验证。

最弱前置条件（最强后置条件）方法的好处在于它提供了数据库修改操作的精确的逻辑语义信息，即该操作保持数据库约束的充分必要条件。但这种方法也存在着局限性：

- 最弱前置条件方法无法适用于所有的数据库修改操作。Benedikt 和 Griffin 等 [5] 证明了，存在一些简单的数据库事务无法得到一阶逻辑的最弱前置条件，而且可以完全推导出一阶逻辑的最弱前置条件的一类数据库事务无法被任何数据库事务编程语言所表达。
- 由于最弱前置条件的推导一般采用 deductive programming，需要数据库修改操作所用的程序语言具有可被演绎的语法结构，所以这对数据库修改操作所用的程序语言也有限制。Clarke[11] 也证明了对于包含别名（aliasing）等特征的复杂数据库编程语言，无法得到一个定义良好的 Hoare 公理系统，因而也无法生成最弱前置条件。

针对最弱前置条件方法的不足，一些研究者采取了接近最弱前置条件的前置条件（或相应的，接近最强后置条件的后置条件）来进行数据库完整性约束检查。在 [38] 中，Wallace 通过扩展递归更新序列（recursive update consequences）[8] 方法来得到接近最弱前置条件的前置条件。[6] 则是采用了 Dijkstra 的谓词转化器（predicate transformer）理论 [20, 21]，去定义数据库修改操作的前向谓词转换器（forward predicate transformer）和后向谓词转换器（backward predicate transformer），从而分别得到接近最强后置条件的后置条件和接近最弱前置条件的前置条件。这种折衷方式拓展了最弱前置条件（最强后置条件）方法所能验证的问题的范围，而且能够处理具有递归等高级特征的复杂编程语言。但这一类方法由于所用的前置条件（后置条件）在数据库修改操作的逻辑结构的表达上并没有最弱前置条件（最强后置条件）精确和全面，所以尽管它们仍然可以判断出所有的不安全修改操作，但却会把一些安全的修改操作也误判为不安全。

3 完整性约束保持

完整性约束保持问题研究的是：对于给定的数据库修改操作 U，首先得到一个逻辑表达式 f，f 定义了数据库中会被更新的数据的条件，因而也称为数据库修改操作 U 的查询条件——一个直观的例子就是 SQL 的 UPDATE 命令中的

WHERE 子句所限定的查询条件。接着结合查询条件 f 和数据库完整性约束 C 这两个逻辑表达式去生成新的查询条件 f'（数据库修改操作 U 也被更新为 U'），使得在 f' 的限定下，U’ 在约束 C 下安全。

在完整性约束保持方面，Wuthrich[39] 提出了一种针对可拆分为一系列基本的插入和删除操作的数据库修改操作的方法，这种方法根据基本的插入和删除操作执行后数据库增加的约束与原来的完整性约束，对修改操作进行修正。但对于一些修改操作，Wuthrich 的方法仍无法给出修正。Console、Sapino 和 Theseider[13] 的方法则对新的查询条件 f' 进一步根据可能引起冲突的赋值进行枚举和相应简化，他们的方法相比 Wuthrich 的方法可以应用在更多的修改操作上，但可适用的数据库完整性约束却有着较多的限制：例如完整性约束必须是否定式，且不包含超过两个的文字（literal）。Lobo 和 Trajcevski[30] 的方法则是把查询条件 f 转化为析取范式（disjunctive normal form），检查 f 展开的项是否可能与完整性约束 C 冲突（与 Console 等的方法类似，通过枚举可能引起冲突的赋值），最终将这些被修正的项合成新的查询条件 f'。这种方法对于修改操作和完整性约束的限制较少，但对于某些修改操作，这种方法所给出的修正后的数据库修改操作仍然不保持完整性约束。

完整性约束保持通过数据库修改操作 U 的修正，实现了数据库完整性约束的有效性，而且避免了 U 的执行所带来的约束检查和回滚等问题。但这一方案同时也有一些局限性和有争议的地方：

- 完整性约束保持无法适用于所有的完整性约束 [37]。
- 由于完整性约束保持会过滤掉在数据库修改操作执行后违反约束的元组、但同时却对其它元组进行了更新，所以与数据库系统所建议的事务（transaction）原子性（automicity）原则有所矛盾。

4 本章小结

本章主要介绍了完整性约束执行（integrity constraint enforcement）领域的研究问题和研究方案。首先我们给出了数据库约束安全的修改操作等相关的定义、并根据定义阐述了这一领域下的子问题的分类。接着，我们解释了各个子问题下的各种方法，并分析了它们的优势和劣势。对于我们这一课题来说，我们采用的是数据库完整性约束检查（integrity constraint checking）中的冲突预防（violation

prevention）方案，前面所介绍的最弱前置条件（weakest precondition）及谓词转化器（predicate transformer）等方法正是我们这一研究课题的理论基础。在下一章中，我们将详细介绍我们用在 SQL 语言中的研究方案，并针对谓词转化器方法的不足之处提出改进措施。

数据库完整性约束保持

1 数据库完整性约束到 SQL 断言的规约

在 [4] 中，Behrend 等提出，数据库的静态约束都可以规约为在逻辑上等价的断言，并给出了如下的转化层次：

- 域约束可以转化为列约束。
- 列约束可以转化为表约束。
- 表约束可以转化为断言。

这说明我们可以对数据库完整性约束进行必要的简化，这对于后面从逻辑角度描述数据库完整性约束是有益的。然而 [4] 只是通过一些例子来阐述上述方案，因此我们仍然需要将它一般化。本节旨在结合数据库完整性约束的 SQL 标准语法定义给出具体的规约规则，从而证明 SQL 断言在逻辑上足以表达任意的数据库完整性约束，我们可以在不影响一般性的前提下将我们的课题规约到 SQL 断言的验证上。

对于域约束，设其所限制的域为 d，则

CREATE DOMAIN d AS <predefined type>
[CONSTRAINT <constraint name>] CHECK (<search condition>)

假设域 d 所限制的是表 r 的属性 a，即 d 被用在如下 CREATE TABLE 语句中：

CREATE TABLE r (. . ., a d, . . .)

则该域约束在逻辑上等价于以下的表约束：

CREATE TABLE r
(
. . . , a <predefined type>, . . . ,
CHECK (<search condition>)
)

此外对于列约束，[15] 说明了对于任意的列约束，都可以被等价的表约束替换，因此这一问题又被归结到表约束到断言的规约问题。如绪言中所述，表约束分为键约束（unique constraint）、外键约束（referential constraint）和由 CHECK 子

句限定的约束（CHECK constraint），以下我们将分别研究它们到 SQL 断言的规约规则。

1. 键约束：对于如下被 UNIQUE 关键字限定的键约束：

   CREATE TABLE r
   (
   ...
   [CONSTRAINT <constraint name>] UNIQUE ($a_1, \ldots, a_n$)
)

   这一约束的语义是：属性集合 $a_1, \ldots, a_n$ 可以确保在表 r 中唯一标识元组，换句话说就是：在表 r 中不存在两组不同的元组，它们取属性 $a_1, \ldots, a_n$ 的值完全相同。根据这一语义，我们可以将其规约为如下断言：

   CREATE ASSERTION <assertion name>
   CHECK (NOT EXISTS (SELECT * FROM $r\ x$
   WHERE (EXISTS (SELECT * FROM $r\ y$ WHERE $x <> y$
   AND $x.a_1 = y.a_1$ AND ... AND $x.a_n = y.a_n$))))

   对于如下被 PRIMARY KEY 关键字限定的键约束：

   CREATE TABLE r
   (
   ...
   [CONSTRAINT <constraint name>] PRIMARY KEY ($a_1, \ldots, a_n$)
)

   PRIMARY KEY 的语义是：

   (a) 属性集合 $a_1, \ldots, a_n$ 可以确保在表 r 中唯一标识元组，这一点与 UNIQUE 的语义一致。

   (b) 属性集合 $a_1, \ldots, a_n$ 中不存在任何一个子集满足条件 (a)。

根据这一语义，我们可以将其规约为如下断言：

CREATE ASSERTION <assertion name>
CHECK (
(NOT EXISTS (SELECT * FROM r x
WHERE (EXISTS (SELECT * FROM r y WHERE $x <> y$
AND $x.a_1 = y.a_1$ AND . . . AND $x.a_n = y.a_n$))))
AND
(EXISTS (SELECT * FROM r x
WHERE (EXISTS (SELECT * FROM r y WHERE $x <> y$
AND (
($x.a_2 = y.a_2$ AND $x.a_3 = y.a_3$ AND . . . AND $x.a_n = y.a_n$)
OR ($x.a_1 = y.a_1$ AND $x.a_3 = y.a_3$ AND . . . AND $x.a_n = y.a_n$)
OR ($x.a_1 = y.a_1$ AND $x.a_2 = y.a_2$ AND . . . AND $x.a_{n-1} = y.a_{n-1}$)))))))

2. 外键约束：对于如下由 FOREIGN KEY 限定的外键约束：

CREATE TABLE r
(
. . .
FOREIGN KEY (a) REFERENCES t(b)
)

我们可以将其规约为如下断言：

CREATE ASSERTION <assertion name>
CHECK (NOT EXISTS (SELECT * FROM r x
WHERE (NOT EXISTS (SELECT * FROM t y WHERE $x.a = y.b$))))

3. 由 CHECK 子句限定的约束：

CREATE TABLE r
(
. . .
CHECK (<search condition>)
)

我们可以将其规约为如下断言：

CREATE ASSERTION <assertion name>
CHECK (NOT EXISTS (SELECT * FROM r x
WHERE NOT <search condition>))

2 SQL 断言到一阶逻辑表达式的转化

2 .1 目标语言

这一节我们将通过元组关系演算（tuple relational calculus）来对我们的目标语言——一阶逻辑表达式（first-order logic formula）进行定义。

一阶逻辑表达式可以包括：

- 常量（0, 1, ⋯, etc.）
- 数据库关系（relation）变量（r, etc.），元组（tuple）变量（x, etc.）和属性（attribute）变量（a, etc.）
- 数值运算符（+, -, *, /, ⋯, etc.）
- 比较符（=, $\neq$, <, >, $\leq$, $\geq$, ⋯, etc.）和集合运算符（$\in$）
- 量词（$\forall$, $\exists$）

一阶逻辑表达式的项（term）定义如下：

- 所有的常量和变量都是项。
- 如果 x 是一个元组变量，a 是 x 的一个属性，则 $x.a$ 是项。
- 如果 p 和 q 都是项，则“p < 数值运算符 > q”是项。

一阶逻辑表达式按照以下规则被构造：

- 如果 x 是一个元组，r 是一个关系变量，则“$x \in r$”是一阶逻辑表达式。
- 如果 p 和 q 是项，则“p < 比较符 > q”是一阶逻辑表达式。
- 如果 ϕ 和 φ 是一阶逻辑表达式，则 $\phi \wedge \varphi$，$\phi \vee \varphi$ 和 $\neg\phi$ 是一阶逻辑表达式。
- 如果 ϕ 是一阶逻辑表达式，x 是一个元组变量，则 $\forall x, \phi$ 和 $\exists x, \phi$ 是一阶逻辑表达式。

2 .2 源语言

由上一节所述，我们的源语言可以从数据库完整性约束规约到 SQL 断言上。我们参考 SQL2003 标准，用以下语法规则来表达 SQL 断言（以后若非特别说明，我们将用带尖括号的正体变量表示非终结符，而带尖括号的斜体变量表示终结符）：

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>

$$
\begin{array}{rcl}
\text{<search condition>} & ::= & \text{<boolean term>} \\
& | & \text{<search condition> OR <boolean term>} \\
\text{<boolean term>} & ::= & \text{<boolean factor>} \\
& | & \text{<boolean term> AND <boolean factor>} \\
\text{<boolean factor>} & ::= & \text{<predicate>} \\
& | & \text{[NOT] (<search condition>)} \\
\\
\text{<predicate>} & ::= & \text{<exists predicate>} \\
& | & \text{<comparison predicate>} \\
& | & \text{<between predicate>} \\
& | & \text{<in predicate>} \\
\text{<comparison predicate>} & ::= & \text{<expression}_1\text{ > <comp op> <expression}_2\text{ >} \\
\text{<comp op>} & ::= & = \mid <> \mid < \mid \leq \mid > \mid \geq \\
\text{<expression>} & ::= & \text{<term>} \\
& | & \text{<expression> } \{+ \mid -\} \text{ <term>} \\
\text{<term>} & ::= & \text{<factor>} \\
& | & \text{<term> } \{* \mid /\} \text{ <factor>} \\
\text{<factor>} & ::= & (\text{<expression>}) \\
& | & [+ \mid -] \text{ <}\textit{constant}\text{>} \\
& | & [+ \mid -]\, x.a \\
\text{<between predicate>} & ::= & \text{<expression> [NOT]} \\
& & \text{BETWEEN <}\textit{constant}_1\text{ > AND <}\textit{constant}_2\text{ >} \\
\text{<in predicate>} & ::= & \text{<expression> [NOT] IN (<in value list>)} \\
\text{<in value list>} & ::= & \text{<}\textit{constant}\text{>} \\
& | & \text{<in value list>, <}\textit{constant}\text{>}
\end{array}
$$

2.3 转化函数

我们设将 SQL 断言映射成一阶逻辑表达式的函数为 $\mathcal{T}$，则函数 $\mathcal{T}$ 的定义如下（为了避免和 SQL 断言所用的括号“(”“)”相混淆，我们用 $\mathcal{T}[]$ 而非 $\mathcal{T}()$ 来表示）：

$$
\begin{array}{rcl}
\mathcal{T}[\text{CREATE ASSERTION <}\textit{assertion name}\text{>} \\
\quad \text{CHECK <exists predicate>}] & \rightsquigarrow & \text{<}\textit{assertion name}\text{>} : \mathcal{T}[\text{<exists predicate>}] \\
\mathcal{T}[\text{EXISTS (<query expression>)}] & \rightsquigarrow & \exists\ (\mathcal{T}[\text{<query expression>}]) \\
\mathcal{T}[\text{NOT EXISTS (<query expression>)}] & \rightsquigarrow & \neg\exists\ (\mathcal{T}[\text{<query expression>}])
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}[\ \text{SELECT *} & \rightsquigarrow & x_1 \in r_1, \ldots, x_n \in r_n; \mathcal{T}[\text{<search condition>}] \\
\quad \text{FROM } r_1\ x_1, \cdots, r_n\ x_n & & \\
\quad \text{WHERE <search condition> }] & & \\
\mathcal{T}[\text{<search condition> OR <boolean term>}] & \rightsquigarrow & \mathcal{T}[\text{<search condition>}] \vee \mathcal{T}[\text{<boolean term>}] \\
\mathcal{T}[\text{<boolean term> AND <boolean factor>}] & \rightsquigarrow & \mathcal{T}[\text{<boolean term>}] \wedge \mathcal{T}[\text{<boolean factor>}] \\
\mathcal{T}[\text{NOT (<search condition>)}] & \rightsquigarrow & \neg\ (\mathcal{T}[\text{<search condition>}])
\end{array}
$$

$$\begin{aligned}
\mathcal{T}[\text{<expression}_1\text{>} \text{<comp op>} \text{<expression}_2\text{>}] &\rightsquigarrow \mathcal{T}[\text{<expression}_1\text{>}] \\
&\quad \mathcal{T}[\text{<comp op>}]\, \mathcal{T}[\text{<expression}_2\text{>}] \\
\mathcal{T}[\text{<comp op>}] &\rightsquigarrow \text{<comp op>}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\text{<expression}_1\text{>} \text{<numerical op>} \text{<expression}_2\text{>}] &\rightsquigarrow \mathcal{T}[\text{<expression}_1\text{>}] \\
&\quad \mathcal{T}[\text{<numerical op>}]\, \mathcal{T}[\text{<expression}_2\text{>}] \\
\text{<numerical op>} &::= + \mid - \mid * \mid / \\
\mathcal{T}[\text{<numerical op>}] &\rightsquigarrow \text{<numerical op>}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\text{<}constant\text{>}] &\rightsquigarrow \text{<}constant\text{>} \\
\mathcal{T}[x.a] &\rightsquigarrow x.a \\
\mathcal{T}[\text{<expression> BETWEEN } \text{<}constant_1\text{>} \text{ AND } \text{<}constant_2\text{>}] &\rightsquigarrow (\mathcal{T}[\text{<expression>}] \geq \text{<}constant_1\text{>}) \\
&\quad \wedge (\mathcal{T}[\text{<expression>}] \leq \text{<}constant_2\text{>}) \\
\mathcal{T}[\text{<expression> NOT BETWEEN } \text{<}constant_1\text{>} \text{ AND } \text{<}constant_2\text{>}] &\rightsquigarrow (\mathcal{T}[\text{<expression>}] < \text{<}constant_1\text{>}) \\
&\quad \vee (\mathcal{T}[\text{<expression>}] > \text{<}constant_2\text{>}) \\
\mathcal{T}[\text{<expression> IN (<in value list>)}] &\rightsquigarrow \mathcal{T}[\text{<expression>}] \in \{\text{<in value list>}\} \\
\mathcal{T}[\text{<expression> NOT IN (<in value list>)}] &\rightsquigarrow \mathcal{T}[\text{<expression>}] \notin \{\text{<in value list>}\}
\end{aligned}$$

设 exp 为一个 SQL 断言子句，则我们有：

$$\mathcal{T}[(exp)] \rightsquigarrow (\mathcal{T}[exp])$$

3 SQL 语言数据库修改操作的逻辑语义

在把 SQL 断言转化为一阶逻辑表达式之后，我们也需要对数据库修改操作的语义在逻辑层面进行形式化，以确保我们之后用在证明和实现中的数据库修改操作的逻辑语义与相应操作的实际语义相符。与 [6] 等研究的是 O_2、$Prolog$ 等数据库编程语言、[28] 等研究的是自行定义的编程语言的数据库修改操作不同，这一节我们研究的对象是 SQL 语言标准的数据库修改操作。

在 SQL 语言中，数据库修改操作分为 INSERT、DELETE 和 UPDATE 三类，下面我们将分别给出它们的一般形式和逻辑语义。为了阐述问题的简洁起见，我们在这里只给出这些操作一般形式的最主要的部分，具体的语法细节（例如 DELETE 语句可以加上 USING 子句来引用相关的数据库表等）将会在下一章详细给出。以下我们设初始数据库表为 r，SQL 数据库修改操作为 U，执行完 U 后的数据库表为 $U(r)$。

3 .1 SQL 的 INSERT 操作

设要插入数据库的元组为 t，则把 t 插入到表 r 中的 SQL 语句的一般形式为：

INSERT INTO r
VALUES t

SQL 的 INSERT 操作的语义为：数据库表 r 中的所有旧元组仍在 $U(r)$ 中，而元组 t 也会被插入到数据库表 $U(r)$ 中；除此之外，$U(r)$ 不会有其它元组。这一语义可以被形式化表示为：

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

3 .2 SQL 的 DELETE 操作

SQL 的 DELETE 语句的一般形式为：

DELETE FROM r
WHERE <search condition>

设 SQL 的 DELETE 语句中的搜索条件（<search condition>）所对应的逻辑表达式为 g，则 DELETE 语句的语义为：数据库 r 中所有满足 g 的元组都将被删除，因此不会出现在数据库 $U(r)$ 中，而其它的元组将被继续保留在数据库 $U(r)$ 中。这一语义可以被形式化表示为：

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

由于 $(g(x) \Rightarrow \neg(x \in U(r))) \equiv (\neg g(x) \Leftarrow x \in U(r))$，上述逻辑表达式等价于：

$$\forall x \in r : \neg g(x) \Leftrightarrow x \in U(r)$$

此外，任一 SQL 的 DELETE 语句都不会导致新的元组被插入到数据库 $U(r)$ 中，数据库 $U(r)$ 中的元组必定也是数据库 r 中的元组（即 $x \in U(r) \Rightarrow x \in r$）。综合这一语义信息，上述逻辑表达式可以被重写为：

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

3 .3 SQL 的 UPDATE 操作

SQL 的 UPDATE 语句的一般形式为；

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

设 SQL 的 UPDATE 语句中 <set clause> 所对应的赋值函数为 σ，搜索条件（<search condition>）所对应的逻辑表达式为 g，则 UPDATE 语句的语义为：数据库 r 中所有满足 g 的元组都将被替换为 $\sigma(x)$，而其它元组不变。这一语义可以被形式化表示为：

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

4 最弱前置条件及改进的谓词转化器方法

我们的主要方案是应用最弱前置条件（weakest precondition）来进行数据库约束的验证。下面我们给出最弱前置条件的相关定义。

前置条件（precondition）来自于 Hoare 逻辑中的 Hoare 三元组，具体到数据库中，前置条件可以形式化地定义为：

设 U 为修改操作，f 为一个逻辑表达式。如果逻辑表达式 g 满足：对于所有的数据库 $\mathcal{B}$，都有

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

成立，则逻辑表达式 g 称为修改操作 U 和逻辑表达式 f 的前置条件。

如果前置条件 h 满足：对于所有的前置条件 g 和数据库 $\mathcal{B}$，都有

$$\mathcal{B} \models g \ \Rightarrow \ \mathcal{B} \models h$$

成立，则前置条件 h 称为 U 和 f 的最弱前置条件。

最弱前置条件一般可以通过演绎编程（deductive programming）的方法来得到，即定义从基本函数块和后置条件到最弱前置条件的转化以及一套演绎规则，再对一般的函数体和后置条件演绎出对应的最弱前置条件。

我们所要验证的是执行完修改操作 U 后的数据库 $U(\mathcal{B})$ 仍然满足完整性约束 C（即 $U(\mathcal{B}) \models C$），而我们已知的是数据库 $\mathcal{B}$ 在修改操作 U 执行前满足约束（即 $\mathcal{B} \models C$），应用最弱前置条件方法，我们只需验证 C 和 U 的最弱前置条件 $wpc(C, U)$ 满足 $C \Rightarrow wpc(C, U)$，就可以由已知条件得到 $\mathcal{B} \models wpc(C, U)$，再根据最弱前置条件的性质，我们就得到了 $\mathcal{B} \models C$，这正是我们所要验证的目标。

作为我们研究方案的补充的是 [6] 所用的前向或后向谓词转化器（predicate transformer）。前向谓词转化器 $\vec{U}$ 是具有如下性质的逻辑表达式：

$$\mathcal{B} \models C \Rightarrow U(\mathcal{B}) \models \vec{U}\ (C)$$

通过定义前向谓词转化器，[6] 把数据库完整性约束验证问题转化为 $\overrightarrow{U}(C) \Rightarrow C$ 是否成立，而后者的验证不需要在执行完数据库修改操作后遍历数据库进行检查，从而实现在编译时间（compile-time）的冲突预防。与此相对应的，后向谓词转化器 $\overleftarrow{U}$ 具有如下性质：

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

后向谓词转化器定义后，数据库完整性约束验证问题转化为 $C \Rightarrow \overleftarrow{U}(C)$ 是否成立，后者的验证同样可在编译时间完成。

但相比最弱前置条件方法，谓词转化器无法最精确地表达数据库修改操作的行为，所以 [6] 的方法尽管可以检测出所有的约束不安全的数据库修改操作，却也会把部分约束安全的数据库修改操作误判为约束不安全。以后向谓词转化器 $\overleftarrow{U}$ 为例，我们的目标是验证 $U(\mathcal{B}) \models C$，已知的是 $\mathcal{B} \models C$，由 $C \Rightarrow \overleftarrow{U}(C)$ 可以推出 $\mathcal{B} \models \overleftarrow{U}(C)$，而由于 $\mathcal{B} \models \overleftarrow{U}(C)$ 只是我们所要验证的目标的充分不必要条件，因此该方法无法正确判断出所有的约束安全的数据库修改操作。假如我们可以找到这样的逻辑表达式 C'，使得：

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

则执行完修改操作的数据库是否保持约束 C 与初始数据库是否满足逻辑表达式 C' 等价，通过验证 $\mathcal{B} \models C'$ 来检测 $U(\mathcal{B}) \models C$ 不会导致约束安全的数据库修改操作被误判。下面我们将说明，在之前所阐述的 SQL 语言的数据库修改操作语义的基础上，我们可以对两类普遍的数据库完整性约束 C 找到等价的逻辑表达式 C'。

1. C 是形如下式的由存在量词限定的谓词公式：

$$\exists x \in R, f(x) \tag{3.1}$$

其中，R 是数据库关系变量，$f(x)$ 是含有自由变量 x 的逻辑表达式。在 C 中，R 是自由变量，x 成为约束变量，因而我们也可以用 $C(R)$ 来表示 C。已知初始的数据库表 r 满足 $C(r)$，即 $r \models \exists x \in r, f(x)$，我们要验证的目标是执行完修改操作后的数据库表 $U(r)$ 满足 $C(U(r))$，即 $U(r) \models \exists x \in U(r), f(x)$。

定理 1. 对于形如(3.1)的约束 C，任一 SQL 的 INSERT 操作都是约束安全的。

证明 . 设 x 为在数据库表 r 中满足 $f(x)$ 的元组。根据上一节中 SQL 的 INSERT 操作的性质：$x \in r \Rightarrow x \in U(r)$ ，我们得知元组 x 也在数据库表 $U(r)$ 中。因为数据库表 $U(r)$ 存在一个元组 x 满足逻辑表达式 f，即 $U(r) \models \exists x \in U(r), f(x)$，所以 $U(r) \models C(U(r))$ 成立。 □

定理 2. 执行完 SQL 的 DELETE 操作的数据库表 $U(r)$ 满足约束 $C(U(r))$ 的充分必要条件是：$\exists x \in r, f(x) \wedge \neg g(x)$

证明 . 下面我们分别证明其充分性和必要性：

- 充分性 $r \models \exists x \in r, f(x) \wedge \neg g(x) \Rightarrow U(r) \models C(U(r))$：
  设 x 为在数据库表 r 中满足 $f(x) \wedge \neg g(x)$ 的元组。根据上一节中 SQL 的 DELETE 操作的性质：

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

  元组 x 必定也在数据库表 $U(r)$ 中。因为数据库表 $U(r)$ 存在一个元组 x 满足逻辑表达式 f，即 $U(r) \models \exists x \in U(r), f(x)$，所以

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

  成立，充分性得证。
- 必要性 $U(r) \models C(U(r)) \Rightarrow r \models \exists x \in r, f(x) \wedge \neg g(x)$：
  我们证明与其等价的逆否命题：

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

  注意到：

  $$\begin{aligned} &\neg(\exists x \in r, f(x) \wedge \neg g(x)) \\ \equiv &\forall x \in r, \neg(f(x) \wedge \neg g(x)) \\ \equiv &\forall x \in r, \neg f(x) \vee g(x) \\ \equiv &\forall x \in r, f(x) \Rightarrow g(x) \end{aligned}$$

  也就是说，对于数据库表 r 中任一满足 f 的元组 x，x 必定也满足逻辑表达式 g。根据上一节中 SQL 的 DELETE 操作的性质：

  $$\neg g(x) \Rightarrow x \notin U(r)$$

满足 f 的元组 x 都会被删除、都不在数据库表 $U(r)$ 中。又因为 $x \in U(r) \Rightarrow x \in r$，即 DELETE 操作不会引入新的满足 f 的元组，所以数据库表 $U(r)$ 中不存在满足 f 的元组，即 $U(r) \models \neg(\exists x \in U(r), f(x))$。因此逆否命题成立，必要性得证。

□

定理 3. 执行完 SQL 的 UPDATE 操作的数据库表 $U(r)$ 满足约束 $C(U(r))$ 的充分必要条件是：$\exists x \in r, (f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x)))$

证明 . 下面我们分别证明其充分性和必要性：

- 充分性 $r \models \exists x \in r, (f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x))) \Rightarrow U(r) \models C(U(r))$：
  所以数据库表 r 中要么存在满足 $f(x) \wedge \neg g(x)$ 的元组，要么存在满足 $g(x) \wedge f(\sigma(x))$ 的元组：
  - 设 x 为在数据库表 r 中满足 $f(x) \wedge \neg g(x)$ 的元组。根据上一节中 SQL 的 UPDATE 操作的性质：

    $$\neg g(x) \Rightarrow U(x) = x$$

    元组 x 必定也在数据库表 $U(r)$ 中。因为数据库表 $U(r)$ 存在一个元组 x 满足逻辑表达式 f，即 $U(r) \models \exists x \in U(r), f(x)$，所以

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

    成立。
  - 设 x 为在数据库表 r 中满足 $g(x) \wedge f(\sigma(x))$ 的元组。根据上一节中 SQL 的 UPDATE 操作的性质：

    $$g(x) \Rightarrow U(x) = \sigma(x)$$

    以及元组 x 满足 $g(x)$，所以元组 $y = \sigma(x)$ 在数据库表 $U(r)$ 中。又因为元组 x 满足 $f(\sigma(x))$，所以数据库表 $U(r)$ 存在一个

元组 $y = \sigma(x)$ 满足逻辑表达式 f，即 $U(r) \models \exists y \in U(r), f(y)$，所以

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

成立。

综上所述，充分性得证。

- 必要性 $U(r) \models C(U(r)) \Rightarrow r \models \exists x \in r, (f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x)))$：

  我们证明与其等价的逆否命题：

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

  注意到：

  $$\begin{aligned}
  &\neg(\exists x \in r, (f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x)))) \\
  &\equiv \forall x \in r, \neg((f(x) \wedge \neg g(x)) \vee (g(x) \wedge f(\sigma(x)))) \\
  &\equiv \forall x \in r, \neg(f(x) \wedge \neg g(x)) \wedge \neg(g(x) \wedge f(\sigma(x))) \\
  &\equiv \forall x \in r, (\neg f(x) \vee g(x)) \wedge (\neg g(x) \vee \neg f(\sigma(x))) \\
  &\equiv \forall x \in r, ((\neg f(x) \vee g(x)) \wedge \neg g(x)) \vee ((\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x))) \\
  &\equiv \forall x \in r, ((\neg f(x) \wedge \neg g(x)) \vee (g(x) \wedge \neg g(x))) \vee ((\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x))) \\
  &\equiv \forall x \in r, (\neg f(x) \wedge \neg g(x)) \vee ((\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x)))
  \end{aligned}$$

  所以对于数据库表 r 任一元组 x，要么 x 满足 $\neg f(x) \wedge \neg g(x)$，要么 x 满足 $(\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x))$：

  - 如果元组 x 满足 $f(x)$，则 x 必须满足 $(\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x))$。由 $(\neg f(x) \vee g(x)) \equiv (f(x) \Rightarrow g(x))$，我们有：元组 x 满足 $g(x)$。根据上一节中 SQL 的 UPDATE 操作的性质：

    $$g(x) \Rightarrow U(x) = \sigma(x)$$

    所以元组 $y = \sigma(x)$ 在数据库表 $U(r)$ 中。又因为元组 x 满足 $\neg f(\sigma(x))$，所以在数据库表 $U(r)$ 中的元组 y 都满足 $\neg f(y)$。
  - 如果元组 x 不满足 $f(x)$、但满足 $g(x)$，则 x 也必须满足 $(\neg f(x) \vee g(x)) \wedge \neg f(\sigma(x))$。同上可得：元组 $y = \sigma(x)$ 在数据库表 $U(r)$ 中，并且都满足 $\neg f(y)$。

- 如果元组 x 同时不满足 $f(x)$ 和 $g(x)$，即 x 满足 $\neg f(x) \wedge \neg g(x)$。根据上一节中 SQL 的 UPDATE 操作的性质：

$$\neg g(x) \Rightarrow U(x) = x$$

任一满足 $\neg f(x) \wedge \neg g(x)$ 的元组 x 都在数据库表 $U(r)$ 中，而且都满足 $\neg g(x)$。

综上所述，逆否命题成立，必要性得证。

□

2. C 是形如下式的由全称量词限定的谓词公式：

$$\forall x \in R, f(x) \tag{3.2}$$

其中，R 是数据库关系变量，$f(x)$ 是含有自由变量 x 的逻辑表达式，在 C 中，R 是自由变量，x 成为约束变量，因而我们也可以用 $C(R)$ 来表示 C。已知初始的数据库表 r 满足 $C(r)$，即 $r \models \forall x \in r, f(x)$，我们要验证的目标是执行完修改操作后的数据库表 $U(r)$ 满足 $C(U(r))$，即 $U(r) \models \forall x \in U(r), f(x)$。

定理 4. 执行完 SQL 的 INSERT 操作的数据库表 $U(r)$ 满足约束 $C(U(r))$ 的充分必要条件是：新插入的元组 t 满足 $f(t)$。

证明 . 根据上一节中 SQL 的 INSERT 操作的性质：

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

我们有：

$$\forall x \in U(r), f(x) \Leftrightarrow (\forall x \in r, f(x) \wedge f(t))$$

由已知条件 $r \models C(r)$，即 $r \models \forall x \in r, f(x)$，我们得到：$\forall x \in U(r), f(x) \Leftrightarrow f(t)$。 □

定理 5. 对于形如(3.2)的约束 C，任一 SQL 的 DELETE 操作都是约束安全的。

证明. 对于数据库表 $U(r)$ 中的任一元组 x，根据上一节中 SQL 的 DELETE 操作的性质：

$$x \in U(r) \Rightarrow x \in r$$

x 必定也在初始数据库表 r 中。由已知条件：$r \models C(r)$，即 $r \models \forall x \in r, f(x)$，我们有：元组 x 满足逻辑表达式 f。所以 $U(r) \models \forall x \in U(r), f(x)$ 成立。 □

定理 6. 执行完 SQL 的 UPDATE 操作的数据库表 $U(r)$ 满足约束 $C(U(r))$ 的充分必要条件是：$\forall x \in r, \neg g(x) \vee (g(x) \wedge f(\sigma(x)))$

证明. 下面我们分别证明其充分性和必要性：

- 充分性 $r \models \forall x \in r, \neg g(x) \vee (g(x) \wedge f(\sigma(x))) \Rightarrow U(r) \models C(U(r))$：
  对于数据库表 r 中的任一元组 x，要么 x 满足 $\neg g(x)$，要么 x 满足 $g(x) \wedge f(\sigma(x))$：
  - 如果元组 x 满足 $\neg g(x)$：
    根据上一节中 SQL 的 UPDATE 操作的性质：

    $$\neg g(x) \Rightarrow U(x) = x$$

    元组 x 仍然在数据库表 $U(r)$ 中，由已知条件件：$r \models C(r)$，即 $r \models \forall x \in r, f(x)$，我们有：元组 x 满足逻辑表达式 f。
  - 如果元组 x 满足 $g(x) \wedge f(\sigma(x))$：
    根据上一节中 SQL 的 UPDATE 操作的性质：

    $$g(x) \Rightarrow U(x) = \sigma(x)$$

    元组 $y = \sigma(x)$ 必定在数据库表 $U(r)$ 中。又因为元组 x 满足 $f(\sigma(x))$，所以元组 y 满足 $f(y)$。

  综上所述，充分性得证。
- 必要性 $U(r) \models C(U(r)) \Rightarrow r \models \forall x \in r, \neg g(x) \vee (g(x) \wedge f(\sigma(x)))$：
  我们证明与其等价的逆否命题：

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

注意到：

$$\begin{aligned}
&\neg(\forall x \in r, \neg g(x) \vee (g(x) \wedge f(\sigma(x))))\\
&\equiv \exists x \in r, \neg(\neg g(x) \vee (g(x) \wedge f(\sigma(x))))\\
&\equiv \exists x \in r, g(x) \wedge \neg(g(x) \wedge f(\sigma(x)))\\
&\equiv \exists x \in r, g(x) \wedge (\neg g(x) \vee \neg f(\sigma(x)))\\
&\equiv \exists x \in r, (g(x) \wedge \neg g(x)) \vee (g(x) \wedge \neg f(\sigma(x)))\\
&\equiv \exists x \in r, g(x) \wedge \neg f(\sigma(x))
\end{aligned}$$

设 x 为数据库表 r 中满足 $g(x) \wedge \neg f(\sigma(x))$ 的元组，根据上一节中 SQL 的 UPDATE 操作的性质：

$$g(x) \Rightarrow U(x) = \sigma(x)$$

元组 $y = \sigma(x)$ 必定在数据库表 $U(r)$ 中。又因为元组 x 满足 $\neg f(\sigma(x))$，所以元组 y 满足 $\neg f(y)$。所以数据库表 $U(r)$ 存在一个元组 y 不满足 f，即 $U(r) \models \exists x \in U(r), \neg f(x)$，由于 $\exists x \in U(r), \neg f(x) \equiv \neg(\forall x \in U(r), f(x))$，所以逆否命题成立，必要性得证。

□

5 本章小结

前面提到我们用数据库完整性约束检查（integrity constraint checking）中的冲突预防（violation prevention）方案来实现数据库约束的编译时间（compile-time）验证，这一章在 SQL 语言的框架下对我们的这一方案进行了详细的说明。由于我们所采用的方法基于逻辑程序设计（logic programming）、形式化验证（formal verification）等领域的理论，所以我们需要在逻辑层面描述我们的研究对象——数据库完整性约束和数据库修改操作。对于数据库完整性约束，我们首先证明了 SQL 的任一完整性约束都可以规约为逻辑上等价的断言（assertion），从而简化了问题；接下来我们将 SQL 断言转化为一阶逻辑表达式（first-order logic formula）：我们根据元组关系演算（tuple relational calculus）和 SQL2003 标准对我们的目标语言以及源语言 SQL 断言做了具体的限定，最后给出了 SQL 断言到一阶逻辑表达式的转化规则。对于数据库修改操作，我们分析了 SQL 语言中的 INSERT、DELETE 和 UPDATE 三大数据库修改操作的逻辑语义。最后我

们具体说明了我们所采用的最弱前置条件（weakest precondition）和谓词转化器（predicate transformer）方法。我们还对约束的逻辑表达式是形如(3.1)或(3.2)的谓词公式情况给出了谓词转化器的改进，并给出相关结论的证明。值得一提的是，我们所得到的这组完整性约束的等价验证条件都可以在数据库修改操作执行前就进行验证，对这组等价验证条件的验证仍然具有冲突预防方法的优势。

基于 Why3 平台的数据库约束自动验证

1 Why3 平台及约束自动验证的具体实现

我们借助于 Why3[7] 平台来实现数据库约束的自动验证。Why3 是一个基于一阶逻辑（first-order logic）的程序验证平台，支持 Alt-Ergo[12]、Z3[16]、CVC3[3]、Yices[22]、Simplify[19]、Gappa[2]、Coq[1] 等多个后端定理自动证明器（prover）。Why3 定义了一种简单的具有类似 ML 函数式语言特征的 WhyML 语言，该语言采取 Hoare 三元组的形式来定义函数（其中 p 为前置条件，e 为函数体，q 为后置条件）：

$$\texttt{let}\ \ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \{p\}\ e\ \{q\}$$

并利用 Hoare 逻辑的推导规则得到该语言的执行语义（operational semantic）。基于此，Why3 平台为基本的程序块定义了完全的计算最弱前置条件方法的递归规则。根据这套规则，Why3 可以自动生成后置条件 q 关于函数体 e 的最弱前置条件。

在我们的课题中，Why3 平台可以用来自动得到数据库完整性约束 C 和数据库修改操作 U 的最弱前置条件 $wpc(C, U)$，同时将 Why3 程序本身所用的逻辑和句法（syntax）转化为后端各种定理自动验证器（prover）所用的逻辑和程序句法，由定理验证器来自动证明验证条件（verification condition）：$C \Rightarrow wpc(C, U)$。于是实现数据库完整性约束的自动验证我们需要完成的是从 SQL 语句到 WhyML 程序的自动转化。为此，我们实现了五个解释器（parser）：

1. SQL 的 CREATE TABLE 语句的解释器，从 CREATE TABLE 语句得到 WhyML 程序所需要的类型定义及导入的库。
2. SQL 断言的解释器，将 SQL 断言转化为 WhyML 程序的 predicate。
3. SQL 的 INSERT 操作的解释器，将 SQL 的 INSERT 操作转化为 WhyML 程序的函数。
4. SQL 的 DELETE 操作的解释器，将 SQL 的 DELETE 操作转化为 WhyML 程序的函数。

5. SQL 的 UPDATE 操作的解释器，将 SQL 的 UPDATE 操作转化为 WhyML 程序的函数。

我们的解释器自动生成的 WhyML 程序主要由以下几个部分组成：

1. 程序头部，主要是 import 所需要的 theory 或 module。
2. 变量定义。
3. 对应于 SQL 断言的 predicate 定义。
4. 对应于 SQL 数据库修改操作的函数。

下面我们将具体描述我们实现的五个解释器：我们将给出相应 SQL 语句的详细语法，并通过转化规则阐述如何将 SQL 语句转化为 WhyML 程序。在这里，我们所自动生成的目标程序的逻辑语义都基于上一章对数据库完整性约束和数据库修改操作的逻辑形式化工作。

2　SQL 的 CREATE TABLE 语句的解释器

在 WhyML 程序中，我们把数据库关系变量看作是元组变量的 list，把元组变量看作是记录属性（即表的列）的名称及其对应类型的 record 型变量。

对于如下的一般形式的 CREATE TABLE 语句：

<table definition> ::= CREATE TABLE <*table name*>
(<table element list>)
<table element list> ::= <table element>
| <table element list>, <table element>
<table element> ::= <*attribute name*> <*attribute type*>

我们将生成如下的 WhyML 代码：

`type` tupleType_<*table name*> = {|<attribute list>|}
<attribute list> ::= <attribute>
| <attribute list>; <attribute>
<attribute> ::= <*attribute name*>: <*attribute type*>

并根据属性的数据类型<*attribute type*> 导入相应的 theory 或 module。之后用到表名为<*table name*> 的关系变量，其类型为 `list` tupleType_<*table name*>。

3　SQL 断言的解释器

由于 WhyML 采用的是一阶逻辑（first order logic），所以我们可以采用上一章所述的 SQL 断言到一阶逻辑表达式的转化规则直接得到 SQL 断言到 WhyML 代码的转化规则。为了与之前将 SQL 断言映射成一阶逻辑表达式的函数 $\mathcal{T}$ 相区分，我们设将 SQL 断言映射成 WhyML 程序的函数为 $\mathcal{T}'$。以下我们给出函数 $\mathcal{T}'$ 的定义，为了避免和 SQL 断言所用的括号“(”“)”相混淆，我们用 $\mathcal{T}'[]$ 而非 $\mathcal{T}'()$ 来表示；此外，为了便于理解，在之后的 WhyML 代码中，我们把 WhyML 语言表示合取的 \/ 记为 $\wedge$，把表示析取的 /\ 记为 $\vee$，把表示蕴涵（implication）的 $->$ 记为 $\rightarrow$，把表示等价的 $<->$ 记为 $\leftrightarrow$。

$$\begin{array}{l}\mathcal{T}'[\text{CREATE ASSERTION <}\textit{assertion name}\text{>}\\ \text{CHECK <exists predicate>}]\\ \rightsquigarrow\\ \texttt{predicate}\ \text{<}\textit{assertion name}\text{>}\ \text{<parameter>} =\\ \mathcal{T}'[\text{<exists predicate>}]\end{array}$$

$$\begin{array}{rlcl}\mathcal{T}'[\text{EXISTS (} & \text{SELECT *} & \rightsquigarrow & \texttt{exists}\ x_1,\ldots,x_n.\\ & \text{FROM } r_1\ x_1,\cdots,r_n\ x_n & & \texttt{mem}\ x_1\ r_1 \wedge \cdots \wedge \texttt{mem}\ x_n\ r_n\\ & \text{WHERE <search condition>)}] & & \wedge\mathcal{T}'[\text{<search condition>}]\\ \mathcal{T}'[\text{NOT EXISTS (} & \text{SELECT *} & \rightsquigarrow & \texttt{not}\ (\texttt{exists}\ x_1,\ldots,x_n.\\ & \text{FROM } r_1\ x_1,\cdots,r_n\ x_n & & \texttt{mem}\ x_1\ r_1 \wedge \cdots \wedge \texttt{mem}\ x_n\ r_n\\ & \text{WHERE <search condition>)}] & & \wedge\mathcal{T}'[\text{<search condition>}])\end{array}$$

$$\begin{array}{rcl}\mathcal{T}'[\text{<search condition> OR <boolean term>}] & \rightsquigarrow & \mathcal{T}'[\text{<search condition>}] \vee \mathcal{T}'[\text{<boolean term>}]\\ \mathcal{T}'[\text{<boolean term> AND <boolean factor>}] & \rightsquigarrow & \mathcal{T}'[\text{<boolean term>}] \wedge \mathcal{T}'[\text{<boolean factor>}]\\ \mathcal{T}'[\text{NOT (<search condition>)}] & \rightsquigarrow & \texttt{not}\ (\mathcal{T}'[\text{<search condition>}])\end{array}$$

$$\begin{array}{l}\mathcal{T}'[\text{<expression}_1\text{ > <comp op> <expression}_2\text{ >}]\\ \rightsquigarrow\\ \mathcal{T}'[\text{<expression}_1\text{ >}]\ \mathcal{T}'[\text{<comp op>}]\ \mathcal{T}'[\text{<expression}_2\text{ >}]\end{array}$$

$$\begin{array}{rcl}\text{<comp op>} & ::= & = \mid <> \mid < \mid \leq \mid > \mid \geq\\ \mathcal{T}'[\text{<comp op>}] & \rightsquigarrow & \text{<comp op>}\end{array}$$

$$\begin{array}{l}\mathcal{T}'[\text{<expression}_1\text{ > <numerical op> <expression}_2\text{ >}]\\ \rightsquigarrow\\ \mathcal{T}'[\text{<expression}_1\text{ >}]\ \mathcal{T}'[\text{<numerical op>}]\ \mathcal{T}'[\text{<expression}_2\text{ >}]\end{array}$$

$$\begin{array}{rcl}\text{<numerical op>} & ::= & + \mid - \mid * \mid /\\ \mathcal{T}'[\text{<numerical op>}] & \rightsquigarrow & \text{<numerical op>}\\ \mathcal{T}'[\text{<}\textit{const}\text{>}] & \rightsquigarrow & \text{<}\textit{const}\text{>}\\ \mathcal{T}'[x.a] & \rightsquigarrow & x.a\end{array}$$

$\mathcal{T}'$[<expression> BETWEEN <$const_1$ > AND <$const_2$ >]
$\rightsquigarrow$
($\mathcal{T}'$[<expression>] $\geq$ <$const_1$ >) $\wedge$ ($\mathcal{T}'$[<expression>] $\leq$ <$const_2$ >)

$\mathcal{T}'$[<expression> NOT BETWEEN <$const_1$ > AND <$const_2$ >]
$\rightsquigarrow$
($\mathcal{T}'$[<expression>] $<$ <$const_1$ >) $\vee$ ($\mathcal{T}'$[<expression>] $>$ <$const_2$ >)

$\mathcal{T}'$[<expression> IN	$\rightsquigarrow$	`mem` $\mathcal{T}'$[<expression>]
(<$const_1$ >, $\cdots$, <$const_m$ >)]		(Cons <$const_1$ > (Cons $\cdots$(Cons <$const_m$ > Nil) $\cdots$))
$\mathcal{T}'$[<expression> NOT IN	$\rightsquigarrow$	`not` (`mem` $\mathcal{T}'$[<expression>]
(<$const_1$ >, $\cdots$, <$const_m$ >)]		(Cons <$const_1$ > (Cons $\cdots$(Cons <$const_m$ > Nil) $\cdots$)))

设 exp 为一个 SQL 断言子句，则我们有：

$$\mathcal{T}'[(exp)] \quad \rightsquigarrow \quad (\mathcal{T}'[exp])$$

4 SQL 的 INSERT 操作的解释器

对于如下一般形式的 SQL 语言 INSERT 操作：

<insert statement>	::=	INSERT INTO <*table name*> VALUES (<column value list>)
	|	INSERT INTO <*table name*> (<column name list>)
		VALUES (<column value list>)
<column name list>	::=	<*column name*>
	|	<column name list>, <*column name*>
<column value list>	::=	<*column value*>
	|	<column value list>, <*column value*>

我们生成如下的 WhyML 代码：

<insert function>	::=	`let` <*fun name*> <*fun parameters*> =
		{ <precondition> }
		<*target table*> ++ <new tuple>
		{ <postcondition> }
<precondition>	::=	<*assertion name*> <*assertion arguments*>
	|	<precondition> $\wedge$ <*assertion name*> <*assertion arguments*>
<new tuple>	::=	(Cons {| <new column list> |} Nil)
<new column list>	::=	<*column name*> = <*column value*>
	|	<new column list>; <*column name*> = <*column value*>

其中，前置条件 <precondition> 是所有之前定义的 SQL 断言的合取，表示这些 SQL 断言所表示的数据库完整性约束在数据库修改操作执行前都满足。后置条件 <postcondition> 的一般形式与前置条件 <precondition> 的一般形式基本相

同，只不过 <assertion arguments> 中目标关系变量的名称替换为关键字 result，关键字 result 在 WhyML 语言中表示 WhyML 函数执行完后返回的变量。在这段程序中，result 即是<*target table*> ++ <new tuple> 所得到的结果，它表示目标表插入元组后所得到的表。

5　SQL 的 DELETE 操作的解释器

对于如下一般形式的 SQL 语言 DELETE 操作：

<delete statement> ::= DELETE FROM <*target table name*>
[USING <table reference list>]
[WHERE <search condition>]
<table reference list> ::= <*table name*>
| <table reference list>, <*table name*>
<search condition> ::= <boolean term>
| <search condition> OR <boolean term>
<boolean term> ::= <boolean factor>
| <boolean term> AND <boolean factor>
<boolean factor> ::= <predicate>
| [NOT] (<search condition>)
<predicate> ::= <comparison predicate>
| <between predicate>
| <in predicate>

其余非终结符的语法规则与 SQL 断言中的一致，这里从略。

如果 USING 子句和 WHERE 子句不出现在 DELETE 语句中，则表明该 DELETE 操作将删除目标表所有的元组，因此我们生成如下的 WhyML 代码：

<delete function> ::= `let rec` <*fun name*> <*fun parameters*> =
{ true }
`match` <*target table*> `with`
| Nil → Nil
| Cons {| <tuple exp> |} <*left table*> →
(<*fun name*> <*fun arguments*>)
`end`
{ <postcondition> }
<tuple exp> ::= <*column name*> = <column value string>
| <tuple exp>; <*column name*> = <column value string>
<column value string> ::= <*table name*>_<*column name*>_value
<postcondition> ::= <condition> → <consequence>
<condition> ::= <*assertion name*> <*assertion arguments*>
| <condition> ∧ <*assertion name*> <*assertion arguments*>

其中，Cons {| <tuple exp> |} <*left table*> 表示连接元组 {| <tuple exp> |} 和表<*left table*>，变量<*left table*> 表示当前函数中的表<*target table*> 除去元组 {| <tuple exp> |} 后由剩下元组组成的表。函数实参 <fun arguments> 的一般形式与函数形参 <fun parameters> 的一般形式基本相同，除了参数<*target table*> 被变量<*left table*> 所替换。这个函数的语义是，如果当前的表<*target table*> 为空（Nil），则返回空作为结果；否则判断第一个元组是否满足搜索条件，若满足则将其删去、不满足则保留，然后对由剩下元组组成的表 <left table> 继续递归调用该函数。

另外，<condition> 是语义上的前置条件，<consequence> 是语义上的后置条件。这里之所以不把 <condition> 放在程序的前置条件 <precondition>、把 <consequence> 放在程序的后置条件 <postcondition>，是因为而在 WhyML 语言中函数需满足前置条件才会进入函数体，对于某些类型的断言（例如由全称量词限定的断言），当传入的目标表变量<*target table*> 为空时将无法满足，如果我们直接把 <condition> 放在 <precondition> 中，该递归函数最后将无法进入函数体。所以我们在这里把 <precondition> 置为 True，以保证函数体被正常执行，把 <postcondition> 改为蕴涵式，以避免改变其逻辑语义。

<consequence> 的一般形式与前置条件 <condition> 的一般形式基本相同，只不过 <assertion arguments> 中目标关系变量的名称<*target table*> 替换为关键字 result。

如果只有 WHERE 子句出现在 DELETE 语句中，则表明该 DELETE 操作将删除目标表所有满足搜索条件 <search condition> 的元组，而且搜索条件只与目标表有关，因此我们生成如下的 WhyML 代码：

<delete function> ::= `let rec` <*fun name*> <*fun parameters*> =
{ true }
`match` <*target table*> `with`
| Nil → Nil
| Cons {| <tuple exp> |} <*left table*> →
`if` <search condition> `then` (<*fun name*> <*fun arguments*>)
`else` Cons {| <tuple exp> |} (<*fun name*> <*fun arguments*>)
`end`
{ <postcondition> }

如果 USING 子句和 WHERE 子句都出现在 DELETE 语句中，则表明该 DELETE 操作将删除目标表所有满足搜索条件 <search condition> 的元组，而且

搜索条件将涉及到 USING 子句中的多个表，对于这种较复杂的情况，我们将生成与搜索条件相对应的 predicate 及一系列遍历函数和一个执行删除操作的函数。

与搜索条件相对应的 predicate 的一般形式如下：

<sc predicate> ::= `predicate` <sc predicate name> <sc predicate parameters> =
<search condition>

遍历函数 <iter function> 用于在 USING 子句中除遍历目标表以外的表，以得到搜索条件中需要的列变量。

<iter function> ::= `let rec` <*fun name*> <*fun parameters*> =
{ <precondition> }
`match` <*iter table*> `with`
| Nil → False
| Cons {| <tuple exp> |} <*left table*> →
`if` <check condition> `then` True
`else` (<*fun name*> <*fun arguments*>)
`end`
{ <postcondition> }
<precondition> ::= <*assertion name*> <*assertion arguments*>
| <precondition> ∧ <*assertion name*> <*assertion arguments*>

对于检验条件 <check condition>，如果该遍历函数是最先被定义的遍历函数，则 <check condition> 就是搜索条件 <search condition>，否则则为调用前一个被定义的遍历函数，将参数中的 <iter table> 代换为 <left table>。另外，遍历函数的前置条件 <precondition> 是与当前已经遍历过的表相关的断言的合取。

<postcondition> ::= <precondition> ∧ (result = True ↔ <exists statement>)
<exists statement> ::= `exists` <tuple>: <tuple type>.
`mem` <tuple> <*iter table*>∧
(<*sc predicate name*> <*sc predicate arguments*>)

遍历函数被生成之后，我们还需要一个函数来遍历目标表，执行删除操作：

<delete function> ::= `let rec` <*fun name*> <*fun parameters*> =
{ true }
`match` <*target table*> `with`
| Nil → False
| Cons {| <tuple exp> |} <*left table*> →
`if` <check condition> `then` (<*fun name*> <*fun arguments*>)
`else` Cons {| <tuple exp> |} (<*fun name*> <*fun arguments*>)
`end`
{ <postcondition> }
<postcondition> ::= <condition> → <consequence>
<condition> ::= <*assertion name*> <*assertion arguments*>
| <condition> ∧ <*assertion name*> <*assertion arguments*>

6 SQL 的 UPDATE 操作的解释器

对于如下一般形式的 SQL 语言 UPDATE 操作：

<update statement> ::= UPDATE <*target table name*>
SET <set clause list>
[FROM <table reference list>]
[WHERE <search condition>]
<set clause list> ::= <set clause>
<set clause list>, <set clause>
<set clause> ::= <set column> = <*const*>
<set column> ::= <*table name*>.<*attribute name*>

其余非终结符的语法规则与 SQL 断言中的一致，这里从略。

如果 USING 子句和 WHERE 子句不出现在 UPDATE 语句中，则表明该

UPDATE 操作将更新目标表所有的元组，因此我们生成如下的 WhyML 代码：

```
<update function> ::= let rec <fun name> <fun parameters> =
                      { true }
                      match <target table> with
                      | Nil → Nil
                      | Cons {| <old tuple exp> |} <left table> →
                      Cons {| <new tuple exp> |} ( <fun name> <fun arguments> )
                      end
                      { <postcondition> }
      <tuple exp> ::= <column name> = <column value string>
                    | <tuple exp>; <column name> = <column value string>
<column value string> ::= <table>_<column>_<value>
    <postcondition> ::= <condition> → <consequence>
        <condition> ::= <assertion name> <assertion arguments>
                    | <condition> ∧ <assertion name> <assertion arguments>
```

如果只有 WHERE 子句出现在 UPDATE 语句中，则表明该 UPDATE 操作将更新目标表所有满足搜索条件 <search condition> 的元组，而且搜索条件只与目标表有关，因此我们生成如下的 WhyML 代码：

```
<update function> ::= let rec <fun name> <fun parameters> =
                      { true }
                      match <target table> with
                      | Nil → Nil
                      | Cons {| <old tuple exp> |} <left table> →
                      if <search condition>
                      then Cons {| <new tuple exp> |} ( <fun name> <fun arguments> )
                      else Cons {| <old tuple exp> |} ( <fun name> <fun arguments> )
                      end
                      { <postcondition> }
```

如果 USING 子句和 WHERE 子句都出现在 UPDATE 语句中，则表明该 UPDATE 操作将更新目标表所有满足搜索条件 <search condition> 的元组，而且搜索条件将涉及到 USING 子句中的多个表，对于这种较复杂的情况，我们将生成与搜索条件相对应的 predicate 及一系列遍历函数和一个执行更新操作的函数。

与搜索条件相对应的 predicate 的一般形式如下：

```
<sc predicate> ::= predicate <sc predicate name> <sc predicate parameters> =
                   <search condition>
```

遍历函数 <iter function> 用于在 USING 子句中除遍历目标表以外的表，以

得到搜索条件中需要的列变量。

```
<iter function>  ::=  let rec <fun name> <fun parameters> =
                      { <precondition> }
                      match <iter table> with
                      | Nil → False
                      | Cons {| <tuple exp> |} <left table> →
                      if <check condition> then True
                      else ( <fun name> <fun arguments> )
                      end
                      { <postcondition> }
<precondition>   ::=  <assertion name> <assertion arguments>
                   |  <precondition> ∧ <assertion name> <assertion arguments>

<postcondition>  ::=  <precondition> ∧ ( result = True ↔ <exists statement> )
<exists statement>  ::=  exists <tuple>: <tuple type>.
                         mem <tuple> <iter table> ∧ ( <sc predicate> <predicate arguments> )
```

遍历函数被生成之后，我们还需要一个函数来遍历目标表，执行更新操作：

```
<update function>  ::=  let rec <fun name> <fun parameters> =
                        { true }
                        match <target table> with
                        | Nil → False
                        | Cons {| <old tuple exp> |} <left table> →
                        if <search condition>
                        then Cons {| <new tuple exp> |} ( <fun name> <fun arguments> )
                        else Cons {| <old tuple exp> |} ( <fun name> <fun arguments> )
                        end
                        { <postcondition> }
<postcondition>  ::=  <condition> → <consequence>
<condition>      ::=  <assertion name> <assertion arguments>
                   |  <condition> ∧ <assertion name> <assertion arguments>
```

7　本章小结

这一章中我们介绍了 Why3 平台，解释了如何用 Why3 来进行数据库完整性约束的自动验证。我们实现了五个解释器来把 SQL 语句自动转化为 WhyML 程序，这一章对这五个解释器进行了详细的描述：我们给出了相关 SQL 语句的完整语法，根据该语法及上一章所说明的逻辑语义得到 SQL 语句到目标程序的转化规则。根据这些转化规则，我们用 ocamllex 和 ocamlyacc 实现了这五个解释器，从而完成整个验证过程的自动化。

实验结果与分析

在实验中，我们使用 Alt-Ergo[12]、CVC3[3]、Yices[22] 和 Gappa[2] 四个定理自动证明器（prover）作为 Why3 平台的后端（back-end）。

对于任意约束不安全的数据库修改操作，所有的定理自动证明器均给出 Unvalid 或者 Timeout 的结果，表示数据库约束保持的结论无法被证明。

对于约束安全的数据库修改操作，我们取四个定理自动证明器运行结果中最短的运行时间作为该操作的验证时间，则 SQL 语言的 INSERT、DELETE 和 UPDATE 操作的平均验证时间为：

	INSERT 操作	DELETE 操作	UPDATE 操作
平均时间（s）	0.205	2.338	1.149

在我们所测试的例子中，验证时间与相应 SQL 断言的逻辑表达式的复杂度有关。如果我们用合取范式（Conjunctive normal form，简称 CNF）来表达这些 SQL 断言的逻辑表达式，则一般来说合取范式中的子句越多，所需要的验证时间越长。事实上 Why3 平台也提供了 Split 功能，可以把验证目标转化为一系列子目标的合取来分别验证。一般分解出来的子目标越多，总目标所需要的验证时间就越长。此外，验证时间也与数据库修改操作有关。对于涉及多张表的约束安全的 DELETE 或 UPDATE 操作，由于我们需要生成一系列遍历函数，所以其验证时间一般会比只涉及单张表的操作要长。

初期我们使用 0.71 版本的 Why3 平台时，对于数据库完整性约束的逻辑表达式是形如(3.1)的由存在量词限定的谓词公式的情况，定理自动证明器无法判断出约束安全的数据库修改操作。于是我们引入谓词转化器（predicate transformer）方法，找到了一组对 SQL 的 INSERT、DELETE 和 UPDATE 操作的完整性约束等价条件，而且发现对于完整性约束的逻辑表达式是形如(3.2)的由全称量词限定的谓词公式的情况也有类似的结论。这正是在前面第三章中我们所证明的结果。我们测试了对等价条件的验证，结果表明：约束安全和约束不安全的数据库修改操作可以被正确判断出。后来我们使用当时尚未发布的 Why3 平台的 0.72 版本重新进行测试，发现新版的 Why3 平台可以为后端的定理自动证明器生成更加精

确的输入，因而前面被误判的数据库修改操作现在可以被正确判断为约束安全。不过，这两种方法在效率上还是有差别的：

当数据库完整性约束的逻辑表达式是形如(3.1)的由存在量词限定的谓词公式时，这两种方法的性能对比如下：

	INSERT 操作	DELETE 操作	UPDATE 操作
最弱前置条件所需平均时间（s）	0.208	0.119	0.381
等价验证条件所需平均时间（s）	0	0.047	0.054

当数据库完整性约束的逻辑表达式是形如(3.2)的由全称量词限定的谓词公式时，这两种方法的性能对比如下：

	INSERT 操作	DELETE 操作	UPDATE 操作
最弱前置条件所需平均时间（s）	0.199	0.16	0.323
等价验证条件所需平均时间（s）	0.056	0	0.055

以上结果表明，对于(3.1)和(3.2)的情况，验证数据库完整性约束的等价条件要比使用最弱前置条件方法效率更高。这是因为，这组等价条件是根据数据库完整性约束的逻辑结构和数据库修改操作的逻辑语义推导得到的，定理证明器只需验证初始状态的数据库在数据库修改操作执行前满足相应的等价条件，不需要再考虑数据库修改操作在逻辑上对这些等价条件有何影响。

本文总结

1 本文工作总结

我们完成的工作主要有：

1. 补充 [4] 的工作，给出数据库完整性约束规约到 SQL 断言的具体规则，从而证明 SQL 断言在逻辑上足以表达任意的数据库完整性约束。
2. 参考 SQL2003 标准定义了我们所用的 SQL 断言的语法规则，并将 SQL 断言转化为一阶逻辑表达式。
3. 给出了 SQL 语言的 INSERT、DELETE 和 UPDATE 这三大数据库修改操作的逻辑语义。
4. 对数据库完整性约束的逻辑表达式是形如(3.1)或(3.2)的谓词公式的情况给出了谓词转化器的改进，并证明了相关结论。
5. 基于 Why3 验证平台实现了数据库完整性约束的自动验证。
6. 测试了用 Alt-Ergo、CVC3、Yices 和 Gappa 四个定理自动证明器和 Why3 平台来自动验证数据库完整性约束的效率，并在 (3.1)或(3.2)的情况下，对使用最弱前置条件及验证完整性约束的等价条件这两种方法进行了比较。

相比较完整性约束检查（integrity constraint checking）的其它研究成果，我们工作的主要贡献在于：实现了一套切实可行的数据库完整性约束验证的策略，整个验证过程做到了完全的自动化，而且我们的方案是针对被广泛应用的关系数据库（relational databases）和 SQL 语言的，具有较高的实用性；在理论方面，我们分别对数据库断言和数据库修改操作进行了逻辑化描述。针对两种形式的数据库完整性约束，我们找到了更精确的谓词转化器，从而作为我们方案的补充和改进。此外，我们还形式化了 [4] 所提到的数据库完整性约束可以规约到 SQL 断言的结论，这是为了我们课题的简化所做的工作。

2 未来工作展望

我们的工作也存在着一些不足：我们所用的 SQL 断言和数据库修改操作（SQL 语言的 INSERT、DELETE 和 UPDATE 语句）只是 SQL 语言所支持的一部分，有些 SQL 语句不在我们第三章和第四章所给出的语法的范围之内。没能支持完全的 SQL 语言的一大原因是因为某些语义无法完全逻辑形式化，例如 SIMILAR 子句处理的是字符串是否符合某正则表达式的问题。在这一问题仍可以通过对数据类型的逻辑化描述来尝试解决。

对于数据库中被广泛应用的聚合函数（aggregate function），初期我们曾试图对其进行处理，但我们发现它无法用一阶逻辑表达式来表达，而且在实现上也没有切实可行的方案。事实上在数据库完整性约束执行领域，除了 Das[14]、Martinenghi[31] 等少数研究工作之外，聚合函数的处理问题鲜有人提及，而且 Das 等人的工作也都有一定的局限性。这个问题还有待突破性的进一步理论研究。

参 考 文 献

[1] The coq proof assistant. `http://coq.inria.fr/`.

[2] Gappa. `http://gappa.gforge.inria.fr/`.

[3] C. Barrett and C. Tinelli. Cvc3. In *Computer Aided Verification*, pages 298--302. Springer, 2007.

[4] A. Behrend, R. Manthey, and B. Pieper. An amateur's introduction to integrity constraints and integrity checking in sql. In *Proc. BTW*, pages 405--423, 2001.

[5] M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117--127. ACM, 1996.

[6] V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. *Advances in Database Technology—EDBT'98*, pages 309--325, 1998.

[7] F. Bobot, J.C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers . *Boogie*, pages 53--64, 2011.

[8] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Advances in Database Technology—EDBT'88*, pages 488--505, 1988.

[9] U.S. Chakravarthy, J. Grant, and J. Minker. *Foundations of semantic query optimization for deductive databases*. Morgan Kaufmann Publishers Inc., 1988.

[10] I. Chen, A. Min, R. Hull, and D. McLeod. An execution model for limited ambiguity rules and its application to derived data update. *ACM Transactions on Database Systems (TODS)*, 20(4):365--413, 1995.

[11] E.M. Clarke Jr. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *Journal of the ACM (JACM)*, 26(1):129--147, 1979.

[12] S. Conchon and E. Contejean. The alt-ergo automatic theorem prover, 2008. *URL http://altergo. lri. fr*.

[13] L. Console, M.L. Sapino, and D.T. Dupré. The role of abduction in database view updating. *Journal of Intelligent Information Systems*, 4(3):261--280, 1995.

[14] S.K. Das. *Deductive databases and logic programming*. Addison-wesley New York, 1992.

[15] C.J. Date and H. Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley Reading (Ma) et al., 1987.

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337--340, 2008.

[17] H. Decker. An extension of sld by abduction and integrity maintenance for view updating in deductive databases. In *Proc. of the 1996 International Conference on Logic Programming*, pages 157--169, 1996.

[18] H. Decker. One abductive logic programming procedure for two kind of updates. In *Proc. Workshop DINAMICS*, volume 97, 1997.

[19] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365--473, 2005.

[20] E.W. Dijkstra, E.W. Dijkstra, E.W. Dijkstra, and E.W. Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, NJ, 1976.

[21] E.W. Dijkstra and C.S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[22] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[23] G. Gardarin and M.A. Melkanoff. Proving consistency of database transactions. In *Proceedings of the fifth international conference on Very Large Data Bases-Volume 5*, pages 291--298. VLDB Endowment, 1979.

[24] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. *Logics for databases and information systems*, pages 265--306, 1998.

[25] J. Grant and J. Minker. *Integrity constraints in knowledge based systems*. University of Maryland, 1989.

[26] J. Grant and J. Minker. The impact of logic programming on databases. *Communications of the ACM*, 35(3):66--81, 1992.

[27] M. Lawley. Transaction safety in deductive object-oriented databases. *Deductive and Object-Oriented Databases*, pages 395--410, 1995.

[28] M. Lawley, R. Topor, and M. Wallace. Using weakest preconditions to simplify integrity constraint checking. In *Proceedings of the Australian Database Conference*, pages 161--170, 1993.

[29] J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *The Journal of Logic Programming*, 2(2):93--109, 1985.

[30] J. Lobo and G. Trajcevski. Minimal and consistent evolution of knowledge bases. *JOURNAL OF APPLIED NONCLASSICAL LOGICS*, 7:117--146, 1997.

[31] D. Martinenghi. *Advanced techniques for efficient data integrity checking*. PhD thesis, Citeseer, 2005.

[32] W.W. McCune and L.J. Henschen. Maintaining state constraints in relational databases: A proof theoretic basis. *Journal of the ACM (JACM)*, 36(1):46--68, 1989.

[33] J.M. Nicolas. Logic for improving integrity checking in relational data bases . *Acta Informatica*, 18(3):227--253, 1982.

[34] X. Qian. An axiom system for database transactions. *Information Processing Letters*, 36(4):183--189, 1990.

[35] X. Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems (TODS)*, 18(4):626--677, 1993.

[36] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems (TODS)*, 14(3):322--368, 1989.

[37] E. Simon and P. Valduriez. Design and implementation of an extendible integrity subsystem. In *ACM SIGMOD Record*, volume 14, pages 9--17. ACM, 1984.

[38] M. Wallace. Compiling integrity checking into update procedures. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 903--908, 1991.

[39] B. Wuthrich. On updates and inconsistency repairing in knowledge bases. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 608--615. IEEE, 1993.

致 谢

这个课题能够顺利完成，首先要感谢我在法国巴黎十一大 LRI 实验室的导师 Véronique Benzaken 和 Evelyne Contejean 女士。在她们的指导下，我加深了对该课题背景理论的理解，并得以给出相关结论较为严谨的表述或推导。在分析实验现象、提出改善方案方面，她们也给了我相当多非常宝贵的意见。同时我也要感谢 LRI 实验室 ProVal 组各位成员在我法国交流期间的热心帮助，特别是 Why3 平台的开发者和维护者 Andrei Paskevich 先生，他数次从百忙中抽出时间耐心解答了我许多关于 Why3 的疑问。

感谢我在 Eagle 实验室的导师宋明黎老师，在我刚刚踏入计算机科学的研究领域时，宋老师引领我在一个切实可行的方向逐步深入探究，这一过程我不仅收获了知识，而且也锻炼了科研能力。宋老师治学认真踏实、严谨细致，使我受益匪浅。这两年来，Eagle 实验室 VIPA 组和 MMG 组的学长学姐们给予了我很多帮助，在此我也对他们表示衷心感谢。

最后感谢陪我走过了这四年的所有老师和同学，是你们让我的大学时光如此丰富充实、如此绚丽多彩；感谢父亲母亲，我的每一分成就背后都有着你们对我数十年如一日的关心和支持。

袁帅

于求是园

附 录

本项目的关键源代码可以通过如下命令获取：

git clone git://github.com/yszheda/assertion-verification.git

本科生毕业论文(设计)任务书

一、题目: _____

二、指导教师对毕业论文(设计)的进度安排及任务要求:

起讫日期200　年　月　日至200　年　月　日

指导教师(签名) _____　职称 _____

三、系或研究所审核意见:

负责人(签名)_____

年　月　日

毕 业 论 文（设计）考 核

一、指导教师对毕业论文(设计)的评语：

指导教师(签名) _____

年　月　日

二、答辩小组对毕业论文(设计)的答辩评语及总评成绩：

成绩比例	文献综述占 (10%)	开题报告占 (20%)	外文翻译占 (10%)	毕业论文 (设计) 质量及答辩占 (60%)	总评成绩
分值					

答辩小组负责人(签名) _____

年　月　日